NEA Report

Christopher Lang 1132
Typesetting System
2022-2023

Contents

1 Analysis....1
  1.1 Problem Statement....1
  1.2 Problem Background and Analysis....1
  1.3 Intended End-User....2
  1.4 Third Party....2
  1.5 Research and Modelling....2
  1.6 Objectives....4
  1.7 Prototype....5
2 Documented Design....6
  2.1 Target System....6
  2.2 File Structure and Source Control....6
  2.3 Executable Files....7
  2.4 File Formats....8
  2.5 Data Structures....12
  2.6 Key Algorithms....14
  2.7 Example Document....16
3 Technical Solution....18
  3.1 Techniques Used....18
  3.2 Source Code....19
    Makefile....19
    tw.h....19
    tw.c....21
    line_break.c....27
    dbuffer.c....37
    record.c....38
    pdf.c....40
    jpeg.c....48
    ttf.c....49
    utils.c....54
    utils.py....55
    contents.py....56
    markup_raw.py....57
    markup_text.py....57
    pager.py....61
4 Testing....67
5 Evaluation....72
  5.1 Requirements Met....72
  5.2 Improvements....72
  5.3 Feedback....72

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 * .

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' can't 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).

* *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*

* *GNU Troff (Groff) online manual version 1.22.4.*

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 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?

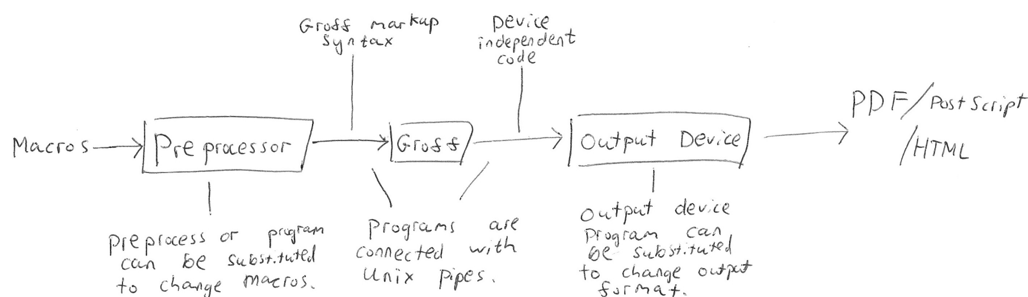
Anthony I would do unspeakable thinks to have access to a program/feature that would achieve what you just described.

From this conversation, it is clear that a document peperation system that is able to intergrate into a Unix envoronment (and therefore be able to automatically typeset program source code) would beinfit my third party.

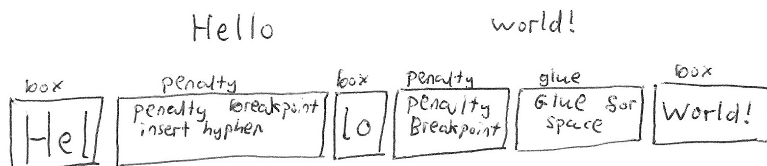
1.5 Research and Modelling

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 reference manual 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.



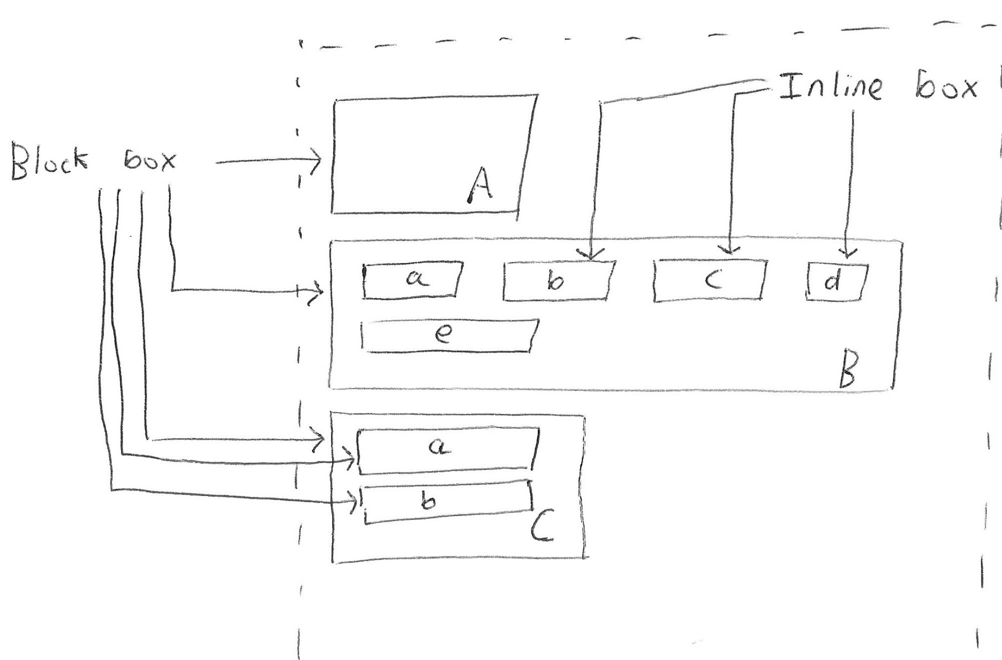
TeX + LaTeX. TeX is a typesetting system first released in 1978. I used a derivative of TeX, LaTeX, to write a number of documents for a school project. My frustration with LaTeX's hard-to-read syntax and confusing extension system was the initial impetus to make an alternative. Donald E Knuth's TeXbook 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' 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 * and a blog about building a browser engine * 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.

* *How Browsers Work: https://web.dev/howbrowserswork/*

* *Lets Build a Browser Engine: https://limpet.net/mbrubeck/2014/08/08/toy-layout-engine-1.html*



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

TrueType Reference Manual. * 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. 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. (Introduction to Algorithms - Third Eddition 2009) 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' 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.

* *https://developer.apple.com/fonts/TrueType-Reference-Manual/*

```
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 space 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 Target System

This project is intended to work only on Unix-like machines 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. The 'demo' directory contain shell scripts and a video which demonstrates the completion of objectives.

```
typewriter
|-- LICENSE
|-- Makefile
|-- README
|-- 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
|   |-- 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
|-- line_break
|-- line_break.c
|-- markup_raw.py
|-- markup_text.py
|-- pager.py
|-- pdf.c
|-- 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
|-- test
|   |-- doc.sh
|   |-- fonts -> ../fonts
|   |-- peppers.jpg
|   `-- typeface -> ../typeface
|-- ttf.c
|-- tw
|-- tw.c
|-- tw.h
|-- typeface
|-- utils.c
`-- utils.py

8 directories, 44 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 standard input 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 standard input, splits this content into *pages* which are written to standard output. *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 standard input into *content* which is written to standard output. 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 standard input, calculates the optimal line breaks and writes *content* to standard output. 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 standard input and write *content* to standard output. 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 appears 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. The *contents* file is generated by *pager* and is the input to the *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 following optional break. This additional 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 font names to a font files. 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 the 0x00 character. *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

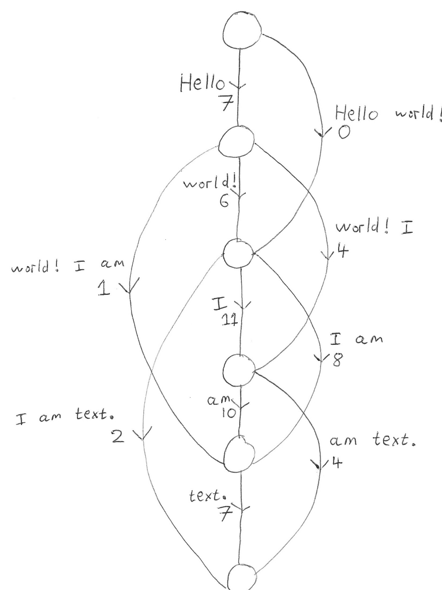
2.6.1 Line Breaker

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

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. 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

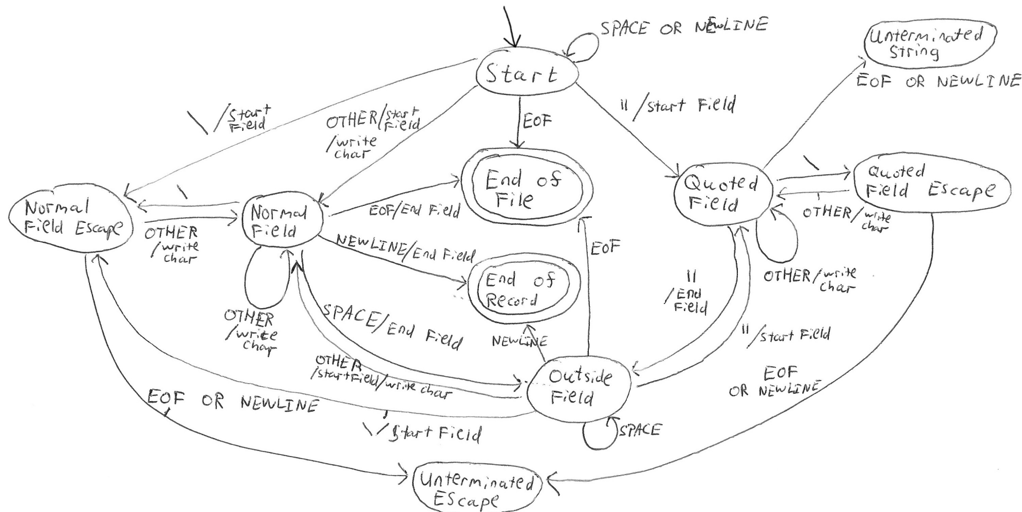
The python implantation of record parsing uses the following regular expression which matches each field in a record:

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

This regex can be split into two sections by the pipe (OR) character. The first section matches a space separated field: match a non-whitespace, non-quotation mark character followed by any non-whitespace characters. The second section matches a quoted field: match a quotation mark followed by any number of characters until a quotation mark which is not preceded by a backslash is found.

To make parsing more efficient in the C code, I will write my own parser. Characters

are to be parsed sequentially using the following state machine of 10 states.



Parsing begins in the 'Start' state. 'EOF' indicates 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. Other complex file formats are described in Documented Deisign and used throughout the codebase.

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 subtype of TextStream. Subtypes can change how a stream of text is 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 /*
007  * tw.c
008  * tw is short for typewriter. This file reads _pages_ from stdin and converts
009  * this to PDF.
010  */
011
012 #include <stdio.h>
013 #include <string.h>
014 #include <unistd.h>
015
016 #include "tw.h"
017
018 /* A page's entire text content and its current state. */
019 struct text_content {
020   int x, y, font_size, word_spacing;
021   struct dbuffer buffer;
022   char font_name[256];
023 };
024
025 static void add_page(FILE *pdf_file, int obj_parent,
026     struct pdf_xref_table *xref, struct pdf_page_list *page_list,
027     struct dbuffer *text_content, struct dbuffer *graphic_content);
028 static void write_pdf_escaped_string(struct dbuffer *buffer,
029     const char *string);
030 static int parse_text(FILE *input, int x, int y,
031     struct text_content *text_content, struct pdf_resources *resources);
032 static int parse_graphic(FILE *input, int origin_x, int origin_y,
033     struct text_content *text_content, struct dbuffer *graphic_content,
034     struct pdf_resources *resources);
035
036 /*
037  * Records are parsed in multiple functions, in order to avoid reallocating
038  * record memory each time, _record_ is kept as a static global variable.
039  * It is reset before each use so its persistent state does not matter.
040  * Use of record is not thread-safe.
041  */
042 static struct record record;
043
044 /* Add a page to _pdf_file_ with _text_content_ and _graphic_content_. */
045 static void
046 add_page(FILE *pdf_file, int obj_parent, struct pdf_xref_table *xref,
047     struct pdf_page_list *page_list, struct dbuffer *text_content,
048     struct dbuffer *graphic_content)
049 {
050   int obj_content, obj_page;
051   long length;
052   /* Allocate pdf indirecto objects for the page and its conetnt. */
053   obj_content = allocate_pdf_obj(xref);
054   obj_page = allocate_pdf_obj(xref);
055
056   /* ET is the PDF control sequence for ending text content. */
057   dbuffer_printf(text_content, "ET\n");
```

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

