UNIVERSITY OF DERBY
Derbyshire Business School

A project completed as part of the
requirements for the

BSc (Hons) Computer Studies

entitled

Document Formatting Systems

by

Viktor C. Pavlu
in the years 2003 – 2004

ABSTRACT

DOCUMENT FORMATTING SYSTEMS

April 24, 2004

VIKTOR C. PAVLU

Document formatting is the process of mapping information to layout. Since the first document formatting systems were developed in the 1960s, the use of computerized document formatting systems has steadily increased. Today document formatting is one of the most widespread applications of computers.

This report gives an overview of the historical development of document formatting systems and approaches introduced by document formatting systems. It studies and reviews employed methods and their contribution to the future of document formatting.

Biloba, a prototypical document formatting system that addresses special requirements in context of the Web, is developed to support the theoretical work. The possible value of a document formatting system tailored to problems inherent to the Web is formulated, the prototype is presented and analysed.

The work concludes with some ideas for the advancement of the system and future prospects related to the field of document formatting in general.

ACKNOWLEDGEMENTS

Above all, I would like to thank Carlton McDonald for supervising this project and providing many helpful advices during the course of the project.

I want to express my gratitude to my parents who made it possible for me to study abroad.

Further I want to thank Robert Sedgewick of Princeton University for providing valuable information by answering my inquiries about his work.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

Documents can transport ideas into another person's mind even more than spoken language can do, as written text bridges gaps of time and acquaintanceship.

Authors want to compose thoughts into documents instead of programming formatting commands to typeset their compositions. Unfortunately existing systems run counter to these preferences and force authors into typesetting.

One possible solution is to use interactive what-you-see-is-what-you-get (WYSIWYG) editors that hide the details of typesetting from their users. Evidently these systems add their own issues: documents are machine- and application dependent, often lack an inherent semantic structure, and interactively applying visual attributes leads to inconsistencies affecting the overall quality of the output.

This draws authors who seek high quality material to use systems such as troff and TEX. The markup that is required for these systems is seen as necessary evil to create publication quality material (Lipton & Sedgewick 1990).

Writing for the Web adds additional constraints to documents that current systems fail to address.

This project attempts a to find a compromise between high quality documents suitable for the Web and an easy to use interface allowing authors to concentrate on the writing.

The implementation of this compromise in the Biloba DF system is explained.

1.2 Aims and Objectives

The original aims and objectives as stated in the project proposal (see Appendix H) kept unchanged as the project progressed.

The purpose of the project *Document Formatting Systems* is to get knowledge of existing document formatting (DF) systems, their history and the techniques they use, in order to be able to implement a prototypical DF system for the Web that combines advantages of existing systems.

The author tries to find an answer to the question what a DF system has to fulfill to be, in particular, suitable for the Web.

The DF system prototype is intended to be used online in a collaborative environment, however it is not concerned with concurrency, network transport, security aspects or version control. Neither does it deal with the actual typesetting — this is left to the browser or other systems which create the final representation.

The objectives for this project thus are:

- to present and discuss the sequence of developments in the field of DF,
- to formulate the possible value of a DF system for the Web,
- to review and evaluate related work, and

- to implement, review and discuss the DF prototype.

This project was completed as part of the requirements for the BSc. (Hons) Computer Studies by Viktor C. Pavlu under the supervision of Carlton McDonald at the University of Derby in the years 2003-2004.

1.3 Report Overview

The report is organized as follows:

After establishing a common vocabulary, the sequence of developments in the field of document formatting is summarized reviewing influential systems.

A Document Formatting System for the Web (Chapter 3) outlines the problems a DF system has to address to be used in a collaborative environment where documents need to conform to a number of standards and rules inherent to documents on the Web. The result of this analysis leads to the requirements for a DF system for the Web.

The design and implementation of Biloba is explained in detail before the system is critically reviewed and compared to related work in the discussion.

After providing a list of unanswered questions as starting point for further research and improvements to the system, this final year project's conclusions are drawn.

1.4 What is a Document?

There are a number of different definitions of *document* in the academic world (Briet (1951), Spring (1991), Buckland (1997), ...) as well as in everyday use.

> "Ask any group of ten information scientists to define 'document' and you will get ten different answers." – *Spring (1991)*

According to the formal definition of a document given by the League of Nations' International Institute for Intellectual Cooperation (IIIC), a document is "Any source of information, in material form, capable of being used for reference or study or as an authority. Examples: manuscripts, printed matter, illustrations, diagrams, museum specimens, etc." (Buckland 1997)

This project further restricts the definition of *document* to those available electronically and in textual form. These documents may have other types of documents (images, tables, interactive programs, ...) embedded within them but the embedded objects are not treated as documents but atomic data.

1.5 What is Document Formatting?

Document formatting is the process of mapping information to layout. Before the use of computerized systems, this task was completed by the typesetter who relied on notes added by the editor to the margin of a manuscript. This is also the reason why programs used for this task are also referred to as typesetters.

There are two approaches towards formatting a document: the first, and as it seems, the more popular one today, is to use a graphical what-you-see-is-what-you-get (WYSIWYG) editor to apply physical formatting to a document. The second approach is to manually insert formatting commands into a plain text document and have a batch formatter produce the output.

While the latter method seems obsolete, it does have advantages over current WYSIWYG applications. Batch formatting systems usually use logical markup

instead of physical formatting and therefore give the documents an inherent structure. Changing the appearance of all headings in a document with logical structure is easily done on one central location, whereas in a document with physical formatting, it is tedious and error-prone, and often results in inconsistent layout.

CHAPTER 2

HISTORICAL DEVELOPMENTS IN DOCUMENT FORMATTING

This chapter summarizes influential developments in the field of document formatting. It is intended as a concise overview of important work. Suggestions for further reading are given to provide extensive review of a broader set of DF systems: Surveys of editing systems (Furuta, Scofield & Shaw 1982), (Meyrowitz & van Dam 1982*a*) & (Meyrowitz & van Dam 1982*b*), and (van Dam & Rice 1971), historical reviews (Furuta 1992), (André, Brüggemann-Klein, Furuta & Quint 1994), and (Myers 1998), and an annotated bibliography (Reid & Hanson 1981).

2.1 Automation of Writing

Word Processing was the result of gradual automation of the physical aspects of writing.

The initial steps towards automation were the invention of printing and Gutenberg's movable type at the end of the Middle Ages. Ottmar Mergenthaler further simplified the process of printing with Linotype, a machine for producing printing bars, in 1886. (*Linotype History: 1886 – 1899* n.d.)

The first major advance from manual writing as far as the individual was concerned was the typewriter, first built by Henry Mill in 1714. In 1961 IBM brought out

the Selectric typewriter. Instead of movable carriages and type bars it used a revolving typeball (often referred to as the "golfball"). This could print faster than the traditional typewriter and changing fonts could easily be done by replacing the typeball. (*IBM Archives: 1961* 1961)

Three years later, IBM introduced the Magnetic Tape Selectric Typewriter (MTST) which was the first typewriter equipped with a magnetic recording device making it possible to revise drafted work prior to actual printing. This gave rise to the concept known today as Word Processing.

Once documents could be edited on a computer system, the need for automated layout systems, or document formatting systems, arose.

2.2 *roff Family of Typesetters

2.2.1 RUNOFF

One of the first DF systems was RUNOFF, written by Jerome H. Saltzer for the Compatible Time-Sharing System (CTSS) in 1963 and 1964 to typeset his PhD thesis. (van Vleck 1995, Saltzer 1965)

It used formatting instructions that consisted of lines starting with a period. This syntax was chosen because it was common practice to denote formatting requests to the person who would perform the typesetting in the same way. (Fisher 1994)

Bob Morris ported RUNOFF to the General Electrics 635 platform and renamed it to *roff*. Doug McIlroy rewrote, simplified, and extended the system in 1969.

2.2.2 UNIX derivatives

With the growing popularity of the UNIX operating system, which could also run on more affordable machines, a DF system for the new environment was required, too. Joseph F. Ossanna wrote *new roff* (or *nroff*) for the UNIX operating system. Instead of trying to provide every single style a user might need, Ossanna made *nroff* programmable, so that new formatting tasks could be solved by programming in the *nroff* programming language. (Kernighan, Lesk & Ossanna 1978)

Later Ossanna wrote *troff* (typesetter roff) to drive a Wang Graphic Systems CAT typesetter at the Bell Laboratories. nroff and troff are basically the same programs; nroff is used for on-screen display ignoring font changes and other commands not suitable for screen display while troff was used to prepare the same document for printing. (Fisher 1994, Darwin 1984, Ossanna 1976)

By creating *nroff* as a programming language, Joseph Ossanna made it very flexible and extensible. Many preprocessors and extensions, so called macro packages, have been developed to address specific formatting needs. Popular preprocessors include *tbl* for tables, *eqn* for equations, *chem* for chemical structure diagrams, *grap* for graphs, and *pic* for graphics to name a few.

In 1979, Brian W. Kernighan modified troff to be device independent. (Kernighan 1981) James J. Clark completely rewrote the troff system as a GNU project where it is available for free under the name *groff*. (Fisher 1994)

Despite its age, troff is still widely used and continues to be the standard format for UNIX documentation.

2.2.3 Generic Coding with Troff

Using macros it was possible to start a new `.chapter` and have the numbering updated automatically along with formatting the chapter heading.

The `-ms` macros that accompanied troff were the first simple form of generic coding in documents. There were paragraphs that open a new section (`.LP`), normal paragraphs (`.PP`), and paragraphs used to format quotes (`.IP`). The first one would have a larger margin in the first line of text as commonly found after a new section. The last paragraph type indented output to set off the quote from the rest of the text. (Lesk 1978)

2.3 Generalized Markup

According to Goldfarb (1990), a presentation entitled "The separation of information content of documents from their format" by William Tunnicliffe, chairman of the Graphic Communications Association (GCA) Composition Committee, started the generic coding movement in September 1967 (Goldfarb 1990).

Generic coding uses descriptive tags like "heading" instead of control codes or macros that caused the document to be formatted in a particular way.

At that time, New York book designer Stanley Rice proposed the idea of universal catalog of parameterized 'editorial structure' tags. These trends were recognised by Norman Scharpf, director of the GCA, who established a generic coding project called 'GenCode Concept' which was later to become the GenCode committee. Unfortunately this project never came to a solution as the members could not agree on the common markup (Goldfarb 1990).

```
:document.
:line....:eline.
:quote.
:speech.Man is not the sum of what he has
  already, but rather the sum of what he
  does not yet have, of what he could have.
:source.Jean-Paul Sartre
:equote.
:line....:eline.
```

Figure 2.1. GML Source with use of Minimization

2.3.1 Generalized Markup Language

The Goldfarb, Mosher, Lorie research team at IBM invented the Generalized Markup Language (GML) based on the ideas of Tunnicliffe and Rice in 1969. The hierarchical markup used by GML was formally defined in document type definitions (DTD) which explicitly specified which elements were allowed to contain what other elements. (Goldfarb 1990)

Users could define their own markup tags enclosed in '`:`' and '`.`' character pairs. Figure 2.1 shows a sample GML document. Markup minimization permitted that certain end tags (as '`:speech.`' and '`:source.`' in Fig. 2.1) could be omitted if the context allows this without ambiguity. Parsers interpreting the DTD ensured documents contain valid structures only.

2.3.2 Standard Generalized Markup Language

In 1978, Goldfarb led a project for the American National Standards Institute (ANSI) to produce a standard text description language based on GML which became the Standard Generalized Markup Language (SGML) and was later transferred to the International Organization for Standardization (ISO) to become the international standard 'SGML ISO 8879/1986' in 1986 (Hopgood 2002).

```
<document>
<line>...</line>
<quote>
<speech>Man is not the sum of what he has
  already, but rather the sum of what he
  does not yet have, of what he could have.
<source>Jean-Paul Sartre
</quote>
<line>...</line>
```

Figure 2.2. SGML Source with use of Minimization

While most of the time counted to the group of declarative markup languages, SGML is really a standard to define markup languages. SGML applications, markup languages defined with the SGML, can be either of declarative or procedural nature. The most widely known application of SGML, the Hypertext Markup Language (HTML), has elements of both types: `<U>` for underlined text is procedural while `<H1>` for a level one header is declarative.

Figure 2.2 shows a sample SGML document. Although the standard allowed any character pairs to be used to enclose the tags as long as they are declared in the SGML declaration, it was common practice to use angle brackets to enclose the tags. A forward slash was used to denote closing tags.

SGML was and continues to be widely used for electronic documents by a large number of users, including Her Majesty's Stationery Office (legal text), the Commission of the European Communities (FORMEX), the US Department of Defense (ATOS), Oxford University Press (OED) (Barron 1989), and by the European Organization for Nuclear Research (CERN) where it became ancestral to the World Wide Web's Hypertext Markup Language HTML.

2.3.3 Extensible Markup Language

In 1996 a first working draft of the Extensible Markup Language (XML) was published. XML is an extremely simplified dialect of SGML targeted to make interoperability with parsers and other tools easier.

2.3.4 Document Style Semantics and Specification Language (DSSSL)

The international standard DSSSL, defined in ISO/IEC 10179/1996, describes a document processing language that can be used to transform, query, and style SGML documents (Technical Committee JTC 1/SC 34 1996).

2.4 The Bravo Interactive Editor

Designed in the 1970s by Butler Lampson and Charles Simonyi, the Bravo editor was the first system that had an approximation of the document in its final form displayed on a high-resolution screen for interactive editing. Bravo coined the term what-you-see-is-what-you-get (WYSIWYG) and continues to have a major influence on today's most widely used word processing product, Microsoft Word, which was also developed by the team around Simonyi. (Lampson 1976)

2.5 The Scribe System

Brian K. Reid picked upon the ideas of the Generalized Markup Language (GML) and generic coding in general to develop Scribe in the late 1970s at Carnegie-Mellon University.

The emphasis was on a simple input language that lets the user express the abstract objects within a document, leaving the responsibility for the appearance to

an expert "who establishes definitions mapping the logical structure to the printed page". (Furuta et al. 1982)

Although very simple forms of generic coding have already been implemented in *nroff/troff*, Scribe was the first DF system to use declarative markup throughout. With Scribe, authors described the logical structure of a document rather than a physical representation. (Reid 1980*a*)

There even were mechanisms for automatic creation of numbering of captions and list items, cross-references, a table of contents, and a bibliography. When Scribe finds a citation, it looks up the unique identifier in a bibliography database and adds the actual text of the citation into the document as well as creating a bibliography that is inserted at the end of the document. (Furuta et al. 1982, Reid & Walker 1980, Reid 1980*b*)

2.6 The TEX family

2.6.1 TEX

When revising the second volume of *The Art of Computer Programming* in the late seventies, Donald E. Knuth was disappointed with the typographic output he received. This motivated him to create a system for individuals to create documents with publication quality.

The original version, TEX78, was modified based on his and other users' experiences. He described TEX82 — the same version still in use today — in Knuth (1984) which continues to be the standard work on TEX.

METAFONT is the other part of Knuth's typesetting system. It is used to create fonts for TEX.

Similar to troff, TEX is a programming language rather than a fixed set of commands making it very flexible. However there is a set of basic commands (300 primitives and 600 control sequences), referred to as *Plain TEX*, predefined at startup to control the system and to create new commands.

The TEX User Group puts it concisely:

> ... TEX is a special-purpose programming language that is the centerpiece of a typesetting system that produces publication quality mathematics (and surrounding text), available to and usable by individuals. (TEX Users Group 2000)

2.6.2 LATEX

On top of the TEX language, Leslie Lamport implemented LATEX which uses a generic coding approach rather than physical formatting. The first publicly available version was LATEX 2.09 in 1985. In 1994 LATEX 2ϵ was released in response to many nonstandard enhancements to the original version. (Goossens n.d.)

"LATEX is now extremely popular in the scientific and academic communities, and it is used extensively in industry" (Lamport 1986).

2.7 Adobe Page Description Languages

2.7.1 PostScript

Since its inception in 1978, the PostScript page description language has enabled unprecedented control over the look and feel of printed documents in a device independent manner.

"The capabilities PostScript makes possible, have established it as the industry page description language standard" – *Adobe Systems (1999)*

2.7.2 Portable Document Format

The today very popular Portable Document Format (PDF) page description language uses the same Adobe Imaging Model as PostScript does, making conversions between the two easy. The main difference between the two languages is that PDF is made of static data structures suitable for random access while PostScript "is a simple interpretive programming language with powerful graphics capabilities" (Adobe Systems 1999).

2.8 The Lout System

The Lout DF system, first released in 1991 is a relatively new approach and draws upon to be the future of DF. Instead of making a DF system extensible by adding programming language capabilities, the author of Lout chose to design a functional high-level programming language for typesetting first and then to implement a DF system in that clean language. Here, 'clean' describes a language that has a regular structure with trivial syntax that is well suited to the problem domain and has no illogical restrictions or 'special case' confusions. "Examples of illogical context restrictions are extremely common in document formatting systems. FrameMaker permits objects to be rotated in certain contexts (when they are table entries, for example) but not others." A clean language would have no illogical restrictions, but be extensible, and tries to "find the best possible layout for the given content." (Kingston 1993*b*)

While the use of macros as means of extending an existing DF system is an excellent approach in theory, the author of this report has experienced that the lack of higher level semantics often leads to very long and eventually unstructured macros that are hard to debug. The overall process of extending these systems is very time consuming in practice.

According to Jeffrey H. Kingston (author of lout), the result of using a high-level language for DF is improved productivity allowing an unprecedented repertoire of advanced features presented to the non-expert user. He claims that "an equation formatting application, which may be difficult or impossible to add to other systems, can be written in Lout in a few days." (Kingston 1993*a*)

To the expert user lout is a functional programming language able to manipulate *objects* (text with vertical and horizontal alignment), *definitions* (of operators that take objects and return objects), *cross references*, and *galleys*. Galleys are used to insert text into the document in a place other than where it was entered (the same concept as diversions and traps in troff, or floating insertions in TEX). For example a footnote is entered inside a paragraph but it should appear at the bottom of the page. (Kingston 2000, Kingston 1992)

While to the non-expert user it is a markup language not any more complex than troff or TEX.

2.9 Classification of DF Systems

The following table 2.1 classifies the systems described in this chapter according to the markup technique they employed. It also serves as a condensed overview of features introduced.

Table 2.1. Classification of DF Systems

System	Year	Markup	Significance
RUNOFF	1964	procedural	Commands as physical characteristics
GML	1969	declarative	Declarative, hierarchical markup, DTDs
nroff	1976	procedural	Programmable through macros
Bravo	1978	procedural	What-you-see-is-what-you-get
TEX82	1982	procedural	Publication Quality Mathematics
LATEX	1985	declarative	Generic Coding with TEX
ISO SGML	1986	both	Standard to define markup, widely adopted
PostScript	1987	procedural	Industry-Standard Page Description Language
Lout	1991	declarative	Typesetting Programming Language
DSSSL	1996	procedural	Style Sheet Standard for SGML
XML	1996	declarative	Simplified SGML, simplified parsers

CHAPTER 3

A DOCUMENT FORMATTING SYSTEM FOR THE WEB

3.1 Problems with documents on the Web

A document formatting system for the Web is one whose features aim at problems peculiar to writing for the Web. The following paragraphs outline some of those problems:

- the layout of collaboratively edited documents needs to be consistent
- usability studies (Morkes & Nielsen (1997), Lewenstein, Edwards & Tatar (2003)) have shown that most users scan the page, reading only a fraction of the text; reading from screen is significantly slower than from paper, and documents on the Web should have half the word count of their paper equivalents (Nielsen, Schemenaur & Fox 1998, Petersen 2001)
- documents need to be small in terms of storage for reduced bandwidth requirements and faster load times
- different devices and browsers are used to view the documents (sometimes they are not viewed at all, but read out by the browser)
- documents should to be accessible to people with disabilities

- varying representations of one document are required for browsing, syndication, print, ...
- Web sites are expected to be updated frequently
- special attributes innate to the Web (metadata for search engines, navigation through hyperlinks, ...) need to be connected with documents

3.2 Consistency

> "Part of web page design includes the consistent use of textual elements" – *Nielsen et al. (1998)*

In order to achieve a consistent look and feel for a Web site it is vital that the author sticks to the same rules defining the layout throughout. This can be tedious and error-prone for one author, but it is even harder in a collaborative environment where many different authors work on the same set of documents — a common problem, not only for web-based authoring systems.

A possible solution to this problem is to use declarative instead of procedural markup.

Declarative markup as opposed to procedural markup expresses the logical structure rather than physical attributes. While something is marked as a *caption* (without defining the appearance of a *caption*) using the declarative approach, the same text might be marked as *twelve points in size and to be typeset in boldface using a sans-serif font* when using the procedural approach.

Two things are apparent: Procedural markup allows total control over the typeset output while declarative markup separates content from representation and takes the responsibility over the typeset output away from the writer.

The author believes that declarative markup leaves less room for inconsistency — either a set of words is a caption or it is not; instead of the many different forms possible with procedural markup (same size and font as other captions, but different font-weight, etc.). Therefore declarative markup is a very clean way to address the problem of inconsistency.

Declarative markup already helps if only one author is creating a document as his sense of a nicely formatted caption may change over time, but it helps even more if more authors are involved, as their taste in layout may differ altogether.

> "Procedural markup is also inflexible. If the user decides to change the style of his document [...] he will need to repeat the markup process to reflect the changes." – *Reid (1980*b*)*

The final look and feel of the documents is determined by a set of rules mapping abstract documents to their tangible representation. These rules are possibly created by layout experts with attention to findings in the field of usability and interface design and need only be changed once for all occurrences of a *caption*.

Declarative markup is also referred to as structural markup or generic coded markup while procedural markup is also known as presentational or physical markup. Historically, the procedural approach was first (See Table 2.1, Chapter 2).

3.3 Multiple Output Formats

The Web is not an online simulation of desktop publishing, elements of a hypertext document are not purely visual. The author therefore thinks that approaching design with WYSIWYG editors, purposely designed to hide the details of markup

from the editor, prevents the editor from deciding a document's structure from a conceptual point of view and thus fails to make the most of the possibilities. Additionally, the author draws from his experience that the markup created by WYSIWYG editors tends to be presentational only and unnecessary bloated, resulting in longer load times.

Users with disabilities or those who need to keep their hands free while interacting with a browser, eventually use voice technology currently being incorporated into browsers (Axelsson 2001, Loney & Festa 2003). But also syndication (Nottingham (2003) and Winer (2003)), printing and archiving requires different representations of a single document — hence the need for multiple output formats and another reason for separating the content from its representation.

3.4 Conforming to Standards

Users want to access content using the user agent of their choice. This not only includes the device, operating system, or browser version, but also window size, screen resolution, colour depth, ... and that is even assuming a graphical display, which is not available to some users.

Conforming to established standards ensures compatibility with browsers in use.

- XML 1.0, W3C Recommendation, Bray (2000),
- XHTML 1.1, W3C Recommendation, Pemberton (2001),
- Cascading Style Sheets, W3C Recommendation, Bos (1998), and
- Web Content Accessibility Guidelines, W3C Recommendation, Chisholm (1999)

The latter is not a standard per se, but a set of guidelines to make documents more accessible to all users "whatever user agent they are using [...] or constraints they may be operating under [...]" (Chisholm 1999).

3.5 Frequent Updates

A survey (Rhodes 1998) has shown that the credibility of a Web site depends to a large amount on the freshness of the information presented. Frequent updates to a document should be alleviated, ideally the document should be editable from its current location within the browser.

3.6 Web Artifacts

Users should be informed about updates, usually done at the bottom of the page. Apart from this, a DF system for the Web needs to manage most of the information peculiar to documents on the Web (e.g. the used text encoding, keywords for search engines, and other kinds of metadata). It also has to permit an easy way of linking to other hypertext documents.

3.7 Suitability of Existing Systems

The number of existing DF system that meet this requirements is limited. The industry standard page description languages, PostScript and PDF, serve a completely different need. They were designed to have full control over produced output in a device-independent manner, so programmers would not have to adapt their programmes' output to different printers, but use a standard page description language as means of communicating the graphical representation.

This is contrary to the idea of the Web as a networked collection of inter-connected documents where fast access is more important than exact placement, size, or fonts. Nevertheless it is sometimes necessary to provide documents in PDF format for archiving (all external objects, such as pictures are included in a single file) or printing (to retain the exact layout), so every system serves a special purpose.

Being amenable to transfer without special encoding, easy processability, and coherent markup used throughout to allow for simple information retrieval tools, among other points mentioned in this chapter argue in favor of a generic coding approach.

Multiple representations for different uses underscore the need for a declarative system, which allows easy transforming between physical representations.

As SGML was widely used at the CERN, where the Web was born, the Hypertext Markup Language has been strongly influenced by SGML. But due to the explicit nature of SGML-based markup and the aforementioned best practices that introduce their own set of limitations, it is tedious to be created by hand, hence the need for a system that provides an interface which is as comfortable to humans, as SGML-applications are to parsers.

3.8 Summarized Requirements

The purpose of a DF system for the Web thus is to provide an easy to use interface to collaboratively edit documents that are independent of their representation. Authors should be freed of the responsibility over the layout and supported in composing their thoughts into documents while at the same time the DF system endorses best practices of writing for the Web and ensures standards compliant, minimal markup with a coherent look and feel.

Ideally the DF system can be used from within a browser without resorting to special browser dependent techniques. All the processing could be done on the server side, which also ensures that users are working with the same version of the DF system.

All these factors influenced the design of the Biloba DF system.

CHAPTER 4

THE BILOBA DF SYSTEM

4.1 Overview

The Biloba DF system is an extensible system developed to meet the requirements stated in section 3.8. It consists of two main parts: the *STX Parser*, and a set of *Output Writers*. Data flow diagrams part of the analysis can be found in Appendix A. As shown in Figure 4.1 there are two interfaces to the system: the Web interface that makes it possible to run Biloba on the Web server formatting documents submitted via HTTP requests and a command-line interface so that the system can be used off-line as well. For details on how to use Biloba please refer to the *User's Guide* in appendix C.

Splitting the system in two separate parts, the parser and the output writer, permits maximum output format flexibility.

The Biloba DF system is written in the REBOL programming language. For details see http://www.rebol.com/. The documented source code is available on the accompanying CD.

4.2 Parser

The parser splits the source form into an ordered list of lines to transform the source form according to the structured text rules (see App. E) into an abstract

Figure 4.1. Biloba Architecture

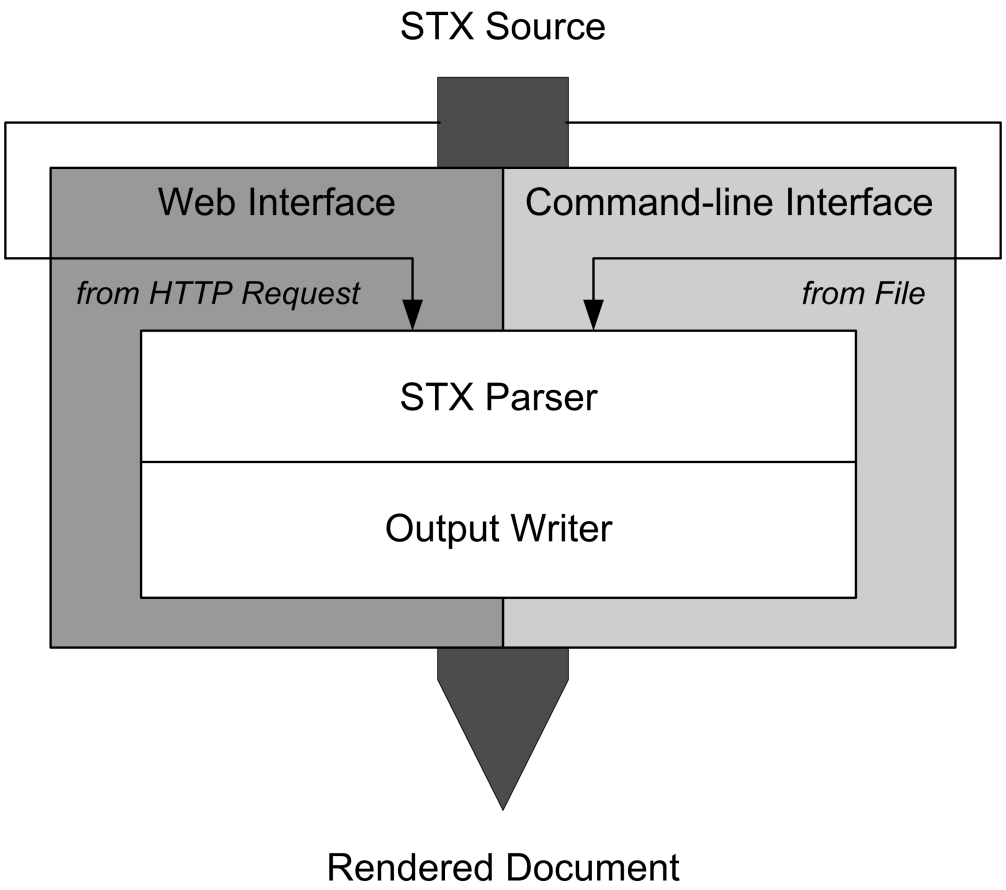


Figure 4.2. Node Definition in BNF

```
<document>        ::= ‘[’ document ‘[’ <content> ‘]’ ‘]’
<content>         ::= (*empty*)
                    | <node-or-string> <content>
<node-or-string   ::= <node>
                    | <string>
<node>            ::= <node-identifier> ‘[’ <content> ‘]’
<node-identifier> ::= (*alphanumeric atom*)
<string>          ::= ‘"’ (*any character*) ‘"’
```

Figure 4.3. Sample Quote in Structured Text Source Form

```
...

"Man is not the sum of what he has already,
but rather the sum of what he does not yet have,
of what he could have." --Jean-Paul Sartre

...
```

tree representation. For efficiency, the document is parsed one line at a time with a lookbehind buffer of one line which is further reduced to the metrics *indentation* and *document element type*.

Every structural element[1] becomes a node in this tree. Nodes consist of a node identifier followed by an optionally empty list of further nodes and strings representing the node's content (Fig. 4.2). This definition allows nodes to have mixed content, which is required for inline formatting.

The root of the tree is the implicit node *document*.

Figure 4.3 shows a sample source for a quote that is part of a larger document. For illustration purposes the rest of the document is only indicated by ‘...’.

[1] captions, paragraphs, list items, ...

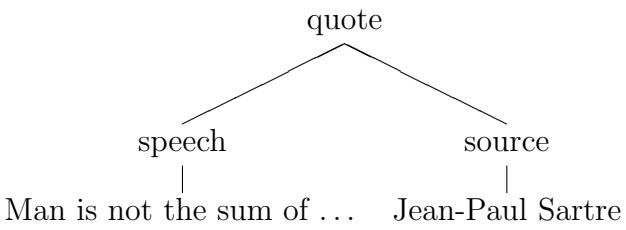


Figure 4.4. Tree Representation of the Sample Source

After parsing the source into the paragraph tree, the example is internally represented as illustrated in Figure 4.4. The document element *quote* consists of two sub-nodes, *speech* and *source*, representing what was said and by whom respectively. The *quote* is contained in the list of sub-nodes of its parent node which is eventually a paragraph or the toplevel node *document*.

4.3 Output Writer

Creating the tangible representation from the internal representation is achieved by a depth-first tree traversal done by the appropriate output writer transforming the nodes into markup.

Currently available output formats are

- a *debug* output writer that dumps the tree in REBOL-readable form
- a *html* output writer that creates valid XHTML to be displayed in a browser
- a *xml* output writer that creates valid XML to be used as source for further processing, and
- a *tex* output writer that translates the tree into TEX commands to be converted into PDF, DVI, PostScript, ... by the TEX program

Figure 4.5. Sample Quote in XML format

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<document>
 <line>...</line>
 <quote>
  <speech>Man is not the sum of what he has
    already, but rather the sum of what he
    does not yet have, of what he could have.</speech>
  <source>Jean-Paul Sartre</source>
 </quote>
 <line>...</line>
</document>
```

Transforming the quote in Figure 4.3 using the *xml* output writer yields the XML document in Figure 4.5. The node identifiers are used as tag names. This XML form can be used to query the data in structured text documents or to create different representations using a transformation language such as XSLT.

Rendering the same quote (Fig. 4.3) using the default *html* output writer creates a document according to the W3C's XHTML 1.0 Strict document type definition. Figure 4.6 shows the document with style sheet information and the document footer being removed for clarity.

When rendered by the browser, the quote eventually appears as illustrated in Figure 4.7. Note that the ultimate layout of the document largely depends on the browser used to view the document, as well as the style sheet chosen from a predefined set embedded into the generated document.

It is possible to add more output formats by writing additional output writers for Biloba. For information on how to do this, please refer to the *Expert's Guide* in Appendix D.

Figure 4.6. Sample Quote in XHTML 1.0 Strict Conformant Format

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
<title>Biloba STX Document</title>
<!-- style sheet info removed -->
</head>
<body>
 <p>...</p>
 <blockquote><div><q>Man is not the sum of what he has already,
   but rather the sum of what he does not yet have, of what
   he could have.</q></div>
 <div class="source">&ndash; Jean-Paul Sartre</div>
 </blockquote>
 <p>...</p>
<!-- footer removed -->

</body>
</html>
```

Figure 4.7. Sample Quote in XHTML 1.0 Strict Conformant Format, Rendered by Browser

...

"Man is not the sum of what he has already, but rather the sum of what he does not yet have, of what he could have."

— Jean-Paul Sartre

...

4.4 Figure Modules

Text that is spontaneously indented, that is indented without a prior heading, is per definition, a figure.

The most basic type of figure is a verbatim area. Everything entered will appear exactly as typed in the source. Physical line breaks as well as blanks are preserved — no formatting is applied.

This is the default figure type, but by explicitly specifying the type of figure with '`#type:<figurename>`' any other figure module can be called to interpret the text in the figure.