```
132       }
133       if (!is_font_name_valid(record.fields[1])) {
134         fprintf(stderr, "Font name contains illegal characters: '%s'\n",
135             record.fields[1]);
136         continue;
137       }
138       if (str_to_int(record.fields[2], &arg1)) {
139         fprintf(stderr, "Text FONT command's 2nd argument must be integer.\n");
140         continue;
141       }
142       /* Update the current font name and size. */
143       strcpy(font_name, record.fields[1]);
144       font_size = arg1;
145       /* Tell _resources_ that this font is being used. */
146       include_font_resource(resources, font_name);
147       continue;
148     }
149     if (strcmp(record.fields[0], "SPACE") == 0) {
150       /* SPACE [WORD_SPACING] */
151       if (record.field_count != 2) {
152         fprintf(stderr, "Text SPACE command must take 1 arguments.\n");
153         continue;
154       }
155       if (str_to_int(record.fields[1], &arg1)) {
156         fprintf(stderr, "Text SPACE command's argument must be integer.\n");
157         continue;
158       }
159       /* Update current word spacing. */
160       word_spacing = arg1;
161       continue;
162     }
163     if (strcmp(record.fields[0], "STRING") == 0) {
164       /* STRING [STRING] */
165       if (record.field_count != 2) {
166         fprintf(stderr, "Text STRING command must take 1 argument.\n");
167         continue;
168       }
169       /* If the font has not been set. */
170       if (*font_name == '\0') {
171         fprintf(stderr, "Text must specify font.\n");
172         continue;
173       }
174       /* If a different font is used for this string. */
175       if (strcmp(text_content->font_name, font_name)
176           || text_content->font_size != font_size) {
177         /* Change the active font. */
178         dbuffer_printf(&text_content->buffer, " /%s %d Tf", font_name,
179             font_size);
180         strcpy(text_content->font_name, font_name);
181         text_content->font_size = font_size;
182       }
183       /* If a different word spacing is used for this string. */
184       if (word_spacing != text_content->word_spacing) {
185         /*
186          * Change the current word spacing. Divide by 1000 to convert to point.
187          */
188         dbuffer_printf(&text_content->buffer, " %f Tw",
189             (float)word_spacing / 1000);
190         text_content->word_spacing = word_spacing;
191       }
192       /* Separate this text command from the last. */
193       dbuffer_putc(&text_content->buffer, ' ');
194       /* Write the string command. */
195       write_pdf_escaped_string(&text_content->buffer, record.fields[1]);
196       dbuffer_printf(&text_content->buffer, " Tj");
197       continue; /* Next record. */
198     }
199     /* If the first field was not recognised. */
200     fprintf(stderr, "Invalid text command: '%s'\n", record.fields[0]);
201   }
202   dbuffer_putc(&text_content->buffer, '\n');
203   return 0;
204 }
205
```

```
206 /* Parse graphic mode _pages_ content. */
207 static int
208 parse_graphic(FILE *input, int origin_x, int origin_y,
209     struct text_content *text_content, struct dbuffer *graphic_content,
210     struct pdf_resources *resources)
211 {
212   int parse_result, x, y, arg1, arg2, image_id;
213   x = origin_x;
214   y = origin_y;
215   for (;;) {
216     parse_result = parse_record(input, &record);
217     if (parse_result == EOF) {
218       fprintf(stderr, "Graphic not ended before end of file.\n");
219       return 1;
220     }
221     if (parse_result) /* If failed to parse record then skip it. */
222       continue;
223     if (strcmp(record.fields[0], "END") == 0)
224       /* Exit graphic mode. */
225       break;
226     if (strcmp(record.fields[0], "MOVE") == 0) {
227       /* MOVE [X_OFFSET] [Y_OFFSET] */
228       if (record.field_count != 3) {
229         fprintf(stderr, "Graphic MOVE command must take 2 arguments.\n");
230         continue;
231       }
232       /* Try to convert the offset arguments to integers. */
233       if (str_to_int(record.fields[1], &arg1)
234           || str_to_int(record.fields[2], &arg2)) {
235         fprintf(stderr, "Graphic MOVE command takes only integer arguments.\n");
236         continue;
237       }
238       /* Update the current position. */
239       x = origin_x + arg1;
240       y = origin_y + arg2;
241       continue;
242     }
243     if (strcmp(record.fields[0], "START") == 0) {
244       /* START [GRAPHIC/TEXT] */
245       /* A graphic may contain text content or another graphic. */
246       if (record.field_count != 2) {
247         fprintf(stderr, "START command must take one argument.\n");
248         return 1;
249       }
250       if (strcmp(record.fields[1], "GRAPHIC") == 0) {
251         if (parse_graphic(input, x, y, text_content, graphic_content,
252             resources))
253           return 1;
254         continue;
255       }
256       if (strcmp(record.fields[1], "TEXT") == 0) {
257         if (parse_text(input, x, y, text_content, resources))
258           return 1;
259         continue;
260       }
261       /* The first argument was not recognised. */
262       fprintf(stderr, "Invalid graphic START command argument: '%s'\n",
263           record.fields[1]);
264       return 1; /* Error. */
265     }
266     if (strcmp(record.fields[0], "IMAGE") == 0) {
267       /* IMAGE [WIDTH] [HEIGHT] [IMAGE_FILE_NAME] */
268       if (record.field_count != 4) {
269         fprintf(stderr, "IMAGE command must take 3 arguments.\n");
270         continue;
271       }
272       /* Convert width and height to integers. */
273       if (str_to_int(record.fields[1], &arg1)
274           || str_to_int(record.fields[2], &arg2)) {
275         fprintf(stderr,
276             "IMAGE command's 1st and 2nd arguments must be integer.\n");
277         continue;
278       }
279       /* Tell _resources_ that we are using this image. */
```

```
280       image_id = include_image_resource(resources, record.fields[3]);
281       /* Push graphic state. */
282       dbuffer_printf(graphic_content, "q\n");
283       /* Set graphic transformation matrix and draw image. */
284       dbuffer_printf(graphic_content, "  %d 0 0 %d %d %d cm\n", arg1, arg2, x,
285           y);
286       dbuffer_printf(graphic_content, "  /Img%d Do\n", image_id);
287       /* Pop graphic state. */
288       dbuffer_printf(graphic_content, "Q\n");
289       continue;
290     }
291     fprintf(stderr, "Invalid graphic command: '%s'\n", record.fields[0]);
292   }
293   return 0;
294 }
295
296 /* Read _pages_ format from _pages_file_ and write PDF to _pdf_file_. */
297 int
298 print_pages(FILE *pages_file, FILE *typeface_file, FILE *pdf_file)
299 {
300   int ret, parse_result;
301   int obj_resources, obj_page_list, obj_catalog;
302   struct pdf_xref_table xref_table;
303   struct pdf_page_list page_list;
304   struct pdf_resources resources;
305   struct text_content text_content;
306   struct dbuffer graphic_content;
307   ret = 0;
308   /* Initialise resources. */
309   init_pdf_xref_table(&xref_table);
310   init_pdf_page_list(&page_list);
311   init_pdf_resources(&resources);
312   /* Allocate essential indirect objects. */
313   obj_resources = allocate_pdf_obj(&xref_table);
314   obj_page_list = allocate_pdf_obj(&xref_table);
315   obj_catalog = allocate_pdf_obj(&xref_table);
316   /* Write PDF header. */
317   pdf_write_header(pdf_file);
318
319   text_content.x = 0;
320   text_content.y = 0;
321   text_content.font_size = 0;
322   text_content.font_name[0] = '\0';
323   dbuffer_init(&text_content.buffer, 1024 * 32, 1024 * 32);
324   /* Begin Text. */
325   dbuffer_printf(&text_content.buffer, "BT\n");
326   dbuffer_init(&graphic_content, 1024 * 4, 1024 * 4);
327   init_record(&record);
328   /* Parse each record in _pages_ document mode. */
329   for (;;) {
330     parse_result = parse_record(pages_file, &record);
331     if (parse_result == EOF) /* Stop at end of file. */
332       break;
333     if (parse_result) /* If failed to parse record then skip it. */
334       continue;
335     if (strcmp(record.fields[0], "START") == 0) {
336       if (record.field_count != 2) {
337         fprintf(stderr, "Pages START command must take 1 argument.\n");
338         /* ret=1 because this is an error and we will return 1. */
339         ret = 1;
340         break;
341       }
342       if (strcmp(record.fields[1], "PAGE") == 0) {
343         /* Try to parse this PAGE's graphic. */
344         if (parse_graphic(pages_file, 0, 0, &text_content, &graphic_content,
345               &resources)) {
346           ret = 1;
347           break;
348         }
349         /* Add the page to the pdf file. */
350         add_page(pdf_file, obj_page_list, &xref_table, &page_list,
351             &text_content.buffer, &graphic_content);
352         /* Reset the text and graphic for the next page. */
353         text_content.x = 0;
```

```
354         text_content.y = 0;
355         text_content.font_size = 0;
356         text_content.font_name[0] = '\0';
357         text_content.buffer.size = 0;
358         dbuffer_printf(&text_content.buffer, "BT\n");
359         graphic_content.size = 0;
360         graphic_content.data[0] = '\0';
361         continue;
362       }
363       fprintf(stderr, "Invalid document START command argument: '%s'\n",
364           record.fields[1]);
365       ret = 1;
366       break;
367     }
368   }
369   /* Cleanup resources. */
370   dbuffer_free(&text_content.buffer);
371   free_record(&record);
372
373   /* Add resources to PDF file and close it. */
374   pdf_add_resources(pdf_file, typeface_file, obj_resources, &resources,
375       &xref_table);
376
377   pdf_start_indirect_obj(pdf_file, &xref_table, obj_page_list);
378   pdf_write_pages(pdf_file, obj_resources, page_list.page_count,
379       page_list.page_objs);
380   pdf_end_indirect_obj(pdf_file);
381
382   pdf_start_indirect_obj(pdf_file, &xref_table, obj_catalog);
383   pdf_write_catalog(pdf_file, obj_page_list);
384   pdf_end_indirect_obj(pdf_file);
385
386   pdf_add_footer(pdf_file, &xref_table, obj_catalog);
387   free_pdf_page_list(&page_list);
388   free_pdf_resources(&resources);
389   free_pdf_xref_table(&xref_table);
390   return ret;
391 }
392
393 int
394 main(int argc, char **argv)
395 {
396   int opt;
397   const char *output_file_name;
398   FILE *pages_file, *typeface_file, *pdf_file;
399   /* Default output to ./output.pdf */
400   output_file_name = "./output.pdf";
401   /* Parse command line arguments. */
402   while ( (opt = getopt(argc, argv, "o:")) != -1) {
403     switch (opt) {
404       case 'o':
405         output_file_name = optarg;
406         break;
407       default:
408         fprintf(stderr, "Usage: %s [-o OUTPUT_FILE]\n", argv[0]);
409         return 1;
410     }
411   }
412   pages_file = stdin;
413   /*
414    * Typeface file is hardcoded to be called 'typeface'. This is a design
415    * decision: if we could call the typeface file anything we wanted, then
416    * other programs part of the typesetting process would struggle to locate
417    * it. Just like 'Makefile', 'typeface' is a hard-coded file name.
418    */
419   typeface_file = fopen("./typeface", "r");
420   pdf_file = fopen(output_file_name, "w");
421   if (typeface_file == NULL) {
422     fprintf(stderr, "tw: Failed to open typeface file.\n");
423     return 1;
424   }
425   if (pdf_file == NULL) {
426     fprintf(stderr, "tw: Failed to open output file '%s'.\n", output_file_name);
427     return 1;
```