This feature is applied to implement images. A special-purpose figure module called *image* is called. It interprets the text as path to an image which is inserted into the document.

Figure modules are designed with extendability in mind. Developers are encouraged to implement their own modules to perform syntax highlighting, formatting of mathematical formulae or music chores, etc. For information on how to write a figure module, please refer to the *Expert's Guide* in Appendix D.

4.5 Cache

The Web interface to Biloba has a caching mechanism built in: before trying to process a document, it is verified that the source has been modified since it was last requested. This is done by comparing a message digest 5 (MD5) hash value of the source with a prior saved hash value.

If the hash codes match, a prior saved version of the requested document in rendered form is served.

If the hash codes differ, the stored snapshots of the document are out of date and the document needs to be re-processed by parser and output writer. The new hash value is stored along with the rendered document for later retrieval.

4.6 The Structured Text Format

Structured text (STX), introduced by the Zope Project (2001), is a "[plain] text [format] that uses simple symbology and indentation to indicate the structure of a document". The original aim of the STX format was to have an easy readable text format that can also be transformed into a nicely formatted output. Apart from simple text structuring and in-line rules (i.e. '`*emphasized text*`' for **emphasized text**) there were few definitions.

The structured text rules had to be extended so that the most common used text elements can be expressed without ambiguity while trying to reduce explicit markup to an absolute minimum. This design takes the responsibility of the layout of a document away from the author.

In order to find out which formatting elements are commonly used and which are of theoretical interest, as used on very rare occasions only, a number of recent (published within the last 10 years) papers from different academic fields have been inspected. The research confirmed what common sense of good style indicated before: sections and subsections are nested to at most three levels, lists to at most two levels, figures have a caption, ... It was remarkable that only very few papers used more than one kind of inline formatting — one style was used throughout, most times to set off new terms from the rest of the text.

Additionally, depending on the academic field, specialized objects were used (mathematical formulae, tabular data, chemical symbols, music scores, ...).

Rules that define how certain text elements are expressed in plain text were developed. As more text elements were added to the definition, the existing rules were continuously refined so that the result was a consistent set of rules that remedy ambiguity but at the same time are as near to common usage of plain text as possible. This makes them easy to follow. The Biloba STX rules can be found in the Appendix E.

CHAPTER 5

DISCUSSION

5.1 Review in Context of Related Work

5.1.1 NOTECH

The project's research revealed NOTECH: Typesetting without Formatting, a project that is closely related to Biloba as both use plain text with markup reduced to an absolute minimum as source and try to infer the appropriate formatting from the indentation and contents of the document. The NOTECH system and report were never published (personal communication with Robert Sedgewick, December 2003) but this author could get unpublished material by contacting Robert Sedgewick in writing (Lipton & Sedgewick 1990).

The main difference between the two systems is that NOTECH detects physical formatting applied to the source in order to reflect the same physical formatting in the typeset output. A centered header or a horizontal rule that spans half the page in the source will be exactly transformed into a centered header and the same rule in the output. If the author inserts enough blanks before a word to make it seem aligned with the center or the right margin, NOTECH will make it a centered or right aligned header, as illustrated by a short excerpt of Lipton & Sedgewick (1990) in source form (Fig. 5.1 and in rendered form (Fig. 5.2).

Figure 5.1. Short Excerpt of "NOTECH: Typesetting Without Formatting" (Lipton & Sedgewick 1990), Source Form

```
[...] the text.  If the textual
material seems to be centered or aligned with the right margin, notech
will do so, as follows:

Primary header

  Subheader

                          Center header

                                                              Right header

If the first nonblank line in a document is a center header, it is
assumed to be a title, and is typeset in larger, boldface type.  If it
is followed by a group of centered lines, they are [...]
```

Figure 5.2. Short Excerpt of "NOTECH: Typesetting Without Formatting" (Lipton & Sedgewick 1990), Rendered Form

the text. If the textual material seems to be centered or aligned with the right margin, `NOTECH` will do so, as follows:

Primary header

Subheader

Center header

Right header

If the first nonblank line in a document is a center header, it is assumed to be a title, and is typeset in larger, boldface type. If it is followed by a group of centered lines, they are

Biloba is more restrictive in this respect, every text element is defined by its relative position to its previous text element. Indenting a single line of text beyond the current level of indentation, i.e. up to what seems to be the center of the page, will not result in a centered header as shown in Figure 5.2 but will be regarded as spontaneous indented line. Consequently the lines "Center header" and "Right header" will be rendered as part of a figure.

Clearly this makes it harder for the user to create centered headers. While inhibiting at first sight, it really is not a burden, but rather a necessity to separate the content from its representation and to create consistent and well structured documents. Documents in Biloba are structured into trees with headers and subheaders introducing the different levels. If a user wants to have all level 2 headers to be centered, this has to be defined in the style sheet once and for all level 2 headers, and not by centering the header in the source with the danger of reintroducing inconsistencies.

Users of Biloba have no direct control over the typeset output. The author sees the mechanisms that enforce consistency as an advantage but apparently they make Biloba unsuitable for applications where the formatting needs to be based on what pleases the eye. For such applications WYSIWYG systems are far superior to non-interactive systems in general.

5.1.2 Zope STX

The Structured Text (STX) Project, part of the larger Zope Project (Zope Project 2001) created rules for simple document elements to be detected and then transformed into HTML. The basic idea is the same as in this project, however the author believes that the rules created for Biloba are less intrusive when composing documents.

Figures, for example, must be spontaneously indented in both systems but Zope STX additionally requires them to have a paragraph that ends in double colons ('::') in front of them. This limitation is required to unambiguously discern figures from subordinate text elements which Biloba does by restricting headers to single lines. It is also worth noting that no two figures can be immediately after each other in Zope STX, the author always has to insert a paragraph followed by '::'.

Motivated by the limitations of the original STX, the Zope Community started to develop the "next generation" of Structured Text which is now easier to handle by authors.

5.2 Evaluation

In its current form, Biloba handles all text elements planned to be developed for this project. This includes most, but not all of the rules stated in Appendix E. According to the large set of unit and regression tests created to verify the system, these rules are correctly implemented. While this verification method was a great help developing the system, it does neither verify that the system can be used by following commonly agreed upon rules of style nor that the rules are complete enough to create real documents.

To get this kind of feedback, the author made the DF system available to former and present colleagues who adopted Biloba to be used in ongoing projects. As a proof of concept, this report and the project's Web site were created using the Biloba DF system.

Based on the feedback gained from these real-world uses, strengths and weaknesses of the Biloba DF system are summarized in the following paragraphs.

5.2.1 Strengths of Biloba

Most of the investigated systems have in common that they are designed to be an easy to use interface to create nicely formatted documents, but they ignore the document's underlying semantics and logical structure. As a result, those systems are limited to a small number of output formats their developers planned for in the first place. Some of the systems even restricted the documents to a single format that was hard-coded into the DF system.

Organising Biloba around the concepts of natural inferences about document structure and a logical representation of the documents provides for an easy to use interface to create valid documents in a large number of formats. Freeing authors of the responsibility over the markup they create ensures valid, standards compliant markup is created at all times.

Using the Biloba DF system for actual report writing tasks reaffirmed that the rules derived from common-sense, commonly accepted style rules free authors from explicitly worrying about formatting commands but enables authors to type documents without thinking about anything but the content they are creating.

Typesetting this report was less laboured than creating the interim report or any of the regular progress reports.

By implementing Biloba as batch formatting system, it is possible to deploy the DF system from directly within the browser without the need for special-purpose software. Users type their documents in a 'textarea' form field and submit the document to be processed at the server. This decision also allows that Biloba is used from the command line for offline preparation of Web sites or other documents.

Finally, using plain text as input format has these additional advantages over text with markup or even binary formats:

- Plain text is amenable to transfer between machines without special encoding or loss of information
- Plain text files are independent of the program they once were created for. Users do not have to fear that documents they created become obsolete.
- Plain text can be edited with a large number of existing tools. Users do not have to change their current editing habits to use Biloba.
- In particular, operating systems as Unix provide a wide range of tools that operate on plain text and are less useful when the text is interspersed with formatting commands.
- Text documents can be put under version control with the benefit of differential updates. RCS, and therefore CVS (which uses RCS) too, can only store full copies of changing binary files. (Note that this limitation does not apply to the recently released version control system Subversion, which also stores binary files differentially.) (Collins-Sussman, Fitzpatrick & Pilato 2004)

5.2.2 Weaknesses of Biloba

The thing most often complained about was the complicated setup, which is due to the flexible nature of the system — it can be deployed in combination with any CGI-capable Web server on more than 40 platforms[1] provided it is correctly configured to work in the chosen environment. A setup wizard that installs and configures the Apache HTTP Server with Biloba is envisaged for further work.

One result of Biloba's philosophy is that it is limited to document elements which can be unambiguously inferred from common-sense rules. Another limitation re-

[1]subject to the availability of a REBOL interpreter for the specific platform. For details see http://www.rebol.com/prod-core.html.

sults from the fact that Biloba does not offer full control over the physical layout or macro programming capabilities (like troff, TEX, and Lout) which could be used to extend and adapt the system to individual needs.

Clearly this poses a drastic limit to what can be done with Biloba. In essence it reduces Biloba to formatting tasks already implemented in the system and in its current form the number of formatting tasks is reduced to a very restricted, but useable set.

The mechanism of figure modules described in Section 4.4 were developed with extendability in mind. However, as it stands these modules can only be used to format text found in figures. Adding functionality outside figures requires re-programming of the system.

Parsing of plain text files based on commonly accepted style rules obviously lacks a coherent theoretical base. The rules proposed as part of this project try to address this problem by finding a common denominator. But after all, it is possible that different users have different ideas of how a certain text element is to be typed in the source file.

Another objection from a completely different kind of user is that not using a graphical interface is just too "old fashioned" (Lipton & Sedgewick 1990). The immediate feedback in exactly the form of the final output as found in WYSIWYG editors is superior to non-interactive systems, but current WYSIWYG systems neglect the structure and semantical aspects of documents so the resulting documents are rarely more than *only* what can be seen. The author believes that this would be a major disadvantage if used for the Web where exact control over the layout is subordinate to structure, processability, load time, and at most: content — which an easy to use interface allows to concentrate on.

CHAPTER 6

CONCLUSION

In scope of this final year project an investigation of existing document formatting systems and their contributions to the future of document formatting has been carried out. The report gave a concise overview of developments that continue to have an influence on today's systems.

It has been shown what the requirements for a document formatting system tailored to the Web are and how these requirements can possibly be addressed. A set of rules that specify common document elements have been proposed, the theoretical work was supported by the implementation of Biloba, a document formatting system prototype that tries to fulfill these needs.

Biloba infers the logical structure of a document from parsing a plain text document by rules based on commonly accepted style. Internally, the documents are declaratively represented as trees, since this reflects their innate structure best. A single-pass traversal of this tree, done by a specific Output Writer, transforms its nodes into valid markup of the chosen format that conforms to useability and accessibility guidelines.

It was argued that this approach not only lets the author concentrate on writing instead of programming formatting commands, but also endorses established best practices and reduces the risk of inconsistencies while supporting the author in creating high-quality content for a Web audience.

Conversely, this approach limits Biloba to document elements which can be unambiguously inferred from common-sense rules. Also, there is a limit of what can be done with declarative markup: adding further document elements requires extending the system, whereas full control over the output, as offered by procedural markup, in combination with macro programming capabilities could be used to extend and adapt the system to individual needs.

The review of Biloba underscored that the prototype, however incomplete it is, can already be employed for a relatively large number of formatting tasks with different formats and in different environments. What follows are open questions that provide for opportunities of further research worthwhile to pursue.

6.1 Extending Biloba

Text elements that were already specified but not implemented by the parser include abbreviations that are automatically looked up in a project-wide abbreviation database, footnotes, referencing, automatic index generation, mathematical formulae, and other special symbols often used in academic or technical documents, could be added to future versions of Biloba.

The author plans to continue the development of Biloba as open source project. The source code will be publicly available through the project's Web site[1] for others to use it and eventually contribute to the development.

The further plan includes implementing a figure module that uses the syntax files of the 'vi' editor to achieve syntax highlighting for source code based on the rich set of existing syntax files and to implement an additional output writer for PDF

[1] http://plain.at/vpavlu/BilobaSTX/

documents. Other output formats that seem interesting in the context of the Web include DocBook XML as well as the syndication formats RSS and Atom.

As mentioned in the discussion (Sect. 5.2.2), parsing of plain text based on commonly accepted style rules lacks a coherent theoretical base. A viable solution to this problem is to formalize the parsing of the plain text sources and to rewrite the Biloba DF system as an interpreter for parsing rules defined in a formal language. The system could be tailored to individual user needs and additional document elements by defining rules in the formal language.

Although this method possibly leads to different rule sets with source documents being incompatible, this idea is worthwhile to pick up for further work as extendability and maintainability would vastly improve if the rules to parse the source files were defined in a formal language that is interpreted by Biloba rather than directly hard-coded into the system.

6.2 Broader Perspective

Putting guidelines of writing for the Web directly into the context of a DF system seems promising. Biloba already offers endorsement of recommendations on a syntactical level but the system could be extended to include guidelines on a semantic level to further support authors in their tasks.

A thesaurus system could be added that recommends replacing complex words with simpler ones, large documents could be split up into multiple hyperlinked documents for easier access, text mining methods could be used to automatically create summaries and keyword lists for users or search engines, etc. (Guidelines: Morkes & Nielsen (1997), Nielsen et al. (1998), Nielsen (1999), Nielsen (2000), and Kilian (2000))

Based on the knowledge gained in the course of this project the author believes that advancements in the field of non-interactive DF systems will also influence interactive WYSIWYG systems, which seem to dominate already, though sometimes inferior to non-interactive typesetting systems. Most considerable for a document formatting system for the Web, these systems often deliberately ignore the semantics of a document.

He further thinks that merging the flexibility achieved through declarative formatting, programmability, and extendability — features currently most often only found in non-interactive systems — with interactive, graphical user interfaces and recent insights in the field of human computer interaction such as demonstrational interfaces and programming by example to yield a system that unifies the advantages of both worlds is strongly desirable and could be successfully incorporated into formatting systems for collaborative and dynamic environments such as the Web.

During research, the author has come over a number of experiments (Quint & Vatton (1986), Graham, Harrison & Munson (1992), Cowan, Mackie, Pianosi & de V. Smit (1991), Furuta, Quint & André (1988), and Murata & Hayashi (1992)) that try to combine structured documents with interactive editing.

> "Implementing a structured documents interactive editor/formatter is still an open challenge" – *Roisin & Vatton (1993)*

Albeit the unquestionable value of a system that unifies advantages of both DF worlds, the author believes, that currently it is a better idea to have a working system that does one thing well. The Biloba document formatting system is to be understood as this working compromise.

BIBLIOGRAPHY

Adobe Systems, I., ed. (1999), *PostScript Language Reference*, third edn, Addison-Wesley.

André, J., Brüggemann-Klein, A., Furuta, R. & Quint, V. (1994), 'History of document processing'.
URL: *citeseer.nj.nec.com/333125.html [accessed 17 October, 2003]*

Axelsson, J. e. a. (2001), 'XHTML+Voice Profile 1.0'.
URL: *http://www.w3.org/TR/2001/NOTE-xhtml+voice-20011221 [accessed 29 March, 2004]*

Barron, D. W. (1989), 'Why use SGML?', *Electronic Publishing - Origination, Dissemination, and Design* **2**(1), 3–24.
URL: *http://portal.acm.org/citation.cfm?id=71826 [accessed 7 November, 2004]*

Bos, B. e. a. (1998), 'Cascading Style Sheets (CSS) layer 2, W3C recommendation'.
URL: *http://www.w3.org/TR/1998/REC-CSS2-19980512 [accessed 4 October, 2003]*

Bray, T. e. a. (2000), 'Extensible Markup Language (XML) 1.0, W3C recommendation'.
URL: *http://www.w3.org/TR/2000/REC-xml-20001006.pdf [accessed 4 October, 2003]*

Briet, S. (1951), *Qu'est-ce que la documentation*, Paris: Éditions Documentaires Industrielles et Techniques (EDIT).

Buckland, M. (1997), 'What is a "document"?', *Journal of the American Society of Information Science* **48**(9), 804–809.

Chisholm, W. e. a. (1999), 'Web Content Accessibility Guidelines 1.0, W3C Recommendation'.
URL: *http://www.w3.org/TR/WCAG10/ [accessed 14 March, 2004]*

Collins-Sussman, B., Fitzpatrick, B. W. & Pilato, C. M. (2004), *Version Control with Subversion*, O'Reilly and Associates.
URL: *http://svnbook.red-bean.com/ [accessed 22 April, 2004]*

Cowan, D. D., Mackie, E. W., Pianosi, G. M. & de V. Smit, G. (1991), 'Rita - an editor and user interface for manipulating structured documents', *Electronic Publishing – Origination, Dissemination, and Design* **4**(3), 125–150.
URL: *citeseer.nj.nec.com/cowan91rita.html [accessed 22 October, 2003]*

Darwin, I. F. (1984), 'A history of UNIX before berkeley: UNIX evolution: 1975-1984'.

Fisher, T. A. (1994), *groff Manual Page*.
URL: *http://www.cs.pdx.edu/~trent/gnu/groff/groff.html*, *[accessed 2 February, 2004]*

Furuta, R. K. (1992), 'Important papers in the history of document preparation systems: basic sources', *Electronic Publishing – Origination, Dissemination, and Design* **5**(1), 19–44.
URL: *citeseer.nj.nec.com/furuta92important.html [accessed 19 October, 2003]*

Furuta, R., Quint, V. & André, J. (1988), 'Interactively editing structured documents', *Electronic Publishing - Origination, Dissemination, and Design* **1**(1), 19–44.

Furuta, R., Scofield, J. & Shaw, A. (1982), 'Document formatting systems: Survey, concepts, and issues', *ACM Computing Surveys* **14**(3), 417–472.
URL: *http://portal.acm.org/citation.cfm?id=356891 [accessed 6 March, 2004]*

Goldfarb, C. F. (1990), *The SGML Handbook*, Clarendon Press, Oxford.

Goossens, M. (n.d.), 'LaTeX, an overview'.
URL: *citeseer.nj.nec.com/516927.html [accessed 25 January, 2004]*

Graham, S. L., Harrison, M. A. & Munson, E. V. (1992), The Proteus presentation system, *in* 'Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments', ACM Press, pp. 130–138.
URL: *citeseer.nj.nec.com/graham92proteus.html [accessed 20 October, 2003]*

Hopgood, B. (2002), 'History of SGML'.
URL: *http://wwwcms.brookes.ac.uk/modules/web_tech/p08770/s1b_sgml/overview.htm [accessed 18 February, 2004]*

IBM Archives: 1961 (1961).
URL: *http://www.ibm.com/ibm/history/history/year_1961.html [accessed 31 January, 2004]*

Kernighan, B. W. (1981), *A Typesetter-Independent TROFF*, Bell Labs.

Kernighan, B. W., Lesk, M. E. & Ossanna, J. F. (1978), 'UNIX time-sharing system: Document preparation', *Technical Journal* **57**(6), 2115–2135.

Kilian, C. (2000), *Writing for the Web*, Self Counsel Press.

Kingston, J. H. (1992), 'A new approach to document formatting'.
URL: *http://snark.niif.spb.su/~uwe/lout/overview.ps.gz [accessed 2 March, 2004]*

Kingston, J. H. (1993*a*), 'The Design and Implementation of the Lout Document Formatting Language', *Software - Practice and Experience* **23**(9), 1001–1041.
URL: *citeseer.nj.nec.com/kingston93design.html [accessed 2 March, 2004]*

Kingston, J. H. (1993*b*), 'The future of document formatting'.
URL: *citeseer.nj.nec.com/340599.html [accessed 2 March, 2004]*

Kingston, J. H. (2000), 'An Expert's Guide to the Lout Document Formatting System'.
URL: *citeseer.nj.nec.com/kingston00experts.html [accessed 2 March, 2004]*

Knuth, D. E. (1984), *The TEXbook*, Addison-Wesley.

Lamport, L. (1986), *LATEX: A Document Preparation System*, Addison-Wesley.

Lampson, B. W. (1976), *Bravo Manual in Alto User's Handbook*.

Lesk, M. E. (1978), *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff*, AT&T Bell Laboratories.

Lewenstein, M., Edwards, G. & Tatar, D. (2003), 'Stanford Poynter Project'.
URL: *http://www.poynterextra.org/et/i.htm [accessed 30 March, 2004]*

Linotype History: 1886 – 1899 (n.d.).
URL: *http://www.linotype.com/webcontent/index.omeco?CURRENTFOLDERID=1663 [accessed 31 January, 2004]*

Lipton, R. J. & Sedgewick, R. (1990), 'NOTECH: Typesetting without Formatting'.

Loney, M. & Festa, P. (2003), 'Opera's browser finds its voice'.
URL: *http://zdnet.com.com/2102-1104_2-5178061.html [accessed 29 March, 2004]*

Meyrowitz, N. & van Dam, A. (1982*a*), 'Interactive editing systems: Part I', *ACM Computing Surveys* **14**(3), 321–352.

Meyrowitz, N. & van Dam, A. (1982*b*), 'Interactive editing systems: Part II', *ACM Computing Surveys* **14**(3), 353–415.

Morkes, J. & Nielsen, J. (1997), 'Concise, scannable, and objective: How to write for the Web'.
URL: *http://www.useit.com/papers/webwriting/writing.html [accessed 29 March, 2004]*

Murata, M. & Hayashi, K. (1992), Formatter hierarchy for structured documents, pp. 77–94.

Myers, B. A. (1998), 'A brief history of human-computer interaction technology', *interactions* **5**(2), 44–54.

Nielsen, J. (1999), 'The Top Ten New Mistakes of Web Design'.
URL: *http://www.useit.com/alertbox/990530.html [accessed 14 March, 2004]*

Nielsen, J. (2000), *Designing Web Usability: The Practice of Simplicity*, New Riders Publishing.

Nielsen, J., Schemenaur, P. & Fox, J. (1998), 'Writing for the Web'.
URL: *http://www.sun.com/980713/webwriting/ [accessed 16 March, 2004]*

Nottingham, M. (2003), 'The Atom Syndication Format 0.3 (PRE-DRAFT)'.
URL: *http://www.atomenabled.org/developers/syndication/atom-format-spec.php [accessed 29 March, 2004]*

Ossanna, J. F. (1976), *Troff User's Manual*.
URL: *http://plan9.bell-labs.com/sys/doc/troff.html [accessed 3 February, 2004]*

Pemberton, S. e. a. (2001), 'XHTML 1.1 - module-based XHTML, W3C recommendation'.
URL: *http://www.w3.org/TR/xhtml11/xhtml11.pdf [accessed 4 October, 2003]*

Petersen, C. (2001), 'Writing for a Web audience'.
URL: *http://www-106.ibm.com/developerworks/usability/library/us-writ/ [accessed 27 March, 2004]*

Quint, V. & Vatton, I. (1986), Grif: an interactive system for structured document manipulation, *in* 'Text Processing and Document Manipulation, Proceedings of the International Conference', Cambridge University Press, pp. 200–213.

Reid, B. K. (1980*a*), A high-level approach to computer document formatting, *in* 'Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages', ACM Press, pp. 24–31.
URL: *http://portal.acm.org/citation.cfm?id=567449 [accessed 6 March, 2004]*

Reid, B. K. (1980*b*), Scribe: A Document Specification Language and its Compiler, PhD thesis, Carnegie-Mellon University.

Reid, B. K. & Hanson, D. (1981), An annotated bibliography of background material on text manipulation, *in* 'Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation', pp. 157–160.

Reid, B. K. & Walker, J. H. (1980), *Scribe User's Manual*, third edn, Unilogic, Ltd.

Rhodes, J. S. (1998), 'How to gain the trust of your users'.
URL: *http://www.webword.com/moving/trust.html [accessed 16 February, 2004]*

Roisin, C. & Vatton, I. (1993), 'Merging logical and physical structures in documents', *Electronic Publishing - Origination, Dissemination, and Design* **6**(4), 327–337.

Saltzer, J. H. (1965), *The Compatible Time-Sharing System, A Programmers Guide*, 2 edn, MIT Press, chapter Manuscript typing and editing: TYPSET, RUNOFF, p. section AH.9.01.

Spring, M. B. (1991), 'Electronic printing and publishing: The document processing revolution'.
URL: *http://www.sis.pitt.edu/~spring/courses/molde_lec_2001.html [accessed 27 January, 2004]*

Technical Committee JTC 1/SC 34, I. (1996), 'ISO/IEC 10179:1996 Document Style Semantics and Specification Language (DSSSL)'.
URL: *[accessed 5 October, 2003]*

TEX Users Group (2000), 'Basic information about TEX'.
URL: *http://www.tug.org/whatis.html [accessed 12 January, 2004]*

van Dam, A. & Rice, D. E. (1971), 'On-line text editing: A survey', *ACM Computing Surveys* **3**(3), 93–114.

van Vleck, T. (1995), 'The IBM 7094 and CTSS'. updated 2003.
URL: *http://www.multicians.org/thvv/7094.html [accessed 20 October, 2003]*

Winer, D. (2003), 'RSS 2.0 specification'.
URL: *http://blogs.law.harvard.edu/tech/rss [accessed 29 March, 2004]*

Zope Project (2001), 'Structured Text Wiki'.
URL: *http://dev.zope.org/Members/jim/StructuredTextWiki/FrontPage [accessed 14 February, 2004]*

APPENDIX A

STRUCTURED ANALYSIS DIAGRAMS

Figure A.1. Context Diagram

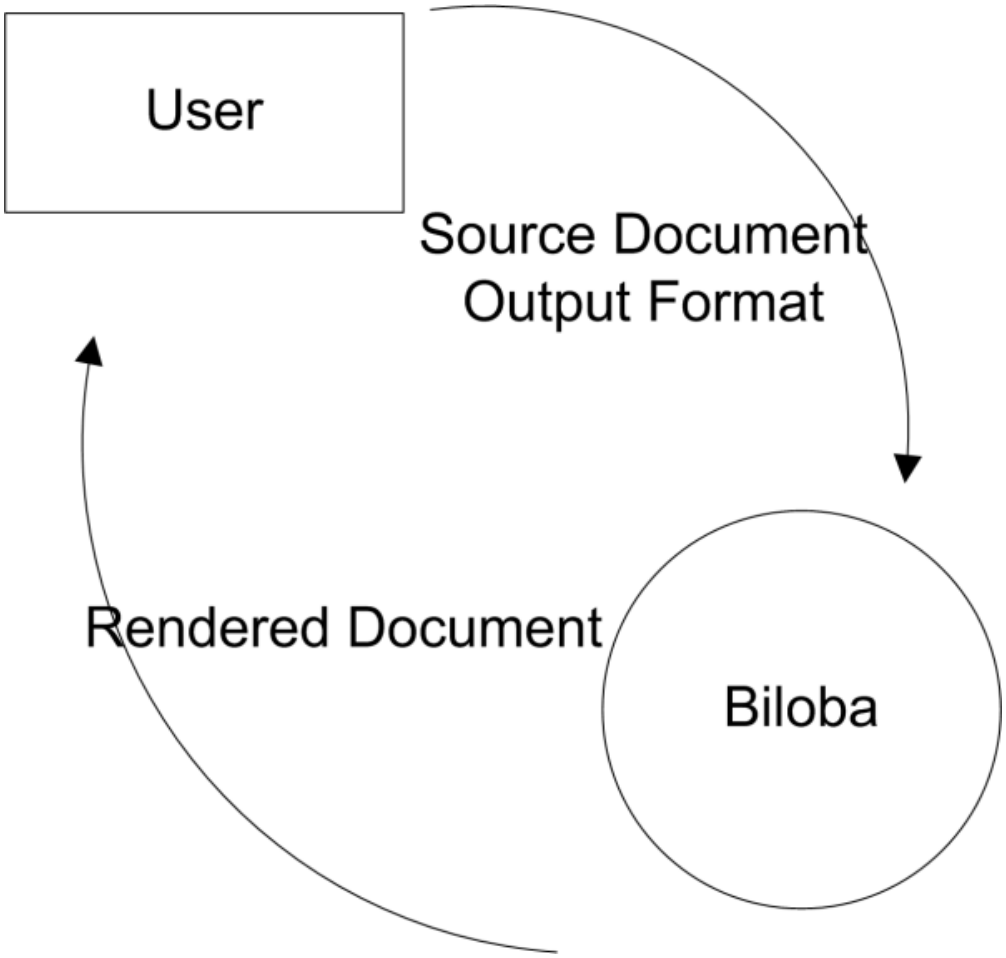


Figure A.2. Diagram 0: Biloba

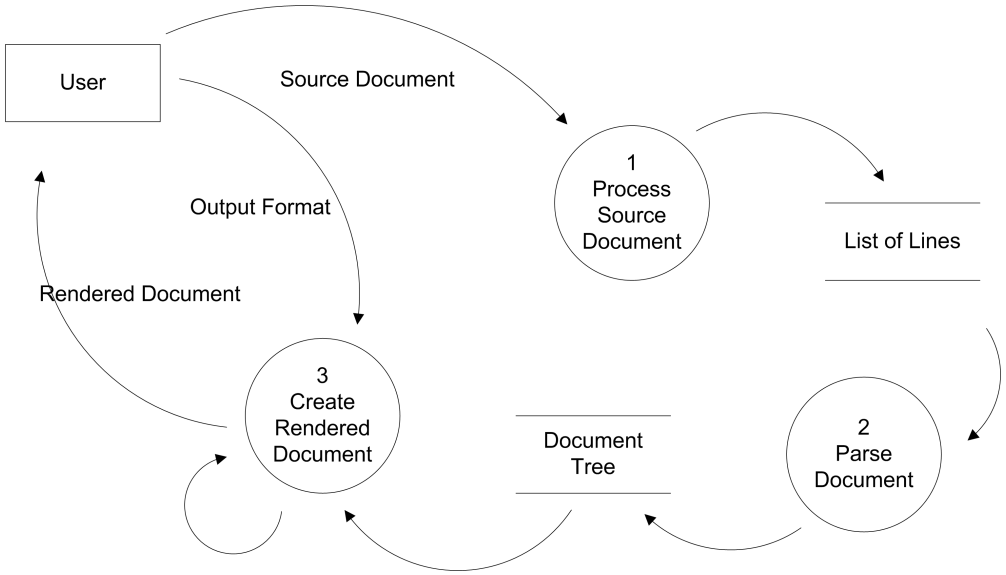
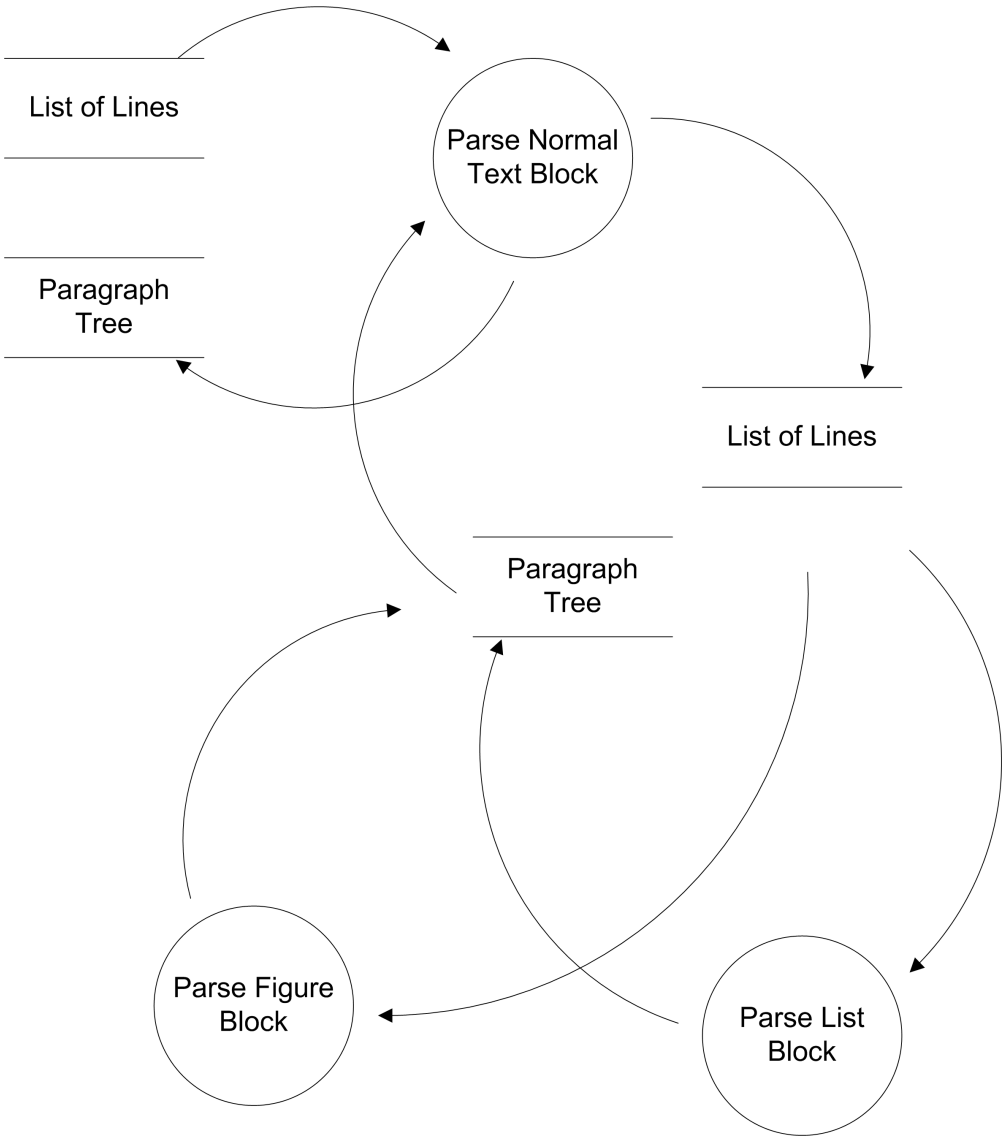


Figure A.3. Detail Diagram 2: Parse Document



APPENDIX B

COMMENTED SOURCE CODE

B.1 Parser Core Files

The commented source code can be found in the directories '`source/`' and '`source/lib/`' on the accompanying CD.