```
428   }
429   print_pages(pages_file, typeface_file, pdf_file);
430   fclose(pages_file);
431   fclose(typeface_file);
432   fclose(pdf_file);
433   return 0;
434 }
```

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 /* Text align modes. */
016 enum align {
017   ALIGN_LEFT,
018   ALIGN_RIGHT,
019   ALIGN_CENTRE,
020   ALIGN_JUSTIFIED,
021 };
022
023 /* Types for (struct gizmo)->type. */
024 enum gizmo_type {
025   GIZMO_TEXT,
026   GIZMO_BREAK,
027   GIZMO_MARK,
028 };
029
030 /* Linked list, each struct maps one _font_name_ to one _font_info_. */
031 struct typeface {
032   /* Font names cant be longer than 255 characters. */
033   char font_name[256];
034   struct font_info font_info;
035   /* Next font in the linked list. */
036   struct typeface *next;
037 };
038
039 /* A text style. */
040 struct style {
041   int font_size;
042   char font_name[256];
043 };
044
045 /* Polymorphic linked list. _type_ defines the meaning of _char _[]_ bytes. */
046 struct gizmo {
047   int type;
048   struct gizmo *next;
049   /* Any number of bytes can follow (gizmo specific data). */
050   char _[];
051 };
052
053 struct text_gizmo {
054   int type; /* GIZMO_TEXT */
055   struct gizmo *next;
056   int width;
057   struct style style;
058   /* A string of any number of bytes can follow. */
059   char string[];
060 };
061
062 struct break_gizmo {
063   int type; /* GIZMO_BREAK */
```

```
064   struct gizmo *next;
065   int force_break, total_penalty, spacing, selected;
066   struct break_gizmo *best_source;
067   struct style style;
068   int no_break_width, at_break_width;
069   /* Pointers to inside _strings_. */
070   char *no_break, *at_break;
071   /* Any number of bytes can follow (for storing strings). */
072   char strings[];
073 };
074
075 struct mark_gizmo {
076   int type; /* GIZMO_MARK */
077   struct gizmo *next;
078   char string[];
079 };
080
081 static struct typeface *open_typeface(FILE *typeface_file);
082 static void free_typeface(struct typeface *typeface);
083 static int get_text_width(const char *string, const struct style *style,
084     const struct typeface *typeface);
085 static struct gizmo *parse_gizmos(FILE *file, const struct typeface *typeface);
086 static void free_gizmos(struct gizmo *gizmo);
087 static void consider_breaks(struct gizmo *gizmo, int source_penalty,
088     struct break_gizmo *source, int line_width);
089 static void optimise_breaks(struct gizmo *gizmo, int line_width);
090 static void print_text(struct dbuffer *buffer, struct style *style,
091     const struct style *new_style, const char *string, int *spaces);
092 static void print_gizmos(FILE *output, struct gizmo *gizmo, int line_width,
093     int align);
094
095 /*
096  * Parse the _typeface_file_ and return a linked list of fonts in the typeface.
097  */
098 static struct typeface *
099 open_typeface(FILE *typeface_file)
100 {
101   struct typeface *first_font;
102   struct typeface **next_font;
103   int parse_result;
104   struct record record;
105   FILE *font_file;
106   first_font = NULL;
107   next_font = &first_font;
108   init_record(&record);
109   /* Loop through all records in the typeface file. */
110   for (;;) {
111     parse_result = parse_record(typeface_file, &record);
112     /* If end of file reached, stop. */
113     if (parse_result == EOF)
114       break;
115     /* If failed to parse this record, skip it. */
116     if (parse_result)
117       continue;
118     if (record.field_count != 2) {
119       fprintf(stderr, "Typeface records must have exactly 2 fields.");
120       continue;
121     }
122     /*
123      * Fonts can't be longer than 255 characters because that would overflow
124      * their buffer.
125      */
126     if (strlen(record.fields[0]) >= 256) {
127       fprintf(stderr,
128           "Typeface file contains font name that is too long '%s'.\n",
129           record.fields[0]);
130       /* Go to next record. */
131       continue;
132     }
133     /* Check the font name does not contain any illegal characters. */
134     if (!is_font_name_valid(record.fields[0])) {
135       fprintf(stderr, "Typeface file contains invalid font name '%s'.\n",
136           record.fields[0]);
137       continue;
```

```
138     }
139     /* Open the font file that the typeface record references. */
140     font_file = fopen(record.fields[1], "r");
141     if (font_file == NULL) {
142       fprintf(stderr, "Failed to open ttf file '%s': %s\n",
143           record.fields[1], strerror(errno));
144       continue;
145     }
146     /* Allocate space on the heap for the next node in the linked list. */
147     *next_font = xmalloc(sizeof(struct typeface));
148     /* Copy the font name into the new node. */
149     strcpy((*next_font)->font_name, record.fields[0]);
150     /* Try to parse the font file to get the font info. */
151     if (read_ttf(font_file, &(*next_font)->font_info)) {
152       fprintf(stderr, "Failed to parse ttf file: '%s'\n", record.fields[1]);
153       free(*next_font);
154     } else {
155       /* Add this font to the linked list. */
156       next_font = &(*next_font)->next;
157     }
158     /* Close the font file. */
159     fclose(font_file);
160   }
161   /* End the linked list here. */
162   *next_font = NULL;
163   free_record(&record);
164   /* Return the first node in the linked list. */
165   return first_font;
166 }
167
168 /* Free a typeface linked list. */
169 static void
170 free_typeface(struct typeface *typeface)
171 {
172   struct typeface *next;
173   /* Loop through every node in the linked list and free it. */
174   while (typeface) {
175     next = typeface->next;
176     free(typeface);
177     typeface = next;
178   }
179 }
180
181 /*
182  * Compute the width of _string_ in _style_ with _typeface_ measured in
183  * thousandths of points.
184  */
185 static int
186 get_text_width(const char *string, const struct style *style,
187     const struct typeface *typeface)
188 {
189   int width;
190   const struct typeface *font;
191   font = typeface;
192   /* Loop over fonts until a name matches. */
193   while (font && strcmp(font->font_name, style->font_name))
194     font = font->next;
195   /* If no names matched. */
196   if (font == NULL) {
197     fprintf(stderr, "Typeface does not include font: '%s'\n", style->font_name);
198     font = typeface;
199   }
200   width = 0;
201   /* Loop over every character in string and add it's width. */
202   for (; *string; string++)
203     width += font->font_info.char_widths[(unsigned char)*string];
204   return width * style->font_size;
205 }
206
207 /*
208  * Parse text gizmos from the text specification _file_ and return them as a
209  * linked list.
210  */
211 static struct gizmo *
```

```
212 parse_gizmos(FILE *file, const struct typeface *typeface)
213 {
214   struct gizmo *first_gizmo;
215   struct gizmo **next_gizmo;
216   struct style current_style;
217   int parse_result, arg1;
218   struct record record;
219   /* An empty linked list is just a NULL pointer. */
220   first_gizmo = NULL;
221   /* _next_gizmo_ points to the end of the linked list. */
222   next_gizmo = &first_gizmo;
223   /* Initialise _current_style_. */
224   current_style.font_name[0] = '\0';
225   current_style.font_size = 0;
226   init_record(&record);
227   /* For each record in _file_. */
228   for (;;) {
229     parse_result = parse_record(file, &record);
230     /* If reach end of file then stop. */
231     if (parse_result == EOF)
232       break;
233     /* If failed to parse record then skip it. */
234     if (parse_result)
235       continue;
236     /* The first field in the record is the command type. */
237     if (strcmp(record.fields[0], "STRING") == 0) {
238       /* STRING [STRING] */
239       if (record.field_count != 2) {
240         fprintf(stderr, "Text STRING command must have 1 option.\n");
241         continue;
242       }
243       /* If the font name is still empty from initialisation. */
244       if (current_style.font_name[0] == '\0') {
245         fprintf(stderr,
246             "Text STRING command can't be called without FONT set.\n");
247         /* Next record. */
248         continue;
249       }
250       /*
251        * Allocate space for the next gizmo in the linked list. It must contain
252        * the text_gizmo struct followed by the string specified in the record.
253        */
254       *next_gizmo = xmalloc(sizeof(struct text_gizmo)
255           + strlen(record.fields[1]) + 1);
256       /* Initialise the new text gizmo.  */
257       (*next_gizmo)->type = GIZMO_TEXT;
258       (*next_gizmo)->next = NULL;
259       /* Find the string width. */
260       ((struct text_gizmo *)*next_gizmo)->width
261         = get_text_width(record.fields[1], &current_style, typeface);
262       ((struct text_gizmo *)*next_gizmo)->style = current_style;
263       /* Copy the string into the gizmo. */
264       strcpy(((struct text_gizmo *)*next_gizmo)->string, record.fields[1]);
265       /* Update _next_gizmo_ to point to the end of the linked list. */
266       next_gizmo = &(*next_gizmo)->next;
267       /* Next record. */
268       continue;
269     }
270     if (strcmp(record.fields[0], "FONT") == 0) {
271       /* FONT [FONT_NAME] [FONT_SIZE] */
272       if (record.field_count != 3) {
273         fprintf(stderr, "Text FONT command must have 2 options.\n");
274         continue;
275       }
276       /* Validate font name. */
277       if (strlen(record.fields[1]) >= 256) {
278         fprintf(stderr, "Text font name too long: '%s'\n", record.fields[1]);
279         continue;
280       }
281       if (!is_font_name_valid(record.fields[1])) {
282         fprintf(stderr, "Text font name contains illegal characters: '%s'\n",
283             record.fields[1]);
284         continue;
285       }
```

```
286       /* Update the current style. */
287       strcpy(current_style.font_name, record.fields[1]);
288       if (str_to_int(record.fields[2], &current_style.font_size)) {
289         fprintf(stderr, "Text FONT command's 2nd option must be integer.\n");
290         current_style.font_size = 12;
291       }
292       /* Next record. */
293       continue;
294     }
295     if (strcmp(record.fields[0], "OPTBREAK") == 0) {
296       /* OPTBREAK [NO_BREAK_STRING] [AT_BREAK_STRING] [SPACING] */
297       if (record.field_count != 4) {
298         fprintf(stderr, "Text OPTBREAK command must have 3 options.\n");
299         continue;
300       }
301       /* Convert the [SPACING] option to an integer. */
302       if (str_to_int(record.fields[3], &arg1)) {
303         fprintf(stderr,
304             "Text OPTBREAK command's 3rd option must be integer.\n");
305         continue;
306       }
307       /*
308        * Allocate memory for the new gizmo. It must contain the break_gizmo
309        * struct, the [NO_BREAK_STRING] and the [AT_BREAK_STRING].
310        */
311       *next_gizmo = xmalloc(sizeof(struct break_gizmo)
312           + strlen(record.fields[1])
313           + strlen(record.fields[2]) + 2);
314       /* Initialise the new break gizmo. */
315       (*next_gizmo)->type = GIZMO_BREAK;
316       (*next_gizmo)->next = NULL;
317       ((struct break_gizmo *)*next_gizmo)->spacing = arg1;
318       /* And OPTBREAK does not force a break here. */
319       ((struct break_gizmo *)*next_gizmo)->force_break = 0;
320       /* We do not yet know if this gizmo will be selected. */
321       ((struct break_gizmo *)*next_gizmo)->selected = 0;
322       /*
323        * Set the total penalty to INT_MAX so the first route to this break
324        * overwrites it.
325        */
326       ((struct break_gizmo *)*next_gizmo)->total_penalty = INT_MAX;
327       /* _best_source_ is used for the shortest path algorithm. */
328       ((struct break_gizmo *)*next_gizmo)->best_source = NULL;
329       ((struct break_gizmo *)*next_gizmo)->style = current_style;
330       /* Get the text width for the [NO_BREAK_STRING] and [AT_BREAK_STRING]. */
331       ((struct break_gizmo *)*next_gizmo)->no_break_width
332         = get_text_width(record.fields[1], &current_style, typeface);
333       ((struct break_gizmo *)*next_gizmo)->at_break_width
334         = get_text_width(record.fields[2], &current_style, typeface);
335       /*
336        * Set the _no_break_ and _at_break_ pointers to point into the correct
337        * location in _strings_.
338        */
339       ((struct break_gizmo *)*next_gizmo)->no_break
340         = ((struct break_gizmo *)*next_gizmo)->strings;
341       ((struct break_gizmo *)*next_gizmo)->at_break
342         = ((struct break_gizmo *)*next_gizmo)->strings
343         + strlen(record.fields[1]) + 1;
344       /* Copy the strings into the gizmo. */
345       strcpy(((struct break_gizmo *)*next_gizmo)->no_break,
346           record.fields[1]);
347       strcpy(((struct break_gizmo *)*next_gizmo)->at_break,
348           record.fields[2]);
349       /* Update _next_gizmo_ to point to the new end of the linked list. */
350       next_gizmo = &(*next_gizmo)->next;
351       /* Next record. */
352       continue;
353     }
354     if (strcmp(record.fields[0], "BREAK") == 0) {
355       /* BREAK [SPACING] */
356       if (record.field_count != 2) {
357         fprintf(stderr, "Text BREAK command must take 1 option.\n");
358         continue;
359       }
```

```
360       if (str_to_int(record.fields[1], &arg1)) {
361         fprintf(stderr, "Text BREAK command's 1st option must be integer.\n");
362         continue;
363       }
364       /* Allocate memory for this gizmo. Plus 1 for the empty string. */
365       *next_gizmo = xmalloc(sizeof(struct break_gizmo) + 1);
366       /* Initialise this gizmo. */
367       (*next_gizmo)->type = GIZMO_BREAK;
368       (*next_gizmo)->next = NULL;
369       ((struct break_gizmo *)*next_gizmo)->spacing = arg1;
370       /* BREAK command forces a break here. */
371       ((struct break_gizmo *)*next_gizmo)->force_break = 1;
372       /* _selected_ is set to zero. Adjacent BREAKS are not both selected. */
373       ((struct break_gizmo *)*next_gizmo)->selected = 0;
374       ((struct break_gizmo *)*next_gizmo)->total_penalty = INT_MAX;
375       ((struct break_gizmo *)*next_gizmo)->best_source = NULL;
376       ((struct break_gizmo *)*next_gizmo)->style = current_style;
377       /* There is no 'at break' or 'no break' text. */
378       ((struct break_gizmo *)*next_gizmo)->no_break_width = 0;
379       ((struct break_gizmo *)*next_gizmo)->at_break_width = 0;
380       /* Both _no_break_ and _at_break_ point to an empty string. */
381       ((struct break_gizmo *)*next_gizmo)->no_break
382         = ((struct break_gizmo *)*next_gizmo)->at_break
383         = ((struct break_gizmo *)*next_gizmo)->strings;
384       ((struct break_gizmo *)*next_gizmo)->strings[0] = '\0';
385       /* Update _next_gizmo to point to the new end of the linked list. */
386       next_gizmo = &(*next_gizmo)->next;
387       /* Next record. */
388       continue;
389     }
390     if (strcmp(record.fields[0], "MARK") == 0) {
391       /* MARK [MARK_STRING] */
392       if (record.field_count != 2) {
393         fprintf(stderr, "Text MARK command must have 1 option.\n");
394         continue;
395       }
396       /* Allocate space for this gizmo and for its mark string. */
397       *next_gizmo = xmalloc(sizeof(struct mark_gizmo)
398           + strlen(record.fields[1]) + 1);
399       /* Initialise this gizmo. */
400       (*next_gizmo)->type = GIZMO_MARK;
401       (*next_gizmo)->next = NULL;
402       /* Copy the mark string into the gizmo. */
403       strcpy(((struct mark_gizmo *)*next_gizmo)->string, record.fields[1]);
404       /* Update _next_gizmo to point to the new end of the linked list. */
405       next_gizmo = &(*next_gizmo)->next;
406       /* Next record. */
407       continue;
408     }
409   }
410   /* A body of text must end in a forced break (the sink of the graph). */
411   *next_gizmo = xmalloc(sizeof(struct break_gizmo) + 1);
412   (*next_gizmo)->type = GIZMO_BREAK;
413   (*next_gizmo)->next = NULL;
414   ((struct break_gizmo *)*next_gizmo)->spacing = 0;
415   ((struct break_gizmo *)*next_gizmo)->force_break = 1;
416   ((struct break_gizmo *)*next_gizmo)->selected = 0;
417   ((struct break_gizmo *)*next_gizmo)->total_penalty = INT_MAX;
418   ((struct break_gizmo *)*next_gizmo)->best_source = NULL;
419   ((struct break_gizmo *)*next_gizmo)->style = current_style;
420   ((struct break_gizmo *)*next_gizmo)->no_break_width = 0;
421   ((struct break_gizmo *)*next_gizmo)->at_break_width = 0;
422   ((struct break_gizmo *)*next_gizmo)->no_break
423     = ((struct break_gizmo *)*next_gizmo)->at_break
424     = ((struct break_gizmo *)*next_gizmo)->strings;
425   ((struct break_gizmo *)*next_gizmo)->strings[0] = '\0';
426   free_record(&record);
427   /* Return the first node in the linked list. */
428   return first_gizmo;
429 }
430
431 /* Free the _gizmo_ linked list. */
432 static void
433 free_gizmos(struct gizmo *gizmo)
```

```
434 {
435   struct gizmo *next;
436   /* Loop over each node and free its heap allocated memory. */
437   while (gizmo) {
438     next = gizmo->next;
439     free(gizmo);
440     gizmo = next;
441   }
442 }
443
444 /* Relax the edges of the _source_ node. */
445 /* _gizmo_ is the next gizmo after the source node. */
446 static void
447 consider_breaks(struct gizmo *gizmo, int source_penalty,
448     struct break_gizmo *source, int line_width)
449 {
450   struct break_gizmo *break_gizmo;
451   int width, penalty;
452   width = 0;
453   /* Iterate over gizmos in topological order. */
454   for (; gizmo; gizmo = gizmo->next) {
455     switch (gizmo->type) {
456     case GIZMO_TEXT:
457       /* Increment width. */
458       width += ((struct text_gizmo *)gizmo)->width;
459       break;
460     case GIZMO_BREAK:
461       break_gizmo = (struct break_gizmo *)gizmo;
462       /* If feasible line. */
463       if (width + break_gizmo->at_break_width <= line_width) {
464         /* Compute edge wight (penalty). */
465         penalty = (line_width - width - break_gizmo->at_break_width) / 1000;
466         /* Forced breaks have no penalty. */
467         if (break_gizmo->force_break)
468           penalty = 0;
469         /* If this is a new best route to get to the break gizmo. */
470         if (break_gizmo->total_penalty > source_penalty + penalty) {
471           /* Update best route information. */
472           break_gizmo->best_source = source;
473           break_gizmo->total_penalty = source_penalty + penalty;
474         }
475       }
476       /* Increment width as if no break occurs. */
477       width += break_gizmo->no_break_width;
478       /* If all subsequent lines will not be feasible then stop. */
479       if (width > line_width)
480         goto stop;
481       if (width && break_gizmo->force_break)
482         goto stop;
483       break;
484     }
485   }
486 stop:
487 }
488
489 /* Find the optimal breaks in the linked list starting with _gizmo_. */
490 static void
491 optimise_breaks(struct gizmo *gizmo, int line_width)
492 {
493   struct break_gizmo *last_break;
494   last_break = NULL;
495   /* Relax the hypothetical leading break node. */
496   consider_breaks(gizmo, 0, NULL, line_width);
497   /* Relax every subsequent break node in topological order. */
498   for (; gizmo; gizmo = gizmo->next)
499     if (gizmo->type == GIZMO_BREAK) {
500       last_break = (struct break_gizmo *)gizmo;
501       consider_breaks(gizmo->next, last_break->total_penalty, last_break,
502           line_width);
503     }
504   /* Backtrack through the graph to set _selected_ to 1 on all chosen breaks. */
505   for (; last_break; last_break = last_break->best_source)
506     last_break->selected = 1;
507 }
```

```
508
509 /*
510  * Add new _string_ with _new_style_ to _buffer_'s _pages_ text mode content.
511  * _style_ is updated to match the font state of the text mode content.
512  */
513 static void
514 print_text(struct dbuffer *buffer, struct style *style,
515     const struct style *new_style, const char *string, int *spaces)
516 {
517   /* If string empty the do nothing. */
518   if (*string == '\0')
519     return;
520   /* If a different font is used. */
521   if (strcmp(style->font_name, new_style->font_name)
522       || style->font_size != new_style->font_size) {
523     /* Update style with the new style. */
524     memcpy(style, new_style, sizeof(struct style));
525     /* Write the new style to the buffer. */
526     dbuffer_printf(buffer, "FONT %s %d\n", style->font_name, style->font_size);
527   }
528   /* Start the string. */
529   dbuffer_printf(buffer, "STRING \"");
530   /* For each character in the string. */
531   while (*string) {
532     /* Skip newline characters. */
533     if (*string == '\n') {
534       string++;
535       continue;
536     }
537     /* Print a backslash before quotation marks or backslashes. */
538     if (*string == '"' || *string == '\\')
539       dbuffer_putc(buffer, '\\');
540     /* Count spaces (used for justified text). */
541     if (*string == ' ')
542       (*spaces)++;
543     /* Add the character to the string. */
544     dbuffer_putc(buffer, *string);
545     string++;
546   }
547   /* End the string. */
548   dbuffer_printf(buffer, "\"\n");
549 }
550
551 /* Using the _selected_ break gizmos, write _content_ to _output_. */
552 static void
553 print_gizmos(FILE *output, struct gizmo *gizmo, int line_width, int align)
554 {
555   int width, height, spaces;
556   struct style style;
557   struct dbuffer line;
558   struct dbuffer line_marks;
559   struct text_gizmo *text_gizmo;
560   struct break_gizmo *break_gizmo;
561   struct mark_gizmo *mark_gizmo;
562   width = 0;
563   height = 0;
564   spaces = 0;
565   style.font_name[0] = '\0';
566   style.font_size = 0;
567   /* _line_ is the text mode _pages_ content for this line. */
568   dbuffer_init(&line, 1024 * 4, 1024 * 4);
569   /* _line_marks_ are newline separated marks that appear on this line. */
570   dbuffer_init(&line_marks, 1024, 1024);
571   line_marks.data[0] = '\0';
572   /* For each gizmo in the linked list. */
573   for (; gizmo; gizmo = gizmo->next) {
574     switch (gizmo->type) {
575     case GIZMO_TEXT:
576       text_gizmo = (struct text_gizmo *)gizmo;
577       /* Update line height if this text is taller. */
578       if (text_gizmo->style.font_size > height)
579         height = text_gizmo->style.font_size;
580       /* Increment line width for this text. */
581       width += text_gizmo->width;
```

```
582       /* Add the text to the buffer. */
583       print_text(&line, &style, &text_gizmo->style, text_gizmo->string,
584           &spaces);
585       break;
586     case GIZMO_MARK:
587       mark_gizmo = (struct mark_gizmo *)gizmo;
588       /* Add this mark to the list of marks on this line. */
589       dbuffer_printf(&line_marks, "^%s\n", mark_gizmo->string);
590       break;
591     case GIZMO_BREAK:
592       break_gizmo = (struct break_gizmo *)gizmo;
593       /* Update line height if the break text is taller. */
594       if (break_gizmo->style.font_size > height)
595         height = break_gizmo->style.font_size;
596       /* If this break was selected. */
597       if (break_gizmo->selected) {
598         /* Line break occurs here so write and reset the line. */
599         width += break_gizmo->at_break_width;
600         print_text(&line, &style, &break_gizmo->style, break_gizmo->at_break,
601             &spaces);
602         /* If line not empty then write the line to _output_. */
603         if (line.size) {
604           dbuffer_putc(&line, '\0');
605           fprintf(output, "box %d\n", height);
606           /*
607            * Some align modes need the text to be horizontally shifted on the
608            * page. To do this, content enters graphic mode with START GRAPHIC
609            * and moves to the right with MOVE. Remember to end the graphic
610            * mode after exiting text mode.
611            */
612           if (align == ALIGN_CENTRE) {
613             fprintf(output, "START GRAPHIC\n");
614             fprintf(output, "MOVE %d 0\n", (line_width - width) / 2000);
615           } else if (align == ALIGN_RIGHT) {
616             fprintf(output, "START GRAPHIC\n");
617             fprintf(output, "MOVE %d 0\n", (line_width - width) / 1000);
618           }
619           /* Write the actual text. */
620           fprintf(output, "START TEXT\n");
621           if (align == ALIGN_JUSTIFIED && spaces && !break_gizmo->force_break) {
622             /* Justified text increases the size of space characters. */
623             fprintf(output, "SPACE %d\n", (line_width - width) / spaces);
624           }
625           fprintf(output, "%s", line.data);
626           /* End text. */
627           fprintf(output, "END\n");
628           /* End the additional graphic mode. */
629           if (align == ALIGN_CENTRE || align == ALIGN_RIGHT) {
630             fprintf(output, "END\n");
631           }
632         }
633         /* Write this lines marks. */
634         fprintf(output, line_marks.data);
635         /* If this break wants spacing, and if its not the last break. */
636         if (break_gizmo->spacing
637             && (break_gizmo->next == NULL || break_gizmo->next->next))
638           fprintf(output, "glue %d\n", break_gizmo->spacing);
639         /* Write the _content_ optional page break. */
640         fprintf(output, "opt_break\n");
641         /* Reset the line. */
642         line_marks.size = 0;
643         line_marks.data[0] = '\0';
644         height = 0;
645         width = 0;
646         spaces = 0;
647         style.font_name[0] = '\0';
648         style.font_size = 0;
649         line.size = 0;
650         line.data[0] = '\0';
651       } else {
652         /* This break gizmo is not selected. So add the no break string. */
653         width += break_gizmo->no_break_width;
654         print_text(&line, &style, &break_gizmo->style, break_gizmo->no_break,
655             &spaces);
```

```
656       }
657       break;
658     }
659   }
660   dbuffer_free(&line);
661   dbuffer_free(&line_marks);
662 }
663
664 static void
665 die_usage(char *program_name)
666 {
667   /* Print program usage and exit. */
668   fprintf(stderr, "Usage: %s -w NUM [-l] [-r] [-j] [-c]\n", program_name);
669   exit(1);
670 }
671
672 int
673 main(int argc, char **argv)
674 {
675   int opt, line_width, align;
676   FILE *input_file, *typeface_file, *output_file;
677   struct typeface *typeface;
678   struct gizmo *gizmos;
679   /* Default command line options. */
680   line_width = 0;
681   align = ALIGN_LEFT;
682   /* Parse command line arguments. */
683   while ( (opt = getopt(argc, argv, "ljrcw:")) != -1) {
684     switch (opt) {
685     case 'l':
686       align = ALIGN_LEFT;
687       break;
688     case 'j':
689       align = ALIGN_JUSTIFIED;
690       break;
691     case 'r':
692       align = ALIGN_RIGHT;
693       break;
694     case 'c':
695       align = ALIGN_CENTRE;
696       break;
697     case 'w':
698       /* Convert text width to integer. */
699       if (str_to_int(optarg, &line_width))
700         die_usage(argv[0]);
701       /* Convert point space to text space. */
702       line_width *= 1000;
703       break;
704     default:
705       /* Unrecognised command line option. */
706       die_usage(argv[0]);
707     }
708   }
709   /* Line width must be set by a command line option. */
710   if (line_width == 0)
711     die_usage(argv[0]);
712   input_file = stdin;
713   /*
714    * Typeface file is hardcoded to be called 'typeface'. This is a design
715    * decision: if we could call the typeface file anything we wanted, then
716    * other programs part of the typesetting process would struggle to locate
717    * it. Just like 'Makefile', 'typeface' is a hard-coded file name.
718    */
719   typeface_file = fopen("typeface", "r");
720   if (typeface_file == NULL) {
721     fprintf(stderr, "Failed to open typeface file.\n");
722     return 1;
723   }
724   output_file = stdout;
725   /* Parse the typeface file. */
726   typeface = open_typeface(typeface_file);
727   fclose(typeface_file);
728   /* Parse gizmos from standard input. */
729   gizmos = parse_gizmos(input_file, typeface);
```

```
730   /* Select the optimal line breaks. */
731   optimise_breaks(gizmos, line_width);
732   /* Print the _content_ to standard output. */
733   print_gizmos(output_file, gizmos, line_width, align);
734   /* Free memory and close files. */
735   free_typeface(typeface);
736   free_gizmos(gizmos);
737   fclose(input_file);
738   fclose(output_file);
739   return 0;
740 }
```

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 /* Initialise the dbuffer pointed to by _buf_. */
013 void
014 dbuffer_init(struct dbuffer *buf, int initial, int increment)
015 {
016   buf->size = 0;
017   buf->allocated = initial;
018   buf->increment = increment;
019   /* Allocate the initial heap memory. */
020   buf->data = xmalloc(buf->allocated);
021 }
022
023 /* Write a character to the end of a dynamic buffer. */
024 void
025 dbuffer_putc(struct dbuffer *buf, char c)
026 {
027   /* If there is insufficient space in the buffer then allocate more. */
028   if (buf->allocated == buf->size) {
029     buf->allocated += buf->increment;
030     buf->data = xrealloc(buf->data, buf->allocated);
031   }
032   /* Add the character and increment _size_. */
033   buf->data[buf->size++] = c;
034 }
035
036 /* Formatted print to the end of the buffer. */
037 void
038 dbuffer_printf(struct dbuffer *buf, const char *format, ...)
039 {
040   va_list args;
041   int len;
042   va_start(args, format);
043   /*
044    * Write to data but don't overflow the buffer. _len_ is the number of bytes
045    * that would have been written if the buffer were large enough.
046    */
047   len = vsnprintf(buf->data + buf->size, buf->allocated - buf->size, format,
048       args);
049   /* If there was not enough space allocated: */
050   if (len >= buf->allocated - buf->size) {
051     /* Allocate enough space for _len_ bytes. */
052     buf->allocated += buf->increment * (1 + len / buf->increment);
053     buf->data = xrealloc(buf->data, buf->allocated);
054     /*
055      * Write to data without checking overflow because we know there is enough
056      * space allocated.
057      */
058     vsprintf(buf->data + buf->size, format, args);
059   }
```