'`biloba.r`' represents the Web interface, '`stxify.r`' represents the command line interface. Both interfaces prepare the incoming data before including the file '`main.r`' that does the actual parsing and inline formatting.

'`lib/parse-block.r`' contains the definition for '`parse-block`' and '`parse-list-block`' which are recursively called for blocks of text and lists, respectively.

'`lib/figure.r`' is called by '`parse-block`' when a figure needs to be parsed. It tries to find a matching figure module located in '`source/modules/`' and executes it or the default module, if none could be found.

'`lib/inline-formatting.r`' is responsible for the inline formatting. It is invoked by '`main.r`'.

Finally, '`lib/escape.r`' provides definitions for the functions that perform the backslash- and HTML- escaping. Invoked by '`main.r`'.

B.2 Output Writer

The commented source code can be found in the file '`source/lib/output-writer.r`' on the accompanying CD.

The output writer is invoked directly from the interfaces ('`biloba.r`' or '`stxify.r`').

B.3 Figure Modules

The commented source code can be found in the directory '`source/modules/`' on the accompanying CD.

B.4 Style Sheets

The style sheets can be found in the directory '`source/style/`' on the accompanying CD.

B.5 Test Harness

The source code for the test harness can be found in the directory '`test_harness/biloba/`' on the accompanying CD.

APPENDIX C

BILOBA STX – USER'S GUIDE

Biloba STX

User's Guide

Version 1.0
7-Apr-2004

Contents

List of Figures

Chapter 1

Preparation

1.1 About This Document

This is the *User's Guide* to the Biloba document formatting system. It takes the reader on a guided tour through the process of creating documents with Biloba. Starting with the creation of a simplistic document, more and more features are explained as the reader progresses.

For details about the setup process, Biloba's internals or how to extend the system, please refer to the *Expert's Guide*.

Biloba is a non-interactive document formatting system specifically designed for documents on the Web. It was developed as a project completed as part of the requirements for the BSc. (Hons) Computer Studies by Viktor C. Pavlu under the supervision of Carlton McDonald at the University of Derby in the years 2003-2004.

1.2 Basic Concepts

Biloba takes a document in plain text format and converts it into the output format of your choice. Currently you can choose from the following formats:

- **XHTML** Extensible Hypertext Markup Language. The standard format for documents on the Internet.
- **TEX** A very popular file format in the scientific and academic communities. Can be converted to PDF (and other) files.
- **XML** Extensible Markup Language. A standard format for structured data.

- **debug, raw** Representation of the document's logical structure.

Most users would want their documents converted to XHTML.

1.3 Test Environment

For the purpose of this guide, we will use the *Biloba STX Test Harness* which can be found on the CD. It is a very simple environment that needs no configuration by the user — it can be started right away but needs to be copied from the CD to your local drive as temporary files have to be created which is not possible on the CD.

To start it, copy the directory '`test_harness`' to your local hard disk and execute '`start.bat`' which will start the Web server required to run Biloba. If the server was successfully started, you will see a window displaying the text

'`Copyright (c) 1991-2003 iMatix Corporation`'.

Do not close this window while using the test harness.

Your browser displays the *Biloba STX Test Harness* page (redirect.html). Go directly to the *Test Environment*, a page where you can enter STX documents for testing purposes. This is the environment used for this guide.

The way documents are formatted does not differ whether you use this simple pre-configured test environment or any other interface. Biloba can be configured to run embedded in your Web server or as a command line tool. The *Expert's Guide* tells you how to setup Biloba for your needs.

Chapter 2

Introduction to Biloba

2.1 Your First Document

Go to the test environment as described in the previous section.

Push the *Render my Document* button and you will see the document from the text area rendered into HTML. Hit your browser's *back* button to return to the text area where you can edit the source of the document. Modify the text and push the *Render my Document* button again. The rendered document has changed accordingly.

Whenever you push the *Render my Document* button, Biloba is invoked to transform your document. It parses the source in the text area and converts it into the format you have selected in the dropdown box left to the button. Have Biloba render your document into another format to get the idea.

2.2 Paragraphs and Captions

Biloba's Structured Text is organised around the concept of text elements. A text element consists of one or more non-empty lines that have the same indentation on each line. Indentation refers to the number of blanks at the beginning of a line.

A structured document consists of sections which may contain sub-sections which again may contain sub-sections and so on. Every section is introduced by a caption giving the name of the section followed by at least one text element that forms the body of the section.

Every text element after a caption that is indented two characters relative to the caption belongs to the body of that caption. Together, a caption and its body

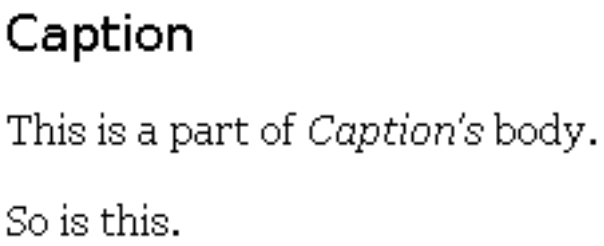
Figure 2.1: Source: A Simple Section

```
Caption

  This is a part
  of /Caption's/ body.

  So is this.
```

Figure 2.2: Rendered: A Simple Section



elements form a section.

Here is an example:

"Caption" is the caption for the section and the two paragraphs "This is..." and "So is this." form the body of the section.

Try the example in the test environment to see how Biloba renders them.

Note that captions are not allowed to span multiple lines.

Sections and sub-sections can be nested:

Here we have a section with two paragraphs followed by three sub-sections each of which contains further paragraphs.

2.3 Term Definitions and Quotes

In the previous section we had an example of three sections that only consisted of a word and its definition. The word was put as the caption of the section and the section's body contained its definition.

As terms and their definition are a very common thing in documents, Biloba has a special form to express that something is such a term/definition pair.

By adding the information that something is a term/definition pair instead of a

Figure 2.3: Source: Nested Sections

```
Language

  Ludwig von Wittgenstein once said "The limits
  of language are the limits of one's world."

  Let's have a closer look at language.

  Syntax

    Syntax describes the structure of
    valid sentences in a language.

  Semantics

    Semantics describe the meaning of
    a syntactically correct sentence.

  Pragmatics

    Pragmatics is the meaning of a
    sentence put into the context of
    background knowledge and experience.
```

Figure 2.4: Rendered: Nested Sections

Language

Ludwig von Wittgenstein once said "The limits of language are the limits of one's world."

Let's have a closer look at language.

Syntax

Syntax describes the structure of valid sentences in a language.

Semantics

Semantics describe the meaning of a syntactically correct sentence.

Pragmatics

Pragmatics is the meaning of a sentence put into the context of background knowledge and experience.

Figure 2.5: Source: Term/Definition Pairs

```
Syntax -- structure of valid sentences in a language.
Semantics -- meaning of a syntactically correct sentence.
Pragmatics -- meaning of a sentence put into the ...
```

Figure 2.6: Rendered: Term/Definition Pairs

Syntax
: structure of valid sentences in a language.

Semantics
: meaning of a syntactically correct sentence.

Pragmatics
: meaning of a sentence put into the ...

common caption followed by a paragraph, the information can be presented in a more succinct way and will therefore be easier to convey. If all term/definition pairs are formatted using this special form, it is also possible to set them off from the rest of the text and thus making it easier for the reader to find a certain definition.

The special form for term/definition pairs looks as follows:

Biloba has special forms for many common text elements in order to give them additional meaning.

Another special form is used for quotes:

Here the left part must be enclosed in quotation marks, otherwise the parser will identify the quote as a very long term that needs to be defined.

Using the special forms we can now re-structure the example from the previous section as follows:

In terms of structure, the result is quite different from the first example. We only have one section instead of three and instead of simple paragraphs we have logically distinct text elements:

Figure 2.7: Source: A Quotation

```
"The limits of language are
the limits of one's world." --Ludwig von Wittgenstein
```

Figure 2.8: Rendered: A Quotation

"The limits of language are the limits of one's world."
– Ludwig von Wittgenstein

Figure 2.9: Source: Nested Sections Restructured Using Special Forms

```
Language

  "The limits of language are
  the limits of one's world." --Ludwig von Wittgenstein

  Let's have a closer look at language.

  Syntax -- structure of valid sentences in a language.
  Semantics -- meaning of a syntactically correct sentence.
  Pragmatics -- meaning of a sentence put into the ...
```

Figure 2.10: Rendered: Nested Sections Restructured Using Special Forms

Language

"The limits of language are the limits of one's world."
– Ludwig von Wittgenstein

Let's have a closer look at language.

Syntax
: structure of valid sentences in a language.

Semantics
: meaning of a syntactically correct sentence.

Pragmatics
: meaning of a sentence put into the ...

Figure 2.11: Source: An Itemized List

```
- item
- item that spans more
  than one line
- item
```

Figure 2.12: Rendered: An Itemized List

- item
- item that spans more than one line
- item

- a quote,
- a paragraph, and
- three term/definition pairs

You should try to use the special forms whenever possible to implicitly add this information to your document which otherwise would be lost. It helps to make your documents more accessible to your audience.

2.4 Lists

There a two kinds of lists in Biloba:

- itemized or bullet lists (such as this one), and
- numbered lists

An item of a bullet list is introduced by a "- " sequence in front of it.

As we can see in the example, list items are allowed to span multiple lines as long as the lines are well aligned with another.

Numbered list items are introduced by a "1) " sequence in front of them.

An empty line ends a list. Consequently, these are two lists, though the author might have intended otherwise:

Figure 2.13: Source: A Numbered List

```
1) item number one
2) item number two
3) and finally, item number three
```

Figure 2.14: Rendered: A Numbered List

1. item number one
2. item number two
3. and finally, item number three

Figure 2.15: Source: Two Lists

```
1) item number one
2) item number two

3) and finally, item number three
```

Figure 2.16: Rendered: Two Lists

1. item number one
2. item number two

1. and finally, item number three

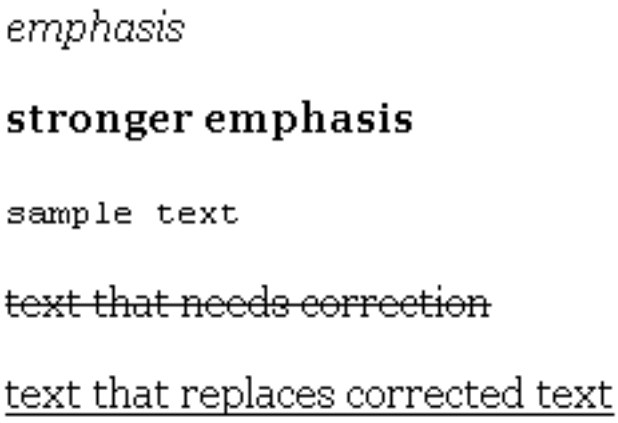
Figure 2.17: Source: Inline Formatting

```
/emphasis/
*stronger emphasis*

''sample text''

~text that needs correction~
_text that replaces corrected text_
```

Figure 2.18: Rendered: Inline Formatting



2.5 Inline Formatting

Inline formatting is sometimes added to *emphasize* words that need to be set off from the rest of the sentence. Do not overuse this feature, otherwise the whole effect of emphasis gets lost in chaos.

This is how it is done in Biloba:

Sample text is used to mark something as an example or to signal that this text has to be typed in or is the output of a program.

Text that needs correction and the accompanying text that replaces corrected text are used to denote that something was faulty and should be corrected accordingly.

Figure 2.19: Source: Hyperlinks

```
[http://www.google.com]
[search google>>http://www.google.com]
```

Figure 2.20: Rendered: Hyperlinks

http://www.google.com search google

Figure 2.21: Source: A Horizontal Rule

```
---
```

Text between square brackets is used to create hyperlinks. The text between is interpreted as the target. Optionally you may add text to a link as illustrated in the last row of the example. The given text will be turned into a link to the given location. If no text is given, the location itself is displayed as link text.

A line that only consist of dashes will be rendered as a horizontal separator.

2.6 Adding Images

Adding images to a document is done with

where the first line represents the caption of the image. The line without the hash sign '`#`' in front gives the path to the image. The '`#type:image`' line is required so that Biloba knows it should render this figure as an image.

Note that these lines must be indented by two spaces relative to the current level. If the line immediately before this indented text element was a single line, it would be converted into a caption and these lines were interpreted as normal text in a paragraph. Therefore it is essential to ensure that text elements immediately before an image consist of at least two lines. If splitting the line to span two lines is not an option you can add a single '`.`' character as the only character to be found in the second line. The '`.`' will not produce any output but ensures that Biloba does not turn the line before into a caption.

Here is an example:

Without the guiding dot, "Consider these leaves" would become a caption for obvious reasons.

Figure 2.22: Rendered: A Horizontal Rule

Figure 2.23: Source: Adding an Image

```
  #Image of Ginkgo Leaves (Tamara Crupi, September 1996)
  #type:image
  ../biloba.jpg
```

Figure 2.24: Rendered: Adding an Image

Fig. 1: Image of Ginkgo Leaves (Tamara Crupi, September 1996).

Figure 2.25: Source: Preventing a Caption

```
Consider these leaves:
.
  #Image of Ginkgo Leaves (Tamara Crupi, September 1996)
  #type:image
  ../biloba.jpg
```

Figure 2.26: Rendered: Preventing a Caption

Consider these leaves:

Fig. 1: Image of Ginkgo Leaves (Tamara Crupi, September 1996).

2.7 Adding Example Text

You can make Biloba retain text just as you entered it and have it copied into a figure. No further formatting is applied to this text element.

This is achieved in the same way as we did with images, except that you must omit the '`#type:image`' line.

The first line not starting with a hash sign '`#`' indicates the beginning of text that should be preserved. It extends until the next line with "normal" indentation.

This is particularly useful when describing external data such as program listings or printouts.

2.8 Further Reading

We have now seen all of Biloba's forms required to create documents. Look at the source of the documents part of the test harness for additional examples. Learning from examples can be very effective.

Figure 2.27: A Simple C Program

```
#include <stdio.h>
#define S "Hello, World\n"

int main(int argc, char **argv)
{
  exit(printf(S) == strlen(S) ? 0 : 1);
}
```

Figure 2.28: Rendered: Adding a Program Listing

```
#include <stdio.h>
#define S "Hello, Worldn"

int main(int argc, char **argv)
{
  exit(printf(S) == strlen(S) ? 0 : 1);
}
```

Fig. 1: A Simple C Program.

If you still have questions you should have a look at the *Expert's Guide* to Biloba which covers features you would not require for everyday use.

APPENDIX D

BILOBA STX – EXPERT'S GUIDE

Biloba STX
Expert's Guide

Version 1.0
17-Apr-2004

Contents

List of Figures

Chapter 1

Introduction

1.1 About This Document

This is the *Expert's Guide* to the Biloba document formatting system. It explains the setup process, program internals and how to write extension modules for the Biloba document formatting system. Formatting commands the average user does not require most of the time are also covered. For a general introduction to document formatting using Biloba, please refer to the *User's Guide*.

Biloba is a non-interactive document formatting system specifically designed for documents on the Web. It was developed as a project completed as part of the requirements for the BSc. (Hons) Computer Studies by Viktor C. Pavlu under the supervision of Carlton McDonald at the University of Derby in the years 2003-2004.

Chapter 2

Setup

2.1 Web Server Installation

This section assists you with setting up Biloba in combination with the Apache HTTP Server. It assumes no prior knowledge of the Apache HTTP Server, however the information on Apache's '`httpd.conf`' configuration file presented here is limited to the minimum required for Biloba. This guide is not intended as replacement for the extensive documentation available on the Apache HTTP Server which is highly recommended before deploying the Web server in a production environment. The documentation is available at http://httpd.apache.org/docs/. Note that it is discouraged to use the Biloba prototype in a production environment, as it can not be guaranteed to be safe. For example, Biloba does not check that the requested documents are located within the DocumentRoot — all files readable to Biloba can be accessed through the HTTP server by clever use of the query string. Making the program source available via the HTTP server is also considered to be harmful due to security concerns! Later releases will fix this.

First you have to download and install the Apache HTTP Server which is available at http://httpd.apache.org/download.cgi. You will also require an interpreter for the REBOL programming language which is available at http://www.rebol.com/platforms.shtml. Installation files for a Microsoft Windows environment can also be found on the accompanying CD in the directory '`w32setup/`'.

Install the HTTP server using the setup program. Then add the following entries to your '`httpd.conf`' located in '`conf/`' inside your Apache installation directory. Search for the '`AddHandler`' directive and add the extension '`.r`' for REBOL files. This will enable '`.r`' files as CGI scripts (see Fig. 2.1).