```
060   buf->size += len;
061   va_end(args);
062 }
063
064 /* Free dynamic buffer heap allocated memory. */
065 void
066 dbuffer_free(struct dbuffer *buf)
067 {
068   free(buf->data);
069 }
```

record.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 /*
007  * record.c
008  * Implements functions for _struct record_ including parsing records from a
009  * file.
010  */
011
012 #include <stdio.h>
013 #include <stdlib.h>
014 #include <string.h>
015
016 #include "tw.h"
017
018 /* Initialise a record's heap memory. */
019 void
020 init_record(struct record *record)
021 {
022   /* Allocate two KiB for the string. */
023   dbuffer_init(&record->string, 1024 * 2, 1024 * 2);
024   record->field_count = 0;
025   /* Allocate 32 elements for field pointers. */
026   record->fields_allocated = 32;
027   record->fields = xmalloc(record->fields_allocated * sizeof(char *));
028 }
029
030 /* Indicate that a new field will point to the current end of _string_. */
031 void
032 begin_field(struct record *record)
033 {
034   /* Allocate more fields if required */
035   if (record->field_count == record->fields_allocated) {
036     record->fields_allocated += 32;
037     record->fields = xrealloc(record->fields, record->fields_allocated * sizeof(char *));
038   }
039   /* Add a new field that points to the current end of _string_. */
040   record->fields[record->field_count++] = record->string.data
041     + record->string.size;
042 }
043
044 /* All parsing state machine states. */
045 enum ParseState {
046   /* PARSING STATES */
047   PARSE_BEGIN,
048   PARSE_NORMAL_FIELD,
049   PARSE_NORMAL_FIELD_ESCAPE,
050   PARSE_QUOTED_FIELD,
051   PARSE_QUOTED_FIELD_ESCAPE,
052   PARSE_OUTSIDE_FIELD,
053   /* TERMINATING STATES */
054   PARSE_FINISH_RECORD,
055   PARSE_EOF,
056   /* ERROR STATES */
057   PARSE_UNTERMINATED_STRING,
058   PARSE_UNTERMINATED_ESCAPE,
059 };
060
```

```
061 /* Parse a record from _file_. */
062 int
063 parse_record(FILE *file, struct record *record)
064 {
065   int c, state;
066   /* Set initial state. */
067   state = PARSE_BEGIN;
068   /* Reset record. */
069   record->string.size = 0;
070   record->field_count = 0;
071   /* While state machine not in terminating or error state. */
072   while (state < PARSE_FINISH_RECORD) {
073     /* Read the next character from _file_. */
074     c = fgetc(file);
075     /* Implement state transitions. */
076     switch (state) {
077     case PARSE_BEGIN:
078       if (c == EOF) {
079         state = PARSE_EOF;
080       } else if (c == ' ' || c == '\n') {
081         continue;
082       } else if (c == '"') {
083         state = PARSE_QUOTED_FIELD;
084         begin_field(record);
085       } else if (c == '\\') {
086         state = PARSE_NORMAL_FIELD_ESCAPE;
087         begin_field(record);
088       } else {
089         state = PARSE_NORMAL_FIELD;
090         begin_field(record);
091         dbuffer_putc(&record->string, (char)c);
092       }
093       break;
094     case PARSE_NORMAL_FIELD:
095       if (c == EOF) {
096         state = PARSE_EOF;
097         dbuffer_putc(&record->string, '\0');
098       } else if (c == '\\') {
099         state = PARSE_NORMAL_FIELD_ESCAPE;
100       } else if (c == ' ') {
101         state = PARSE_OUTSIDE_FIELD;
102         dbuffer_putc(&record->string, '\0');
103       } else if (c == '\n') {
104         state = PARSE_FINISH_RECORD;
105         dbuffer_putc(&record->string, '\0');
106       } else {
107         dbuffer_putc(&record->string, (char)c);
108       }
109       break;
110     case PARSE_NORMAL_FIELD_ESCAPE:
111       if (c == EOF || c == '\n') {
112         state = PARSE_UNTERMINATED_ESCAPE;
113       } else {
114         state = PARSE_NORMAL_FIELD;
115         dbuffer_putc(&record->string, (char)c);
116       }
117       break;
118     case PARSE_QUOTED_FIELD:
119       if (c == EOF || c == '\n') {
120         state = PARSE_UNTERMINATED_STRING;
121       } else if (c == '\\') {
122         state = PARSE_QUOTED_FIELD_ESCAPE;
123       } else if (c == '"') {
124         state = PARSE_OUTSIDE_FIELD;
125         dbuffer_putc(&record->string, '\0');
126       } else {
127         dbuffer_putc(&record->string, (char)c);
128       }
129       break;
130     case PARSE_QUOTED_FIELD_ESCAPE:
131       if (c == EOF || c == '\n') {
132         state = PARSE_UNTERMINATED_ESCAPE;
133       } else {
134         state = PARSE_QUOTED_FIELD;
```

```
135         dbuffer_putc(&record->string, (char)c);
136       }
137       break;
138     case PARSE_OUTSIDE_FIELD:
139       if (c == EOF) {
140         state = PARSE_EOF;
141       } else if (c == '\n') {
142         state = PARSE_FINISH_RECORD;
143       } else if (c == ' ') {
144         continue;
145       } else if (c == '"') {
146         state = PARSE_QUOTED_FIELD;
147         begin_field(record);
148       } else if (c == '\\') {
149         state = PARSE_NORMAL_FIELD_ESCAPE;
150         begin_field(record);
151       } else {
152         state = PARSE_NORMAL_FIELD;
153         begin_field(record);
154         dbuffer_putc(&record->string, (char)c);
155       }
156       break;
157     default:
158       /* The value of _state_ was not handled. */
159       fprintf(stderr, "Unexpected state while parsing record.\n");
160       break;
161     }
162   }
163   /* _state_ is now in its final position. */
164   switch (state) {
165   case PARSE_UNTERMINATED_STRING:
166     fprintf(stderr, "Failed to parse record: unterminated string.\n");
167     goto error;
168   case PARSE_UNTERMINATED_ESCAPE:
169     fprintf(stderr, "Failed to parse record: unterminated escape sequence.\n");
170     goto error;
171   case PARSE_EOF:
172     return EOF; /* Return EOF at end of file. */
173   case PARSE_FINISH_RECORD:
174     return 0; /* Return 0 on success. */
175   default:
176     fprintf(stderr, "Unexpected terminating state while parsing record.\n");
177     goto error;
178   }
179 error:
180   record->field_count = 0;
181   record->string.size = 0;
182   return 1; /* Return 1 on error. */
183 }
184
185 /* Find the index of field that matches _field_str_ otherwise return -1. */
186 int
187 find_field(const struct record *record, const char *field_str)
188 {
189   int i;
190   for (i = 0; i < record->field_count; i++)
191     if (strcmp(record->fields[i], field_str) == 0)
192       return i;
193   return -1;
194 }
195
196 /* Free _record_ heap allocated memory. */
197 void
198 free_record(struct record *record)
199 {
200   dbuffer_free(&record->string);
201   free(record->fields);
202 }
```

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 /* Write the 1.7 header to a new PDF file. */
024 void
025 pdf_write_header(FILE *file)
026 {
027   fprintf(file, "%%PDF-1.7\n");
028 }
029
030 /*
031  * Start an 'indirect object' (see the PDf spec). _obj_ is the unique reference
032  * to use for this object. Add the indirecto object to the xref table.
033  */
034 void
035 pdf_start_indirect_obj(FILE *file, struct pdf_xref_table *xref, int obj)
036 {
037   xref->obj_offsets[obj] = ftell(file);
038   fprintf(file, "%d 0 obj\n", obj);
039 }
040
041 /* End a PDF 'indirect object' (see the PDF spec). */
042 void
043 pdf_end_indirect_obj(FILE *file)
044 {
045   fprintf(file, "endobj\n");
046 }
047
048 /* Write a PDF 'stream' with contents of _data_file_ encoded as hex. */
049 void
050 pdf_write_file_stream(FILE *pdf_file, FILE *data_file)
051 {
052   long size, i;
053   /* Get the size of the file. */
054   fseek(data_file, 0, SEEK_END);
055   size = ftell(data_file);
056   /* Return to the start of the file. */
057   fseek(data_file, 0, SEEK_SET);
058   /* Write the PDF dictionary and start the stream. */
059   fprintf(pdf_file, "<<\n\
060   /Filter /ASCIIHexDecode\n\
061   /Length %ld\n\
062   /Length1 %ld\n\
063   >>\nstream\n", size * 2, size);
064   /* Write all the stream bytes encoded as hex. */
065   for (i = 0; i < size; i++)
066     fprintf(pdf_file, "%02x", (unsigned char)fgetc(data_file));
067   /* End the stream. */
068   fprintf(pdf_file, "\nendstream\n");
069 }
070
```