Then add the following lines to your configuration (see Fig. 2.2) to enable the execution of CGI scripts in the directory '`biloba/`' within your document root. The document root is the directory from which the server obtains the documents

Figure 2.1: Enable .r Files as CGI Scripts

```
AddHandler cgi-script .cgi .r
```

Figure 2.2: Directory Entry That Enables the Execution of CGI Scripts

```
<Directory "YOURDOCROOTHERE/biloba">
  Options ExecCGI
  AllowOverride None

  Order allow,deny
  Allow from all
</Directory>
```

the clients request. By default it is located in the directory '`htdocs/`' within your Apache installation, but can be changed to any directory you like. Replace the text '`YOURDOCROOTHERE`' with your actual path to your document root which is specified by the configuration entry '`DocumentRoot`' followed by the path.

The next step is to setup REBOL. Just copy the file '`rebol031.exe`' to a directory on your local disk. Usually this will be '`C:/rebol/`'.

The next step is to copy the Biloba files to their destination. The directory '`w32setup/biloba/`' on the accompanying CD contains all required files. These must be copied to the directory you specified as DocumentRoot in the '`httpd.conf`' file. The location of the '`biloba.r`' file should be '`YOURDOCROOTHERE/biloba/biloba.r`'.

Edit '`biloba.r`' so that the very first line contains the location of the REBOL interpreter. Figure 2.3 illustrates how this line must look like in order to be interpreted by the Apache HTTP Server.

The setup is now complete — Start the Apache HTTP Server and visit `http://localhost/biloba/biloba.r?test.stx` with your browser!

2.2 Stand-alone Installation

This section explains how to setup Biloba to be used off-line. The only things you require are an interpreter for the REBOL programming language and the Biloba program files. Both are located in '`w32setup/`' on the accompanying CD.

Figure 2.3: Shebang line for biloba.r

```
#!c:/rebol/rebol031.exe -cs
```

Figure 2.4: change-dir Path For stxify.r

```
;change this to the directory where
;%stxify.r is located
change-dir %/C/Program%20Files/biloba/
```

Should you require a REBOL interpreter for a non-Windows platform, you have to download if from http://www.rebol.com/platforms.shtml.

First, copy the REBOL interpreter to a directory of your choice on your local disk. Usually this will be '`C:/rebol/`'.

The next step is to copy the Biloba directory ('`w32setup/biloba/`') to a place on your local disk. Associate '`.r`' files with the REBOL interpreter by double clicking on '`stxify.r`' and selecting the REBOL binary from the list, if the REBOL installation has not already done the association for you.

The Biloba prototype requires you to edit the file '`stxify.r`'. At the beginning of the file you will find a line starting with '`change-dir`' followed by a path introduced with the percent sign. Change the path after the percent sign to the path in which '`stxify.r`' is located. Note that this path *must* use forward slashes, must end with a slash, and must be an absolute path starting at the root. Also note that blanks need to be encoded as '`%20`'. For a typical installation this line is shown in Figure 2.4.

Optionally you can assign the extension '`.stx`' with Biloba's command line interface '`stxify.r`'. To do this double click on '`welcome.stx`' and select '`stxify.r`' to be the default application for '`.stx`' files.

2.3 Using the Command Line Interface

Biloba can be used from the command line to prepare files offline in various formats. Figure 2.5 shows the command line switches that can be used with '`stxify.r`'.

If no output format is specified, you will be asked to enter one. Available output formats are *debug*, *xml*, *html*, and *tex*.

If no source files are specified, you will be asked to enter one.

A file with the same name as the input file but the extension changed to the name of the output format. Note that previous existing files with that name will be overwritten without prior warning!

Figure 2.5: Command Line Switches for stxify.r

```
Usage: stxify.r [options] file...
Options:
    -h            Display this information
    -o <format>   Specify output format of following input files
                  Permissible formats are: debug, xml, html, tex

Examples:
    stxify.r -o xml fileA.stx fileB.stx
```

Chapter 3

Development

3.1 Writing Extension Modules

The figure mechanism in Biloba is designed to be easily extensible. New syntax rules for figures can be added to the system just by adding modules that adhere to the rules outlined in this section.

3.1.1 What are Figure Modules?

Whenever a line in a document is spontaneously indented, that is indented without a prior heading, the line and all following lines with the same level of indentation or more will be extracted from the document and rendered as figure.

The most basic type of figure is a verbatim area. Everything entered will appear exactly as typed in the source. Physical line breaks as well as blanks are preserved — no formatting is applied.

This is the default figure type. Every figure without an explicit type parameter will be treated as verbatim area.

3.1.2 Parameters

If the first lines of a figure are introduced with a hash sign (#), they have a special meaning.

A line with only a hash sign and some text after it will be treated as the figure's title. If the text after the hash sign contains a colon (':'), the line is treated as key/value pair with the key as the left side and the value the right side of the

Figure 3.1: A Sample Figure

```
#This is the title
#type:image
path/to/image.jpg
```

Figure 3.2: Document That Uses the TRANSFORM Figure

```
#Text converted to UPPERCASE
#type:transform
#case:uppercase
This TEXT will be
tRaNsFoRmEd to
UPPERCASE characters,
however useful this
may be ...
```

colon.

One such key, *type*, has a pre-defined meaning: it specifies the figure module that is called to process the parameters and the figure text.

All other key/value pairs have no pre-defined meaning but can be assigned one by developers of a figure module.

Figure 3.1 shows an example of a figure. The figure text will be parsed by the module '`image`' located in the directory '`modules/`'. This is the module that inserts images in your document.

3.1.3 A simple Figure Module

We will now create simple figure module called *TRANSFORM*. It takes the figure text and transforms all characters to lowercase. This behaviour can be influenced with the parameter '`case`', which allows the values 'lowercase', 'uppercase', and 'preserve'. A typical invocation of this figure module can be seen in Figure 3.2.

While this figure module is not particularly useful, it illustrates all the concepts required for creating a figure module.

In order to add a figure module, you have to create a file with the name of the module. In our case, we create a file called *transform* with no extension and put it into the directory '`modules/`' where all figure modules must be located.

Figure modules are incepted as functions in the REBOL programming language. The function has to accept two arguments

Figure 3.3: Source Code for the TRANSFORM Figure Module

```
func [ headers lines /local transformed-string ][
  ;create a string that contains the first line
  transformed-string: copy lines/1

  ;so we can easily append the other lines after
  ;inserting a newline character
  lines: next lines ;skip first, already added, line
  forall lines [
    insert tail transformed-string newline ;insert newline
    insert tail transformed-string lines/1 ;insert the line
  ]

  either find headers "case:uppercase" [
    ;transform to uppercase
    uppercase transformed-string
  ][
    if not find headers "case:preserve" [
      ;transform to lowercase
      lowercase transformed-string
    ]
  ]

  ;return a valid document node [ 'node-type [ "node's content" ] ]
  reduce [ 'verbatim reduce [transformed-string] ]
]
```

- a block containing all headers, and
- a block containing the lines

Figure 3.3 shows the source code of the TRANSFORM figure module. The first line starts the function definition with the first block containing the parameters '`headers`' and '`lines`'. These are the words through which the two blocks will be passed. The '`/local`' string is called a refinement. It specifies that '`transformed-string`' is a local variable.

'`either find headers "case:uppercase"`' tests if '`"case:uppercase"`' was passed as header. If so, the function transforms the text to uppercase characters using the REBOL function '`uppercase`'.

Finally the function has to return a valid document node which consists of a document node identifier and a block of further nodes and strings that make up the node's content. These values are enclosed in a REBOL block.

To create more sophisticated modules, you need to have a background in the REBOL programming language. The REBOL Reference Manual is available online at http://www.rebol.com/users/valurl.html.

Adding an Output Writer

Chapter 4

Expert Formatting Commands

4.1 Escaping

Sometimes you want to prevent Biloba from interpreting characters and have them preserved just as they are. A frequent example is the use of a colon (:) inside a figure caption. Naively writing the colon inside the caption will turn the caption into a key/value pair.

Fortunately there is a mechanism to prevent this. By adding a backslash in front of any character, the character will appear exactly the same way in the output. This can be used to prevent the colon from becoming a key/value pair delimiter. Figure 4.1 shows an example.

Note that this can be applied to any character you feel necessary.

4.2 Empty Paragraph

The empty paragraph, a single period as the only character in a line, can be used to directly influence the current level of indentation without the need for text.

This is especially useful if you want to have a figure immediately following another figure. Without the empty paragraph the second figure would not be detected as a second figure but the will be joined to a single, larger figure. Without the empty

Figure 4.1: Escaping: Use Backslash to Prevent Colon From Becoming a Delimiter

```
#Escaping: Use Backslash to Prevent Colon From Becoming a Delimiter
...
```

Figure 4.2: Two Adjacent Figures

```
  #Image 1
  #type:image
  image1.jpg
.
  #Image 2
  #type:image
  image2.jpg
```

paragraph you would have to add some text between the figures for clarity, which is sometimes not an option.

An example (see Fig. 4.2) will clarify this.

4.3 Comments

Comments are used to prevent Biloba from parsing a single line or a group of lines.

Lines starting with a hash sign (‘`#`’)in the very first column will be ignored.

To ignore a block of lines, enclose the lines between ‘`#=ignore`’ and ‘`#=end`’, both of which must appear on a line of its own and the hash sign needs to be in the very first column.

4.4 Preserving Input

Text between ‘`#=preserve`’ and ‘`#=end`’ will be transferred 1:1 to the output document.

Note that this feature limits the output format independence. Text inside a preserve block can violate rules in the output format. Biloba can no longer ensure that the created markup conforms the the rules of the output format.

Use this only if you know the document will not be transformed into other formats than the one you wrote the preserve blocks for.

APPENDIX E

BILOBA STX – PARSER RULES

Biloba STX

Parser Rules

Version 1.0
7-Apr-2004

Contents

Chapter 1

Introduction

1.1 About this Document

This is a summary of formatting rules for Structured Text that were defined as part of the Final Year Project entitled *Document Formatting Systems* completed as part of the requirements for the BSc. (Hons) Computer Studies by Viktor C. Pavlu in the years 2003-2004. This document is also available on the CD that accompanies the project report.

Biloba implements most of these rules. Future extensions to Biloba should be implemented according to this document to retain a consistent source format and consistent formatting across different versions of the parser and other Structured Text parsers.

The syntax is made to be as intuitive as possible, however intuitive does not mean lax. There is a small set of strict rules that need to be obeyed when creating a structured document. This document describes these rules from a developer's point of view. For a user's point of view, please consult the *User's Guide* and *Expert's Guide*.

1.2 Goal of Rules

The rules must be intuitive, consistent, easy to remember and unambiguous while at the same time explicit markup should be avoided where possible. Users must be able to create a simple document right away without reading a manual.

Chapter 2

Structured Text Parser Rules

2.1 Structural Rules

The basic unit of text in Biloba is the line. A line is a sequence of characters terminated with CRLF (carriage return, line feed; hexadecimal: '`0D 0A`'), CR or LF (depending on operating system used) or terminated by EOF.

A line that consists of only whitespace and the delimiter is an empty or blank line.

Non-empty lines have a certain level of indentation, that is the number of blanks between the start of the line and the first non-whitespace character. Tab characters account for two blanks (this can be configured with the '`tab-size`' variable in %main.r).

An empty line or a change in indentation delimit blocks of text. The various block elements are described below.

2.1.1 Paragraphs

Multiple non-empty lines that share the same indentation level are joined to form one paragraph.

An empty line separates paragraphs.

2.1.2 Captions

A caption is a single line followed by one or more lines that are more indented.

Then the lower level and all following text elements on the same level form the body of the section in the document and the caption line serves as section heading.

Two lines followed by text on a lower level are not captions, rather the text is "spontaneously indented".

2.1.3 Figures

Text that is spontaneously indented, a line of text that starts with two blanks without a caption immediately before, is regarded as a figure.

Figure text is usually retained the way it was typed in and will be rendered using a non-proportional font so that the '`i`' and the '`X`' character have the same width. Whitespace is also preserved. These properties make figure test ideal for code samples or other examples within a technical document.

The layout of a figure depends on the type of the figure. By default no further processing is performed but by adding a figure header of the form '`#mode:image`', the figure text is interpreted by the *image* figure module. The *image* module interprets the figure text as reference to an image and inserts the image as figure.

A programmer can add figure modules to the system by writing a REBOL function with two parameters, *headers* and *lines*. The function processes the figure text contained in *lines* and the optional figure headers contained in *headers* to create a document node that represents the figure in the document tree. This function must be saved in the directory %modules/ under a filename which will be the name of the module. For details see the *image* module in '`modules/image`'.

By adding a figure header of the form '`#An Example`', the figure will be given the caption "An Example". All figures are numbered automatically as well.

In addition to the '`#mode:`' header any header of the general form '`#key:value`' can be added to the top of a figure. These key/value pairs are the way passing parameters to the figure modules (via *headers*).

2.1.4 Term and Definition

Inside a line[1], two dashes '`--`' are used to separate a term and its definition.

If the user wanted a hyphen instead of the terminus/definition pair, three dashes are required.

[1] In the current version of Biloba term/definition pairs are not allowed to span multiple lines.

2.1.5 Quotes

A paragraph enclosed in double quotation marks followed by two dashes and a name is rendered as a quote. The quoted text is the actual quote and the text after the dashes refers to the person that is quoted.

2.1.6 Lists

There are two kinds of lists in Biloba:

- itemized lists (unordered, "bulleted")
- enumerated lists (ordered, "numbered")

Lists are a group of paragraphs introduced with either a bullet or a number indicating the type of list. Every element in a list may span multiple lines and lists can be nested. An empty line delimits a list.

In list items spanning multiple lines, the subsequent lines must be aligned with the text rather than the bullet token.

As users sometimes intuitively indent lists to separate them from the rest of the text without the intention to create a new sub block, this manner is accounted for in Biloba and spontaneously indented lists are treated as if they were not indented.

The result is improved usability in most of the cases where lists are used, however it also introduces ambiguity if a list is the first element after a caption:

Multiple lines of text followed by an indented list are parsed as a paragraph followed by a list on the same level.

A single line of text followed by an indented list is parsed as a line followed by a list on the same level. This is what one would expect. However this clashes with the definition of a caption "...a single line followed by one or more lines that are more indented".

Therefore lists are not allowed to be the very first element after a caption if the caption is followed by an empty line unless the list is spontaneously indented in respect to the level of the sub block introduced by the caption (or double indented in other words).

If there is no blank line between the list and the caption, normal indentation is enough to discern the list in a sub block from a list indented for better readability only. The underlying assumption is that if accentuating the list was the only motivation for indentation, the user also would have added a blank line to bring

the list out more clearly — otherwise the list was indented on purpose yielding a caption and a list in a sub block.

Itemized Lists

Itemized lists are used to group text elements into a concise presentation where the elements do not appear in specific order.

The following tokens can be used to indicate an element of an itemized list:

- ‘`o text`’
- ‘`- text`’
- ‘`* text`’
- ‘`*) text`’

Ordered Lists

Ordered lists are similar to itemized lists but their elements’ order plays a role to the meaning of the text. Therefore the elements are usually numbered.

The following tokens can be used to indicate an element of an ordered list:

- ‘`1. text`’
- ‘`1, text`’
- ‘`1) text`’

Instead of ‘`1`’ any number can be used, however the actual numbering is done automatically by Biloba to allow easy re-ordering of elements.

2.1.7 Horizontal Delimiters

A line that consists of (at least 3) dashes only is regarded as a horizontal delimiter. Either a horizontal rule will be inserted or the text flow continues on the next page or there is a reasonable pause before the rest of the text is to be read out, depending entirely on the output media.

2.2 Inline Rules

All elements discussed so far were "structural" elements. They started a new block or represented a part of an existing block.

Inline formatting is done within the structural elements. Simple symbols inside the text are used to indicate the desired type of the text enclosed by the symbols. As the format of Biloba is declarative throughout, the actual rendered output depends on the output writer, however these symbols are commonly used in newsgroups to apply a certain type of emphasis to words:

strong text enclosed in asterisks will be typeset to bring it out stronger than other text

keyboard text enclosed in a pair of two single quotes is displayed as if typed in by the user

emphasized text enclosed in slashes will be emphasized

deleted text enclosed in tilde characters is used to denote something has been deleted

underlined text enclosed in underscore characters will be rendered underlined

2.2.1 Linking to other Documents

Hyperlinks to other documents are written as '`[label>>target]`' where label is the text that will be displayed and target is the resource that can be reached when following this link.

2.3 Processing Instructions

This section describes processing instructions to the parser. This is the only form of explicit markup in Biloba and will not be required for most uses.

2.3.1 Comments

Lines with a hash sign '`#`' as their very first character will be ignored by the parser. This can be used to add instructions targeted to your editor for example.

Text between the lines '`#ignore`' and '`#end`' is ignored entirely.

2.3.2 Pass-through

Text between the lines '`#preserve`' and '`#end`' is verbatim copied to the output writer encapsulated in a *preserve* node. No further processing is applied to these sections.

Pass-through sections can be used to directly manipulate the physical output of a document. For example it is perfectly valid to add TEX commands to typeset mathematical formulae in such a section. However if the document is then to be rendered in a format other than TEX, the TEX commands will not be interpreted but rather appear in the output as they were typed. Therefore note must be taken that this sacrifices the content/representation independence to a certain amount, nevertheless it is sometimes a powerful feature.

2.3.3 Pseudo-Paragraph

A line that consists of a single period '.' only will produce no output. It is solely there to circumvent ambiguities in indentation, for example after a list.

2.3.4 Escaping

The characters and their influences on the parser were outline in this document. Sometimes, however it is desired to have a hash sign '`#`' as the first character in a line without having the line ignored by the parser, or to have square brackets '`[]`' in a paragraph and not having them replaced with a hyperlink.

Adding a backslash character '\' in front of any character preserves the character as it is. The escaped character is not interpreted as formatting symbol of any kind. The backslash character itself can be added to a document by writing '\\'.

2.4 Rules Not Implemented in the Prototype

All rules this far were implemented and tested in the prototype of Biloba. The following rules act as a starting point for further work.

2.4.1 Footnotes

Adding footnotes to inline tokens while typing is done by appending '`^(footnote-text)`' to the word where the footnote should be added. The text between the parenthe-

ses will be displayed where the output writer deems it appropriate, usually on the bottom of the current page.

2.4.2 Abbreviations

The first time an abbreviation appears in the text it should be written out in parentheses. If the parser finds an abbreviation followed by text enclosed in double parentheses, the abbreviation and the spelled-out form from within the parentheses will be added to an internal synonym database.

Every time the abbreviation is used, the spelled-out form can be added by the output writer automatically, if desired.

The synonym database is intended to be created on the fly for each document alone, but it is also possible to have one central synonym database multiple authors are sharing.

2.4.3 Special Symbols

Special characters normally not available on a standard keyboard could be added via escape sequences. Mathematical operators, greek letters and Umlaute, ... are examples of this type of text.

2.4.4 Tables

Tables are not part of the Biloba rules but should be implemented by means of figure modules instead.

2.4.5 Referencing

Adding references to inline tokens while typing is done by appending '`^[ref-id]`' to the word where the reference should be added. *Ref-id* is an unique identifier of the source being referenced which Biloba looks up in a BibTEX database. Referenced entries will automatically be added to the bibliography at the end of the document.

APPENDIX F

PROJECT SUPPORTING TASKS

F.1 Regression Tests

In order to continuously verify the parser still behaves as expected as new features are added, the author wrote unit tests for all features as they were added. Every unit test represents a document element that has to be detected and transformed into its internal representation by the parser.

The unit tests consist of a file with the input to the parser and a file containing the expected parse tree. A REBOL script compares all created parse trees with their corresponding expected output and reports where differences occur. If adding a parse rule for paragraphs breaks all tests for lists, the regression tests give a clear hint where the programmer has to search for errors. Figures F.1 and F.2 show output from the test environment where tests failed (F.1) and where all tests were successfully passed (F.2).

The test environment can be found in the directory '`test_harness/biloba/tests/`' on the accompanying CD.

For more information about unit tests, please visit http://www.xprogramming.com/ or refer to *Kent Beck's Original Testing Framework Paper* available at http://www.xprogramming.com/testfram.htm.

F.2 Estimation And Time Management

All tasks performed during the course of this project were recorded using a simple tool that logged start- and stop times as well as task descriptions. This data was used to continually assess the time spent on individual tasks and taskgroups in order to adhere to the project plan.

Further it is used to increase the accuracy of future estimations based on the additional data. This was also done for this project's estimation, based on data gathered in earlier projects the author participated.

Figure F.1. Test Tool Sample Screen with Failed Tests

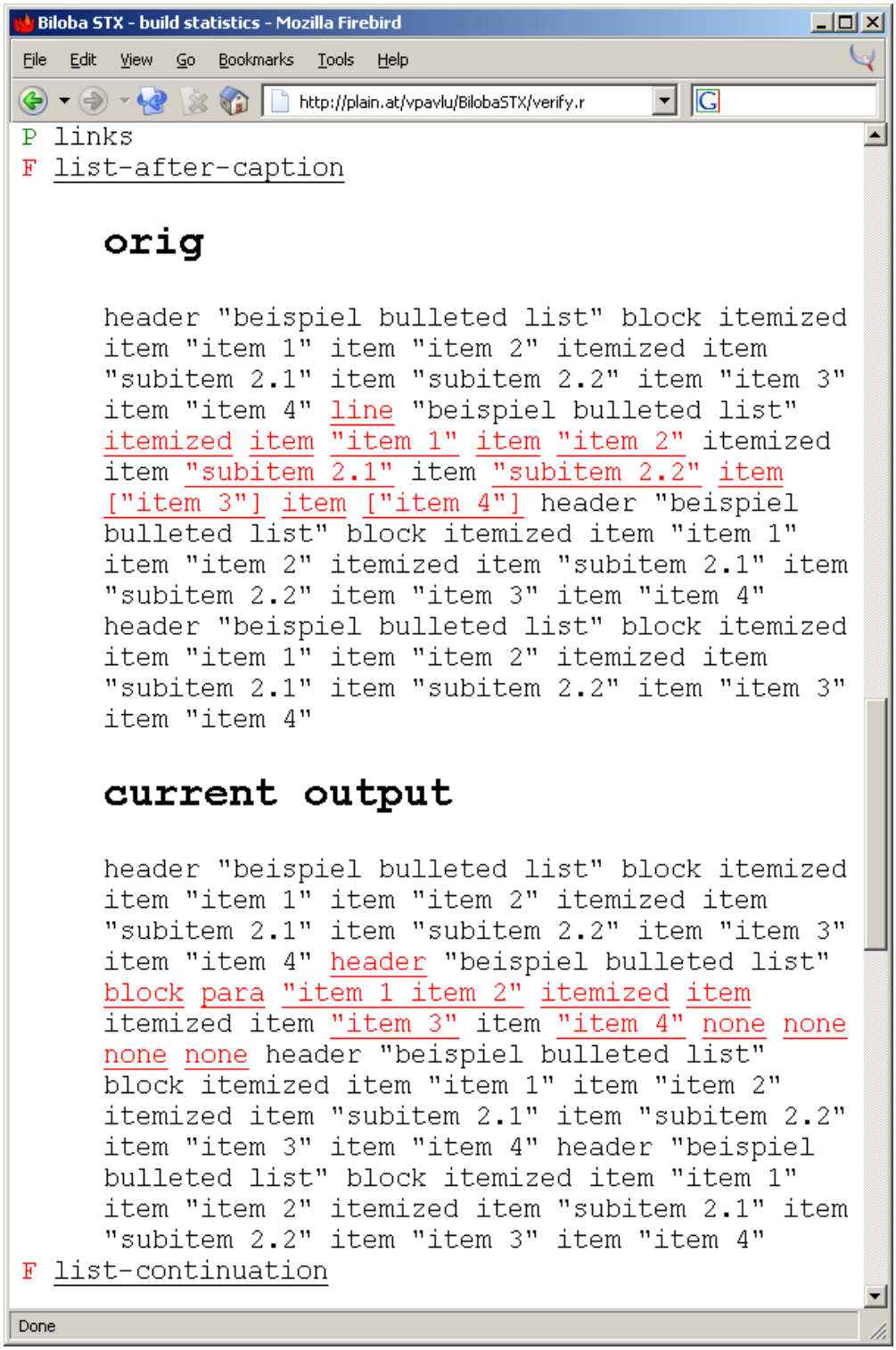


Figure F.2. Test Tool Sample Screen with all Tests Passed

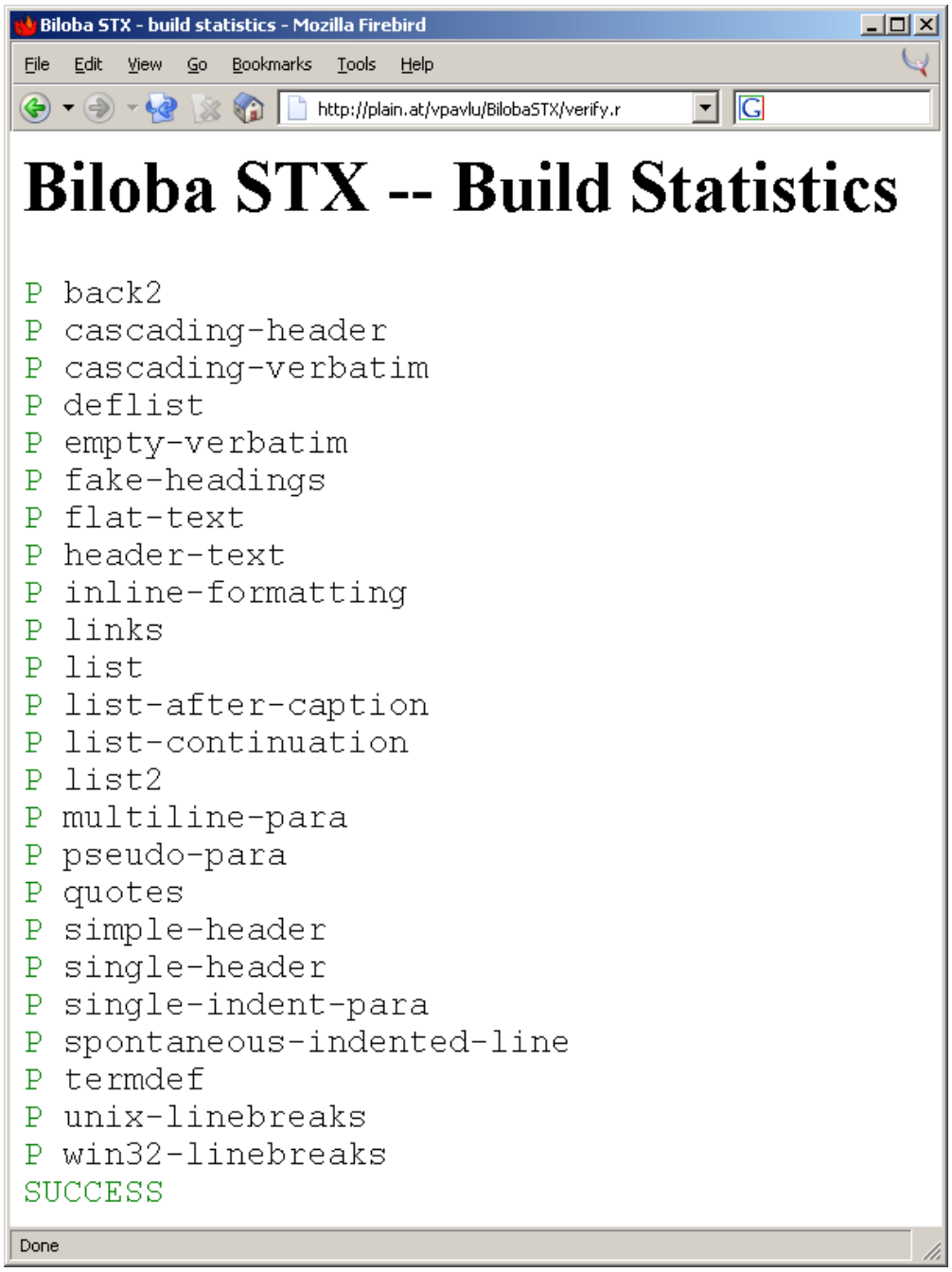


Figure F.3. Sample Data as recorded by the Logging Tool

```
<entry>
  <start>28-Sep-2003/16:22:53+1:00</start>
  <duration>0:00:19</duration>
  <phase>Project accompanying tasks</phase>
  <task>test logger</task>
</entry>
```

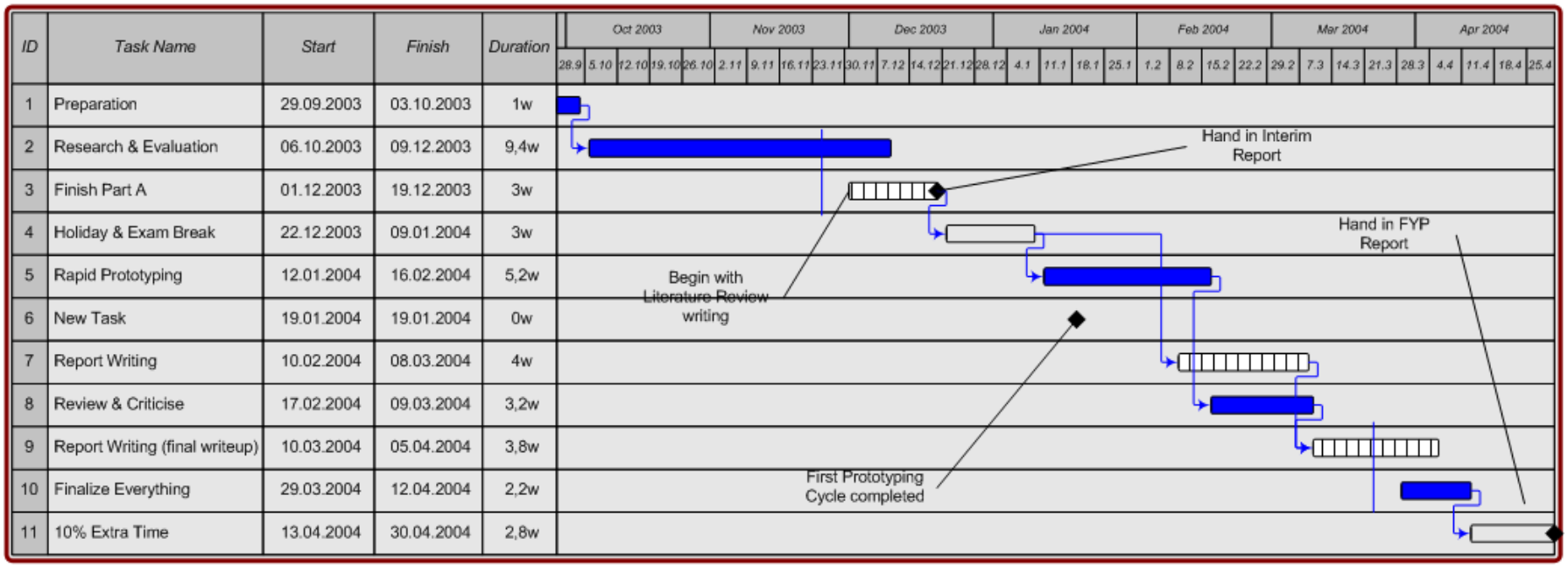
Figure F.3 shows sample data gathered by the recorder.

APPENDIX G

PROJECT OVERVIEW PLANS

Final Year Project
Viktor Pavlu, 2003/2004
entitled

TEXT FORMATTING SYSTEMS



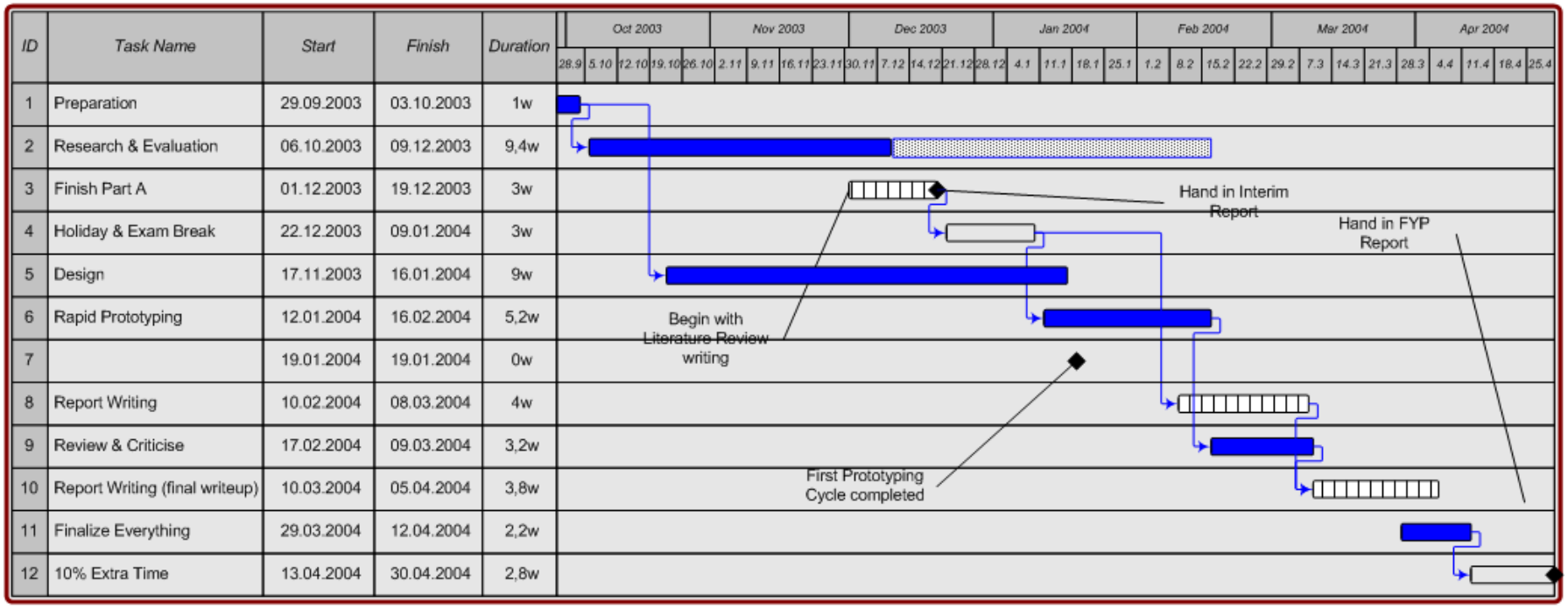
ID	Task Name	Start	Finish	Duration
1	Preparation	29.09.2003	03.10.2003	1w
2	Research & Evaluation	06.10.2003	09.12.2003	9,4w
3	Finish Part A	01.12.2003	19.12.2003	3w
4	Holiday & Exam Break	22.12.2003	09.01.2004	3w
5	Rapid Prototyping	12.01.2004	16.02.2004	5,2w
6	New Task	19.01.2004	19.01.2004	0w
7	Report Writing	10.02.2004	08.03.2004	4w
8	Review & Criticise	17.02.2004	09.03.2004	3,2w
9	Report Writing (final writeup)	10.03.2004	05.04.2004	3,8w
10	Finalize Everything	29.03.2004	12.04.2004	2,2w
11	10% Extra Time	13.04.2004	30.04.2004	2,8w