```
071 /*
072  * Write a PDF text 'stream' with the data pointed to by _data_ of length
073  * _size_.
074  */
075 void
076 pdf_write_text_stream(FILE *file, const char *data, long size)
077 {
078   fprintf(file, "<< /Length %ld >> stream\n", size);
079   fwrite(data, 1, size, file);
080   fprintf(file, "\nendstream\n");
081 }
082
083 /* Write a PDF integer array to _file_ (see the PDF spec). */
084 void
085 pdf_write_int_array(FILE *file, const int *values, int count)
086 {
087   int i;
088   /* Comma separated values, trailing comma is allowed */
089   fprintf(file, "[\n ");
090   for (i = 0; i < count; i++)
091     fprintf(file, " %d", values[i]);
092   fprintf(file, "\n]\n");
093 }
094
095 /* Write a PDF font descriptor dictionary to _file_ (see the PDF spec). */
096 void
097 pdf_write_font_descriptor(FILE *file, int font_file, const char *font_name,
098     int italic_angle, int ascent, int descent, int cap_height,
099     int stem_vertical, int min_x, int min_y, int max_x, int max_y)
100 {
101   fprintf(file, "<<\n\
102   /Type /FontDescriptor\n\
103   /FontName /%s\n\
104   /FontFile2 %d 0 R\n\
105   /Flags 6\n\
106   /FontBBox [%d, %d, %d, %d]\n\
107   /ItalicAngle %d\n\
108   /Ascent %d\n\
109   /Descent %d\n\
110   /CapHeight %d\n\
111   /StemV %d\n\
112 >>\n", font_name, font_file, min_x, min_y, max_x, max_y, italic_angle,
113       ascent, descent, cap_height, stem_vertical);
114 }
115
116 /* Write a PDF page descriptor dictionary to _file_ (see the PDF spec). */
117 void
118 pdf_write_page(FILE *file, int parent, int content)
119 {
120   fprintf(file, "<<\n\
121   /Type /Page\n\
122   /Parent %d 0 R\n\
123   /Contents %d 0 R\n\
124 >>\n", parent, content);
125 }
126
127 /* Write a PDF font dictionary to _file_ (see the PDF spec). */
128 void
129 pdf_write_font(FILE *file, const char *font_name, int font_descriptor,
130     int font_widths)
131 {
132   fprintf(file, "<<\n\
133   /Type /Font\n\
134   /Subtype /TrueType\n\
135   /BaseFont /%s\n\
136   /FirstChar 0\n\
137   /LastChar 255\n\
138   /Widths %d 0 R\n\
139   /FontDescriptor %d 0 R\n\
140 >>\n", font_name, font_widths, font_descriptor);
141 }
142
143 /* Write PDF pages dictionary to _file_ (see the PDF spec). */
144 void
```

```
145 pdf_write_pages(FILE *file, int resources, int page_count, const int *page_objs)
146 {
147   int i;
148   fprintf(file, "<<\n\
149   /Type /Pages\n\
150   /Resources %d 0 R\n\
151   /Kids [\n", resources);
152   /* Add a reference to all page objects. */
153   for (i = 0; i < page_count; i++)
154     fprintf(file, "    %d 0 R\n", page_objs[i]);
155   /* 595x842 is a portrait A4 page. */
156   fprintf(file, "    ]\n\
157   /Count %d\n\
158   /MediaBox [0 0 595 842]\n\
159 >>\n", page_count);
160 }
161
162 /* Write a PDF catalog dictionary to _file_ (see the PDF spec). */
163 void
164 pdf_write_catalog(FILE *file, int page_list)
165 {
166   fprintf(file, "<<\n\
167   /Type /Catalog\n\
168   /Pages %d 0 R\n\
169 >>\n", page_list);
170 }
171
172 /* Initialise a PDF xref table. */
173 void
174 init_pdf_xref_table(struct pdf_xref_table *xref)
175 {
176   /* The 'zero' object is the first (see the PDF spec). */
177   xref->obj_count = 1;
178   xref->allocated = 100;
179   /* Allocate memory for objects. */
180   xref->obj_offsets = xmalloc(xref->allocated * sizeof(long));
181   /*
182    * Set unused elements to zero so if we try to use them as references before
183    * they are set then we will get an error.
184    */
185   memset(xref->obj_offsets, 0, xref->allocated * sizeof(long));
186 }
187
188 /* Allocate a PDF object reference number from the xref table. */
189 int
190 allocate_pdf_obj(struct pdf_xref_table *xref)
191 {
192   /*
193    * If more memory needs to allocated to fit the new object, then allocate 100
194    * more.
195    */
196   if (xref->obj_count == xref->allocated) {
197     xref->obj_offsets
198       = xrealloc(xref->obj_offsets, (xref->allocated + 100) * sizeof(long));
199     /* Set the newly allocated elements to zero. */
200     memset(xref->obj_offsets + xref->allocated, 0, 100 * sizeof(long));
201     xref->allocated += 100;
202   }
203   /* Return the old _obj_count_ and increment _object_count_. */
204   return xref->obj_count++;
205 }
206
207 /* Add the PDF footer to _file_ (see the PDF spec). */
208 void
209 pdf_add_footer(FILE *file, const struct pdf_xref_table *xref, int root_obj)
210 {
211   int xref_offset, i;
212   /* Get our offset in _file_. */
213   xref_offset = ftell(file);
214   /* Write the footer. */
215   fprintf(file, "xref\n\
216 0 %d\n\
217 000000000 65535 f \n", xref->obj_count);
218   for (i = 1; i < xref->obj_count; i++)
```

```
219     fprintf(file, "%09ld 00000 n \n", xref->obj_offsets[i]);
220   fprintf(file, "trailer << /Size %d /Root %d 0 R >>\n\
221 startxref\n\
222 %d\n\
223 %%%%EOF", xref->obj_count, root_obj, xref_offset);
224 }
225
226 /* Free heap allocated memory of the _xref_ table. */
227 void
228 free_pdf_xref_table(struct pdf_xref_table *xref)
229 {
230   free(xref->obj_offsets);
231 }
232
233 /* Initialise _resources_. */
234 void
235 init_pdf_resources(struct pdf_resources *resources)
236 {
237   /* Initialise the records that will be used to store a list of strings. */
238   init_record(&resources->fonts_used);
239   init_record(&resources->images);
240 }
241
242 /* Remember that _font_ was used in the PDF, so it can be embedded later. */
243 void
244 include_font_resource(struct pdf_resources *resources, const char *font)
245 {
246   int i;
247   /*
248    * Loop through all fonts currently in the _fonts_used_ list. If _font_ is
249    * found to already be in this list then reutrn.
250    */
251   for (i = 0; i < resources->fonts_used.field_count; i++)
252     if (strcmp(resources->fonts_used.fields[i], font) == 0)
253       return;
254   /* Add a new field to the _fonts_used_ array. */
255   begin_field(&resources->fonts_used);
256   dbuffer_printf(&resources->fonts_used.string, "%s", font);
257   dbuffer_putc(&resources->fonts_used.string, '\0');
258 }
259
260 /*
261  * Remember that _fname_ image was used in the PD Fso it can be embedded later.
262  */
263 int
264 include_image_resource(struct pdf_resources *resources, const char *fname)
265 {
266   int i;
267   /*
268    * Loop through all images already in resources. If we find the image then
269    * return.
270    */
271   for (i = 0; i < resources->images.field_count; i++)
272     if (strcmp(resources->images.fields[i], fname) == 0)
273       return i;
274   /* If we did not return, the new image is added to resources. */
275   begin_field(&resources->images);
276   dbuffer_printf(&resources->images.string, "%s", fname);
277   dbuffer_putc(&resources->images.string, '\0');
278   /* Return the index of the image resource. */
279   return i;
280 }
281
282 /*
283  * Add a font resource to the PDF file and return the indirect object reference
284  * to it.
285  */
286 static int
287 pdf_add_font(FILE *pdf_file, FILE *font_file, struct pdf_xref_table *xref,
288     const char *name)
289 {
290   struct font_info font_info;
291   int font_program, font_widths, font_descriptor, font;
292
```

```
293   /* Go to the start of the font file, and parse _font_info_ from it. */
294   fseek(font_file, 0, SEEK_SET);
295   if (read_ttf(font_file, &font_info))
296     return -1; /* Return -1 on error */
297   /* Return to start of font file. */
298   fseek(font_file, 0, SEEK_SET);
299
300   /* Allocate pdf indirect object references we are gona use. */
301   font_program = allocate_pdf_obj(xref);
302   font_widths = allocate_pdf_obj(xref);
303   font_descriptor = allocate_pdf_obj(xref);
304   font = allocate_pdf_obj(xref);
305
306   /* Write the font file bytes into a stream in an indirect object. */
307   pdf_start_indirect_obj(pdf_file, xref, font_program);
308   pdf_write_file_stream(pdf_file, font_file);
309   pdf_end_indirect_obj(pdf_file);
310
311   /* Write the character width array into a indirect object. */
312   pdf_start_indirect_obj(pdf_file, xref, font_widths);
313   pdf_write_int_array(pdf_file, font_info.char_widths, 256);
314   pdf_end_indirect_obj(pdf_file);
315
316   /* Write the font descriptor into an indirect object. */
317   pdf_start_indirect_obj(pdf_file, xref, font_descriptor);
318   pdf_write_font_descriptor(pdf_file, font_program, name, -10, 255,
319       255, 255, 10, font_info.x_min, font_info.y_min, font_info.x_max,
320       font_info.y_max);
321   pdf_end_indirect_obj(pdf_file);
322
323   /* Write the font dictionary into an indirect object. */
324   pdf_start_indirect_obj(pdf_file, xref, font);
325   pdf_write_font(pdf_file, name, font_descriptor, font_widths);
326   pdf_end_indirect_obj(pdf_file);
327
328   /* Return the indirect object reference to the font. */
329   return font;
330 }
331
332 /* Add an image resource to a PDF file. */
333 static int
334 pdf_add_image(FILE *pdf_file, FILE *image_file, struct pdf_xref_table *xref)
335 {
336   int image;
337   long image_length, i;
338   struct jpeg_info jpeg_info;
339   const char *color_space;
340   /* Parse the jpeg file to get _jpeg_info_. */
341   if (read_jpeg(image_file, &jpeg_info))
342     return -1; /* Return -1 on error */
343   /*
344    * Set color space to RGB or Grey depending on number of color components in
345    * the jpeg file.
346    */
347   color_space = jpeg_info.components == 3 ? "DeviceRGB" : "DeviceGray";
348   /* Get the length of the _image_file_. */
349   fseek(image_file, 0, SEEK_END);
350   image_length = ftell(image_file);
351   /* Return to the begining of the file. */
352   fseek(image_file, 0, SEEK_SET);
353   /* Allocate a pdf indirect object reference for the image dictionary. */
354   image = allocate_pdf_obj(xref);
355   /* Write the PDF image dictionary into a indirect object. */
356   pdf_start_indirect_obj(pdf_file, xref, image);
357   fprintf(pdf_file, "<<\n\
358   /Type /XObject\n\
359   /Subtype /Image\n\
360   /Width %d\n\
361   /Height %d\n\
362   /ColorSpace /%s\n\
363   /BitsPerComponent 8\n\
364   /Length %ld\n\
365   /Filter /DCTDecode\n\
366   >>\n", jpeg_info.width, jpeg_info.height, color_space, image_length);
```

```
367   /* Write the embedded image stream at the end of the image object. */
368   fprintf(pdf_file, "stream\n");
369   for (i = 0; i < image_length; i++)
370     fputc(fgetc(image_file), pdf_file);
371   fprintf(pdf_file, "\nendstream\n");
372   pdf_end_indirect_obj(pdf_file);
373   /* Return the indirect object reference to the image. */
374   return image;
375 }
376
377 /* Add all resources to the PDF file. */
378 void
379 pdf_add_resources(FILE *pdf_file, FILE *typeface_file, int resources_obj,
380     const struct pdf_resources *resources, struct pdf_xref_table *xref)
381 {
382   FILE *font_file, *image_file;
383   struct record typeface_record;
384   int i, parse_result;
385   int *font_objs, *image_objs;
386   init_record(&typeface_record);
387   /* Initialise arrays for font and image indirect objects. */
388   font_objs = xmalloc(resources->fonts_used.field_count * sizeof(int));
389   image_objs = xmalloc(resources->images.field_count * sizeof(int));
390   /*
391    * Initaliaise the font indirect objects to -1 so if its not found in the
392    * typeface file, we will know.
393    */
394   for (i = 0; i < resources->fonts_used.field_count; i++)
395     font_objs[i] = -1;
396   /* Loop through every font in the TYPEFACE. */
397   while ( (parse_result = parse_record(typeface_file, &typeface_record))
398       != EOF ) {
399     if (parse_result) /* If failed to parse record then skip it. */
400       continue;
401     if (typeface_record.field_count != 2) {
402       fprintf(stderr, "Typeface records must have exactly 2 fields.");
403       continue;
404     }
405     if (strlen(typeface_record.fields[0]) >= 256) {
406       fprintf(stderr,
407           "Typeface file contains font name that is too long '%s'.\n",
408           typeface_record.fields[0]);
409       continue;
410     }
411     /* Check typeface does not contain any illegal characters. */
412     if (!is_font_name_valid(typeface_record.fields[0])) {
413       fprintf(stderr, "Typeface file contains invalid font name '%s'.\n",
414           typeface_record.fields[0]);
415       continue;
416     }
417     /*
418      * If this font name is in _resources->fonts_used_ then it is used
419      * somewhere in the PDF so write the font resource to the PDF.
420      */
421     i = find_field(&resources->fonts_used, typeface_record.fields[0]);
422     if (i == -1) /* Font not used in the PDF. */
423       continue;
424     /* Try to open this font file. */
425     font_file = fopen(typeface_record.fields[1], "r");
426     if (font_file == NULL) {
427       fprintf(stderr, "Failed to open ttf file '%s': %s\n",
428           typeface_record.fields[1], strerror(errno));
429       continue;
430     }
431     /* Add the font resource. */
432     font_objs[i] = pdf_add_font(pdf_file, font_file, xref,
433       typeface_record.fields[0]);
434     if (font_objs[i] == -1) {
435       fprintf(stderr, "Failed to parse ttf file '%s'\n",
436           typeface_record.fields[1]);
437     }
438     fclose(font_file);
439   }
440   free_record(&typeface_record);
```

```
441
442   /* Add all image resources. */
443   for (i = 0; i < resources->images.field_count; i++) {
444     image_file = fopen(resources->images.fields[i], "r");
445     if (image_file == NULL) {
446       fprintf(stderr, "Failed to open image file '%s'\n",
447           resources->images.fields[i]);
448       continue;
449     }
450     image_objs[i] = pdf_add_image(pdf_file, image_file, xref);
451     fclose(image_file);
452   }
453
454   /* Add the PDF resources dictionary into an indirect object. */
455   pdf_start_indirect_obj(pdf_file, xref, resources_obj);
456   fprintf(pdf_file, "<<\n  /Font <<\n");
457   for (i = 0; i < resources->fonts_used.field_count; i++) {
458     /*
459      * If _font_objs[i] is still -1 then it was not found in the typeface file.
460      */
461     if (font_objs[i] == -1) {
462       fprintf(stderr, "Typeface file does not include '%s' font.\n",
463           resources->fonts_used.fields[i]);
464     }
465     /* Add the font indirect reference to the fonts dictionary. */
466     fprintf(pdf_file, "    /%s %d 0 R\n", resources->fonts_used.fields[i],
467         font_objs[i]);
468   }
469   fprintf(pdf_file, "  >>\n  /XObject <<\n");
470   /* Add image indirect references to resources dictionary. */
471   for (i = 0; i < resources->images.field_count; i++)
472     fprintf(pdf_file, "    /Img%d %d 0 R\n", i, image_objs[i]);
473   fprintf(pdf_file, "  >>\n>>\n");
474   pdf_end_indirect_obj(pdf_file);
475   /* Free local arrays. */
476   free(font_objs);
477   free(image_objs);
478 }
479
480 /* Free lists allocated on the heap by _resources_. */
481 void
482 free_pdf_resources(struct pdf_resources *resources)
483 {
484   free_record(&resources->fonts_used);
485   free_record(&resources->images);
486 }
487
488 /* Initialise the list of indirect page object reference. */
489 void
490 init_pdf_page_list(struct pdf_page_list *page_list)
491 {
492   page_list->page_count = 0;
493   page_list->pages_allocated = 100;
494   page_list->page_objs = xmalloc(page_list->pages_allocated * sizeof(int));
495 }
496
497 /* Add this page reference to the list. */
498 void
499 add_pdf_page(struct pdf_page_list *page_list, int page)
500 {
501   /* If more memory needs to be allocated for the list. */
502   if (page_list->page_count == page_list->pages_allocated) {
503     /* Allocate 100 more elements. */
504     page_list->pages_allocated += 100;
505     page_list->page_objs = xrealloc(page_list->page_objs,
506         page_list->pages_allocated * sizeof(int));
507   }
508   /* Set the new page element and increment the page count. */
509   page_list->page_objs[page_list->page_count++] = page;
510 }
511
512 /* Free the page list heap allocated memory. */
513 void
```

```
514 free_pdf_page_list(struct pdf_page_list *page_list)
515 {
516   free(page_list->page_objs);
517 }
```

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   /* Copy each byte individually and reverse order. */
025   fread(1 + (char *)&n, 1, 1, file);
026   fread((char *)&n, 1, 1, file);
027   return n;
028 }
029
030 /*
031  * Skips a jpeg segment by reading segment length and seeking forward that
032  * amount.
033  * */
034 static void
035 skip_segment_payload(FILE *file)
036 {
037   uint16_t segment_length;
038   segment_length = read_uint16(file);
039   /* Subtract two because length does not include first two bytes. */
040   fseek(file, segment_length - 2, SEEK_CUR);
041 }
042
043 /* Parse a jpeg file and write the file info to _info_ */
044 int
045 read_jpeg(FILE *file, struct jpeg_info *info)
046 {
047   unsigned char magic_bytes[2];
048   unsigned char segment_code[2];
049
050   /* Read the first two bytes from the file. */
051   fread(magic_bytes, 1, 2, file);
052   /* jpegs always start with ffd8. The 'magic bytes'. */
053   if (magic_bytes[0] != 0xff || magic_bytes[1] != 0xd8) {
054     fprintf(stderr, "File not JPEG.\n");
055     return 1;
056   }
057
058   /* For each segment in jpeg file. */
059   for(;;) {
060     /* Read the two byte segment code. */
061     fread(segment_code, 1, 2, file);
062     /* Segment codes always start with the byte 0xff. */
063     if (segment_code[0] != 0xFF) {
064       fprintf(stderr, "JPEG file invalid.\n");
065       return 1;
066     }
```

```
067     /* These segments are reserved for application specific segments. */
068     if (segment_code[1] >= 0xe0 && segment_code[1] <= 0xef) {
069       /* Skip application specific segment. */
070       skip_segment_payload(file);
071     } else {
072       /* Handle this segment accordingly. */
073       switch (segment_code[1]) {
074       case 0xfe: /* comment marker */
075       case 0xdb: /* quantization tables */
076       case 0xc4: /* huffman tables */
077         skip_segment_payload(file);
078         break;
079       case 0xc0: /* baseline DCT segment (contains image width and height) */
080         fseek(file, 3, SEEK_CUR);
081         info->height = read_uint16(file);
082         info->width = read_uint16(file);
083         fread(&info->components, 1, 1, file);
084         /* When we find this segment we stop because this is all we want. */
085         goto end_scan;
086       case 0xc2: /* progressive DCT segment (unsupported) */
087         fprintf(stderr, "Progressive DCT JPEGs are unsupported.\n");
088         return 1;
089       case 0xda: /* compressed image data */
090         /*
091          * If we get to the image data and have not found width and height yet
092          * then assume we will not find it.
093          */
094         fprintf(stderr, "JPEG image data reached and no DCT segment found.\n");
095         return 1;
096       case 0xd9: /* end of image marker */
097         fprintf(stderr, "End of JPEG reached and no DCT segment found.\n");
098         return 1;
099       default:
100         /* If we did not handle this segment code. */
101         fprintf(stderr, "Unrecognised JPEG segment code '%02x %02x'\n",
102             segment_code[0], segment_code[1]);
103         return 1;
104       }
105     }
106     /* If we are at the end of the file or if there was an error reading. */
107     if (ferror(file) || feof(file)) {
108       fprintf(stderr, "Error reading JPEG file.\n");
109       return 1;
110     }
111   }
112 end_scan:
113   /* If we are at the end of the file or if there was an error reading. */
114   if (ferror(file) || feof(file)) {
115     fprintf(stderr, "Error reading JPEG file.\n");
116     return 1;
117   }
118   /* PDF only supports JPEGS with rgb (3) or greyscale (1) colorspace. */
119   if (info->components != 1 && info->components != 3) {
120     fprintf(stderr, "JPEG file has unsupported color component count '%d'.\n",
121         info->components);
122     return 1;
123   }
124   return 0;
125 }
```

ttf.c

```
001 /*
002  * Copyright (C) 2023 Christopher Lang
003  * See LICENSE for license details.
004  */
005
006 /*
007  * ttf.c
008  * This file parses true type font files.
009  *
010  * See Apple's TrueType reference manual for file format info:
011  * https://developer.apple.com/fonts/TrueType-Reference-Manual/
```

```
012  */
013
014 #include <stdint.h>
015 #include <stdio.h>
016 #include <string.h>
017 #include <stdlib.h>
018
019 #include "tw.h"
020
021 /* Number of elements in the _required_tables_ constant array. */
022 #define REQUIRED_TABLE_COUNT \
023   (sizeof(required_tables) / sizeof(required_tables[0]))
024
025 static int16_t read_int16(const char *ptr);
026 static int32_t read_int32(const char *ptr);
027 static int read_head_table(
028     const char *table, long table_size, struct font_info *info);
029 static int read_format4_cmap_subtable(
030     const char *table, long table_size, struct font_info *info);
031 static int read_cmap_table(
032     const char *table, long table_size, struct font_info *info);
033 static int read_hhea_table(
034     const char *table, long table_size, struct font_info *info);
035 static int read_hmtx_table(
036     const char *table, long table_size, struct font_info *info);
037
038 /* ttf tables that we need to get the information we parse. */
039 static const char *required_tables[] = {"head", "cmap", "hhea", "hmtx"};
040 /* A list of functions for parsing the tables that we need. */
041 static int (*required_table_parsers[REQUIRED_TABLE_COUNT])
042   (const char *table, long table_size, struct font_info *font) = {
043   read_head_table,
044   read_cmap_table,
045   read_hhea_table,
046   read_hmtx_table,
047 };
048
049 /* Convert big-edian to little-endian. Assume this machine is little-endian. */
050 static int16_t
051 read_int16(const char *ptr)
052 {
053   uint16_t ret;
054   /* Copy the bytes in reverse order. */
055   ((char *)&ret)[0] = ptr[1];
056   ((char *)&ret)[1] = ptr[0];
057   return ret;
058 }
059
060 /* Convert big-edian to little-endian. Assume this machine is little-endian. */
061 static int32_t
062 read_int32(const char *ptr)
063 {
064   uint32_t ret;
065   /* Copy the bytes in reverse order. */
066   ((char *)&ret)[0] = ptr[3];
067   ((char *)&ret)[1] = ptr[2];
068   ((char *)&ret)[2] = ptr[1];
069   ((char *)&ret)[3] = ptr[0];
070   return ret;
071 }
072
073 /* Parse the ttf 'head' table. */
074 static int
075 read_head_table(const char *table, long table_size, struct font_info *info)
076 {
077   /* Head tables must have size of exactly 54 bytes. */
078   if (table_size != 54)
079     return 1;
080   /* Read the important values from this table. */
081   info->units_per_em = read_int16(table + 18);
082   info->x_min = read_int16(table + 36) * 1000 / info->units_per_em;
083   info->y_min = read_int16(table + 38) * 1000 / info->units_per_em;
084   info->x_max = read_int16(table + 40) * 1000 / info->units_per_em;
085   info->y_max = read_int16(table + 42) * 1000 / info->units_per_em;
```

```
086   return 0;
087 }
088
089 /*
090  * Parse the format 4 character map subtable. The purpose of this table it to
091  * map a character code to an index in the glyph array (see the ttf reference).
092  */
093 static int
094 read_format4_cmap_subtable(
095     const char *table,
096     long table_size,
097     struct font_info *info)
098 {
099   unsigned int range_index, char_index;
100   uint16_t seg_count;
101   const char *end_codes, *start_codes, *deltas, *offsets;
102   /* The subtable must be at least 16 bytes. */
103   if (table_size < 16)
104     return 1;
105   /*
106    * Read the segment count (number of continuous mapping ranges in the font).
107    */
108   seg_count = read_int16(table + 6) / 2;
109   /*
110    * The _start_codes_ and _end_codes_ array define the range of a continuous
111    * character mapping.
112    * _deltas_ and _offsets_ define the contents of the mapping range.
113    */
114   end_codes = table + 14;
115   start_codes = end_codes + seg_count * 2 + 2;
116   deltas = start_codes + seg_count * 2;
117   offsets = deltas + seg_count * 2;
118   /* If the table is not large enough to fir the arrays it claims to have. */
119   if (table_size < 16 + seg_count * 8)
120     return 1;
121   /* Set default mappings to zero. */
122   for (char_index = 0; char_index < 256; char_index++)
123     info->cmap[char_index] = 0;
124   /* For each continuous character range. */
125   for (range_index = 0; range_index < seg_count; range_index++) {
126     uint16_t start, end, glyph_delta, glyph_offset;
127     /* Get the start and end caracters. */
128     end = read_int16(end_codes + range_index * 2);
129     start = read_int16(start_codes + range_index * 2);
130     /* If range outside ASCII then ignore. */
131     if (end > 255) end = 255;
132     if (start > 255) break;
133     /* Read the delta and offset for this mapping. */
134     glyph_delta = read_int16(deltas + range_index * 2);
135     glyph_offset = read_int16(offsets + range_index * 2);
136     /*
137      * If glyph offset is not zero then we need to look in the glyph index
138      * array.
139      */
140     if (glyph_offset == 0) {
141       /* Simple character mapping range, 1-1. */
142       for (char_index = start; char_index <= end; char_index++)
143         info->cmap[char_index] = char_index + glyph_delta;
144     } else {
145       /* If the table is not large enough for the glyph index array to contain
146        * all it claims to the throw an error. */
147       if(offsets + range_index * 2 + glyph_offset
148           + 2 * (end - start) + 2
149           > table + table_size)
150         return 1;
151       /*
152        * For each character in the range. Map the character to the glyph index
153        * specified in the glyph index array.
154        */
155       for (char_index = start; char_index <= end; char_index++)
156         info->cmap[char_index] = read_int16(
157           offsets + range_index * 2
158           + glyph_offset
159           + 2 * (char_index - start));
```

```
160     }
161   }
162   /* Retturn zero on success. */
163   return 0;
164 }
165
166 /* Read ttf character map table. */
167 static int
168 read_cmap_table(const char *table, long table_size, struct font_info *info)
169 {
170   uint16_t subtable_count, format, subtable_size;
171   const char *subtable;
172   /* Table must be more than 4 bytes in size. */
173   if (table_size < 4)
174     goto invalid;
175   /* Get the subtable count. */
176   subtable_count = read_int16(table + 2);
177   /* If there is not enough space in this table for all the sub tables. */
178   if (table_size < 4 + subtable_count * 8)
179     goto invalid;
180   /* For each subtable. */
181   for (subtable = table + 4;
182       subtable < table + 4 + subtable_count * 8;
183       subtable += 8) {
184     uint16_t platform_id, platform_specific_id;
185     uint32_t offset;
186     /* Read subtable info. */
187     platform_id = read_int16(subtable);
188     platform_specific_id = read_int16(subtable + 2);
189     offset = read_int32(subtable + 4);
190     /* If we find a table that is supported. */
191     if (platform_id == 0 && platform_specific_id <= 4) {
192       /* Then go to this table. */
193       subtable = table + offset;
194       goto found_cmap;
195     }
196   }
197   /*
198    * If the for loop exits without finding a character map table, then
199    * there are only unsupported tables in the font file.
200    */
201   goto unsupported;
202 found_cmap:
203   /* If there is not enough space in this table for the subtable. */
204   if (subtable + 4 > table + table_size)
205     goto invalid;
206   /* Read the subtable format. */
207   format = read_int16(subtable);
208   subtable_size = read_int16(subtable + 2);
209   /* If there is not enough space in the table for the subtable. */
210   if (subtable + subtable_size > table + table_size)
211     goto invalid;
212   /* We only support format 4 character map subtables. */
213   if (format != 4)
214     goto unsupported;
215   /* Read the character mapping from the subtable. */
216   if (read_format4_cmap_subtable(subtable, subtable_size, info))
217     goto subtable_error;
218   return 0; /* Return zero on success. */
219 invalid:
220   fprintf(stderr, "ttf cmap table invalid\n");
221   return 1; /* Return 1 on failure. */
222 unsupported:
223   fprintf(stderr, "ttf cmap table not supported\n");
224   return 1;
225 subtable_error:
226   fprintf(stderr, "failed to read ttf cmap subtable\n");
227   return 1;
228 }
229
230 /* Read the 'hhea' font table. */
231 static int
232 read_hhea_table(const char *table, long table_size, struct font_info *info)
233 {
```

```
234   if (table_size != 36)
235     return 1;
236   /* Number of character width metrics in the htmx table. */
237   info->long_hor_metrics_count = read_int16(table + 34);
238   return 0;
239 }
240
241 /* Read the character width table. */
242 static int
243 read_hmtx_table(const char *table, long table_size, struct font_info *info)
244 {
245   int i;
246   uint16_t fallback_tt_width;
247   uint16_t glyph, tt_width;
248   /* There must be at least 1 metric. */
249   if (info->long_hor_metrics_count < 1)
250     return 1;
251   /* The table must be large enough for all metrics. */
252   if (table_size < info->long_hor_metrics_count * 4)
253     return 1;
254   /* If a glyph width metric does not exist use this a default. */
255   fallback_tt_width
256     = read_int16(table + 4 * (info->long_hor_metrics_count - 1));
257   /* For each character. */
258   for (i = 0; i < 256; i++) {
259     /* Get this characters glyph index from the character map table. */
260     glyph = info->cmap[i];
261     /*
262      * If this glyph is in the glyph width metric array then use that metric.
263      * Otherwise, use the fallback width for this character.
264      */
265     tt_width
266       = glyph >= info->long_hor_metrics_count
267       ? fallback_tt_width
268       : read_int16(table + 4 * glyph);
269     /* Change the width units based on the font scaling. */
270     info->char_widths[i] = tt_width * 1000 / info->units_per_em;
271   }
272   /* Return 0 on success. */
273   return 0;
274 }
275
276 /* Parse a font file. */
277 int
278 read_ttf(FILE *file, struct font_info *info)
279 {
280   /*
281    * This file was originally written to parse the ttf data from a buffer. I
282    * later decided to change the function definition to read straight from a
283    * file descriptor. I couldn't be bothered to rewrite all the ttf parsing
284    * code to read right from the file, so I added the following hack that reads
285    * the file into a sufficiently large buffer and then the rest of the code
286    * works as it did before.
287    */
288   char *ttf;
289   long ttf_size;
290   fseek(file, 0, SEEK_END);
291   ttf_size = ftell(file);
292   fseek(file, 0, SEEK_SET);
293   ttf = xmalloc(ttf_size);
294   fread(ttf, 1, ttf_size, file);
295
296   int i;
297   uint32_t magic_bytes;
298   uint16_t table_count;
299   uint32_t required_table_offsets[REQUIRED_TABLE_COUNT];
300   uint32_t required_table_lengths[REQUIRED_TABLE_COUNT];
301   const char *table_directory;
302   const char *table;
303
304   if (ttf_size < 12)
305     goto table_directory_invalid;
306   magic_bytes = read_int32(ttf);
307   /* TTF font file must start with these bytes. */
```

```
308   if (magic_bytes != 0x00010000)
309     goto not_ttf;
310   table_count = read_int16(ttf + 4);
311   table_directory = ttf + 12;
312   /* If the ttf is too small to fit all the tables. */
313   if (ttf_size < 12 + table_count * 16)
314     goto table_directory_invalid;
315   /*
316    * Initalise the required table lengths to zero so we know if one is not
317    * found.
318    */
319   for (i = 0; i < REQUIRED_TABLE_COUNT; i++)
320     required_table_lengths[i] = 0;
321   /* For each table directory. */
322   for (table = table_directory;
323       table < table_directory + table_count * 16;
324       table += 16) {
325     char tag[4];
326     uint32_t offset, length;
327     /* Read the tag, offset and length. */
328     memcpy(tag, table, 4);
329     offset = read_int32(table + 8);
330     length = read_int32(table + 12);
331     /* If we find the tag in the required table list, then save it. */
332     for (i = 0; i < REQUIRED_TABLE_COUNT; i++)
333       if (strcmp(tag, required_tables[i]) == 0) {
334         required_table_offsets[i] = offset;
335         required_table_lengths[i] = length;
336       }
337   }
338   /* For each required table. */
339   for (i = 0; i < REQUIRED_TABLE_COUNT; i++) {
340     /* If the table was not found. */
341     if (required_table_lengths[i] == 0)
342       goto table_no_exist;
343     /* Parse this table with the appropriate parser function. */
344     if (required_table_parsers[i](ttf + required_table_offsets[i],
345           required_table_lengths[i], info))
346       goto table_error;
347   }
348   /* Return zero on success and one on failure. */
349   free(ttf);
350   return 0;
351 not_ttf:
352   fprintf(stderr, "failed to read ttf: not a ttf\n");
353   free(ttf);
354   return 1;
355 table_directory_invalid:
356   fprintf(stderr, "ttf table directory invalid\n");
357   free(ttf);
358   return 1;
359 table_no_exist:
360   fprintf(stderr, "ttf does not have %s table\n", required_tables[i]);
361   free(ttf);
362   return 1;
363 table_error:
364   fprintf(stderr, "failed to read ttf %s table\n", required_tables[i]);
365   free(ttf);
366   return 1;
367 }
```

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 /* Check that the _font_name_ does not include any illegel characters. */
031 int
032 is_font_name_valid(const char *font_name)
033 {
034   unsigned char c;
035   /* An empty font name is invalid. */
036   if (*font_name == '\0')
037     return 1;
038   while (*font_name) {
039     c = *font_name;
040     /*
041      * '-' or 0-9 or '_' or A-Z or a-z
042      * these are the only legal characers.
043      */
044     if (c == 45 || (c >= 48 && c <= 57) || c == 95 || (c >= 65 && c <= 90)
045         || (c >= 97 && c <= 122)) {
046       font_name++;
047     } else {
048       return 0; /* Return 0 on success. */
049     }
050   }
051   return 1; /* Return 1 on failure. */
052 }
053
054 /* Tyy to convert _str_ to an integer stored in n. */
055 int
056 str_to_int(const char *str, int *n)
057 {
058   char *endptr;
059   *n = strtol(str, &endptr, 10);
060   if (*str == '\0' || *endptr != '\0')
061     return 1; /* Return 1 on failure. */
062   return 0; /* Return 0 on success. */
063 }
```

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, 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 test 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.

```
                                  2025

      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  2  3  4  5                      1  2                      1  2
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       3  4  5  6  7  8  9
13 14 15 16 17 18 19      10 11 12 13 14 15 16      10 11 12 13 14 15 16
20 21 22 23 24 25 26      17 18 19 20 21 22 23      17 18 19 20 21 22 23
27 28 29 30 31            24 25 26 27 28            24 25 26 27 28 29 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  3  4  5  6                1  2  3  4                         1
 7  8  9 10 11 12 13       5  6  7  8  9 10 11       2  3  4  5  6  7  8
14 15 16 17 18 19 20      12 13 14 15 16 17 18       9 10 11 12 13 14 15
21 22 23 24 25 26 27      19 20 21 22 23 24 25      16 17 18 19 20 21 22
28 29 30                  26 27 28 29 30 31         23 24 25 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  3  4  5  6                   1  2  3       1  2  3  4  5  6  7
 7  8  9 10 11 12 13       4  5  6  7  8  9 10       8  9 10 11 12 13 14
14 15 16 17 18 19 20      11 12 13 14 15 16 17      15 16 17 18 19 20 21
21 22 23 24 25 26 27      18 19 20 21 22 23 24      22 23 24 25 26 27 28
28 29 30 31               25 26 27 28 29 30 31      29 30

      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  2  3  4  5                      1  2       1  2  3  4  5  6  7
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       8  9 10 11 12 13 14
13 14 15 16 17 18 19      10 11 12 13 14 15 16      15 16 17 18 19 20 21
20 21 22 23 24 25 26      17 18 19 20 21 22 23      22 23 24 25 26 27 28
27 28 29 30 31            24 25 26 27 28 29 30      29 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.

1. Parsing Input the escape sequences for different types of text have been successful. When writing my report, I found it intuitive to surround text with starts (*) to make it bold or underscores (_) to make it italic. Headers and footnotes were also used effectively in my report.

2. Break Text Into Lines The line breaking algorithm has produced visually appealing paragraphs for my report. The insertion of spaces between words has been essential. Although hyphenation was not used in the report, it is a supported feature albeit one which is not so useful for single column A4 pages. Multiple font sizes in the same paragraph are supported but also not used in this report.

3. Break lines into pages. Lines have been fitted into pages in this report. In some rare occasions, orphan and widow lines generated by 'line_break' reduce the ascetics of a page, but this is a low-priority issue that does not violate any of the initial objectives. Footnotes have appeared correctly at the bottom of the intended page. A automatically generated contents page has been included in this report. A header has been added to the top of each page. When writing the report, I was able to customize the margin sizes to my needs. Standard compliant PDF files have been successfully written for the report with embedded fonts and image.

5.2 Improvements

In its current state, the software considers line breaking and page breaking to be two separate problems with 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.