Project Overview Gantt, Version 1.0 (1-Oct-2003)

Figure 1: Original Project Gantt-Diagram

Final Year Project
Viktor Pavlu, 2003/2004
entitled

DOCUMENT FORMATTING SYSTEMS



ID	Task Name	Start	Finish	Duration
1	Preparation	29.09.2003	03.10.2003	1w
2	Research & Evaluation	06.10.2003	09.12.2003	9,4w
3	Finish Part A	01.12.2003	19.12.2003	3w
4	Holiday & Exam Break	22.12.2003	09.01.2004	3w
5	Design	17.11.2003	16.01.2004	9w
6	Rapid Prototyping	12.01.2004	16.02.2004	5,2w
7		19.01.2004	19.01.2004	0w
8	Report Writing	10.02.2004	08.03.2004	4w
9	Review & Criticise	17.02.2004	09.03.2004	3,2w
10	Report Writing (final writeup)	10.03.2004	05.04.2004	3,8w
11	Finalize Everything	29.03.2004	12.04.2004	2,2w
12	10% Extra Time	13.04.2004	30.04.2004	2,8w

Project Overview Gantt, Version 2.1 (2-Dec-2003)

Figure 2: Updated Project Gantt-Diagram

APPENDIX H

PROJECT PROPOSAL

School of Computing & Technology
University of Derby

Computing Scheme - Final Year Project Proposal

Name: Viktor PAVLU
Project Title: Document Formatting Systems

Main Supervisor: Carlton McDonald
Second Supervisor: Dr. Zaigham Mahmood

Does this project meet the BCS Accreditation requirements?
Please include any constraints.

Aims and Objectives of Project:

- **Conduct background research on the topic of Document Formatting (DF)**
- **Give overview of existing systems and their historical development**
- **Evaluate ways of implementing a prototypical DF System for the Web**
- **Clearly formulate the possible value of a DF System for the Web**
- **Implement a prototypical DF system**
- **Critically review the prototype**

Expected Outcomes or Deliverables:

- **Project Report**
- **Prototypical DF toolkit**

Methodology:

- **Literature review of published papers on this subject**
- **Inspect existing systems (especially Tex, LaTex and Zope STX)**
- **Study techniques of implementing a web-based DF system and choose one for the prototype (data format, e.g. markup; environment, e.g. Apache Module, standalone or scripted)**
- **Project development work to implement prototype**

Hardware and Software Requirements (these MUST be available before the project starts):

Signature: **Date:**

Supervisor's Signature: **Date:**
(If you sign here, you are AGREEING to manage this student's project)

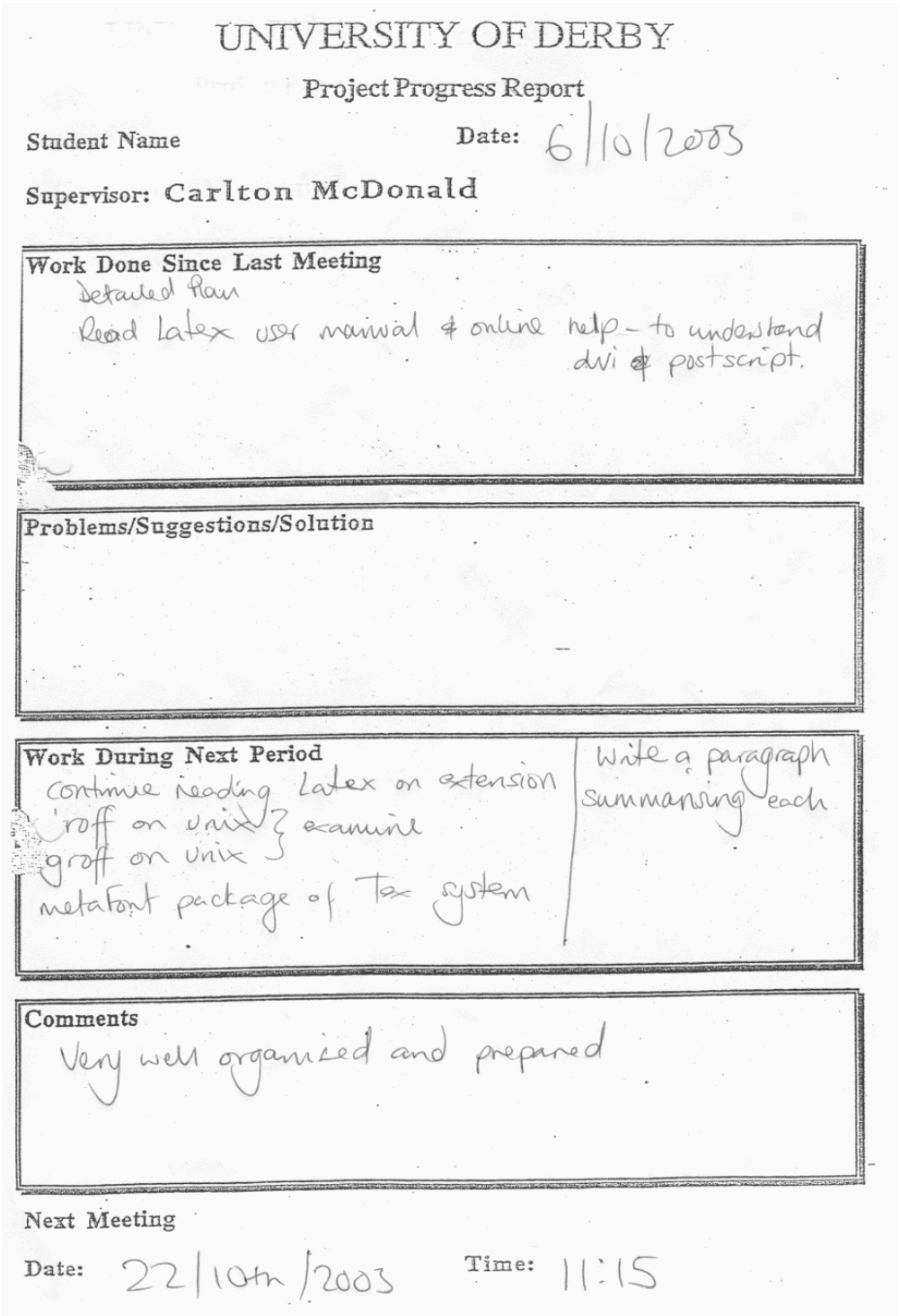
APPENDIX I

PROGRESS REPORTS

The following progress reports were given out by the supervisor at the regular project meetings. They summarize what was done by the student, problems that were encountered and how they could possibly be solved. Also, they provide space for the supervisor to comment on the student's performance.

The progress reports prepared by the student as agenda for the meetings can be found on the accompanying CD in the directory '`docs/progress_small/`'.

Figure I.1. Progress Report: October 10, 2003



UNIVERSITY OF DERBY

Project Progress Report

Student Name Date: 6/10/2003

Supervisor: Carlton McDonald

Work Done Since Last Meeting

Detailed Plan

Read latex user manual & online help - to understand dvi & postscript.

Problems/Suggestions/Solution

Work During Next Period

Continue reading Latex on extension

'roff on unix } examine

groff on unix

metafont package of Tex system

Write a paragraph summarising each

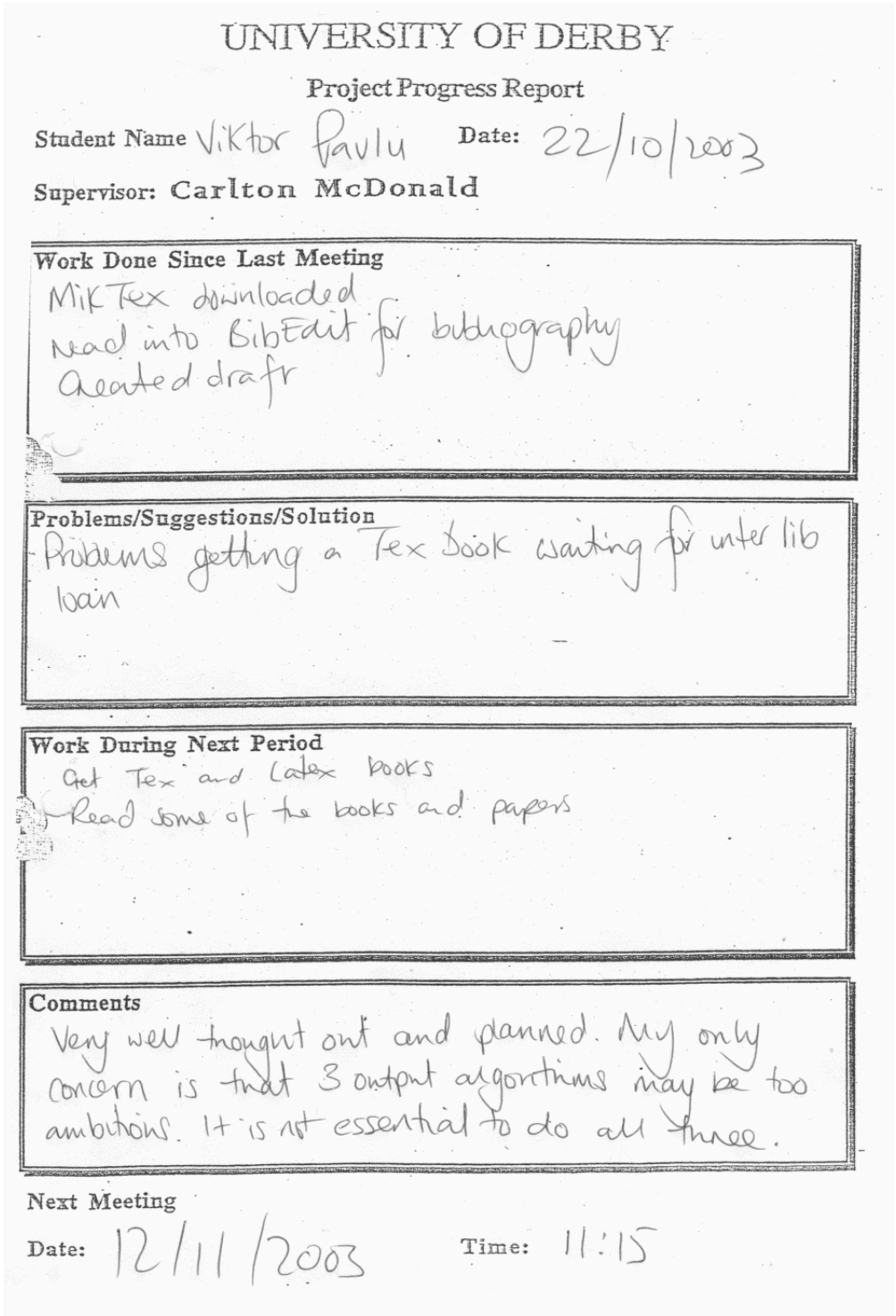
Comments

Very well organized and prepared

Next Meeting

Date: 22/10th/2003 Time: 11:15

Figure I.2. Progress Report: October 22, 2003



UNIVERSITY OF DERBY

Project Progress Report

Student Name Viktor Pavlu Date: 22/10/2003

Supervisor: Carlton McDonald

Work Done Since Last Meeting

MiKTex downloaded
Read into BibEdit for bibliography
Created draft

Problems/Suggestions/Solution

Problems getting a Tex book waiting for inter lib loan

Work During Next Period

Get Tex and Latex books
Read some of the books and papers

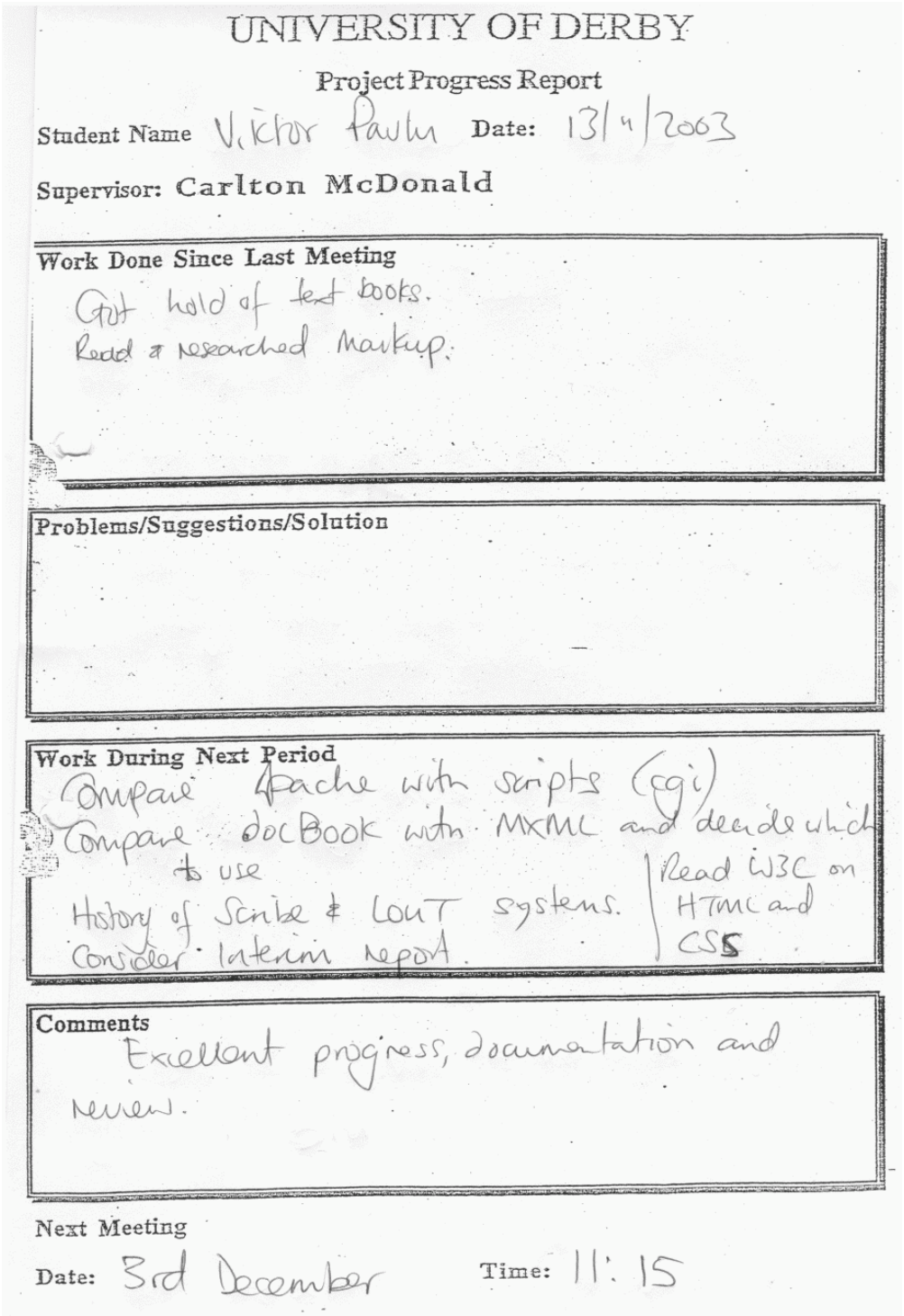
Comments

Very well thought out and planned. My only concern is that 3 output algorithms may be too ambitious. It is not essential to do all three.

Next Meeting

Date: 12/11/2003 Time: 11:15

Figure I.3. Progress Report: November 13, 2003



UNIVERSITY OF DERBY

Project Progress Report

Student Name Victor Pavlu Date: 13/11/2003

Supervisor: Carlton McDonald

Work Done Since Last Meeting

Got hold of ~~text~~ books.
Read & researched Markup.

Problems/Suggestions/Solution

Work During Next Period

Compare Apache with scripts (cgi)
Compare docBook with MXML and decide which to use
History of Scribe & LOUT systems.
Consider Interim report.
Read W3C on HTML and CSS

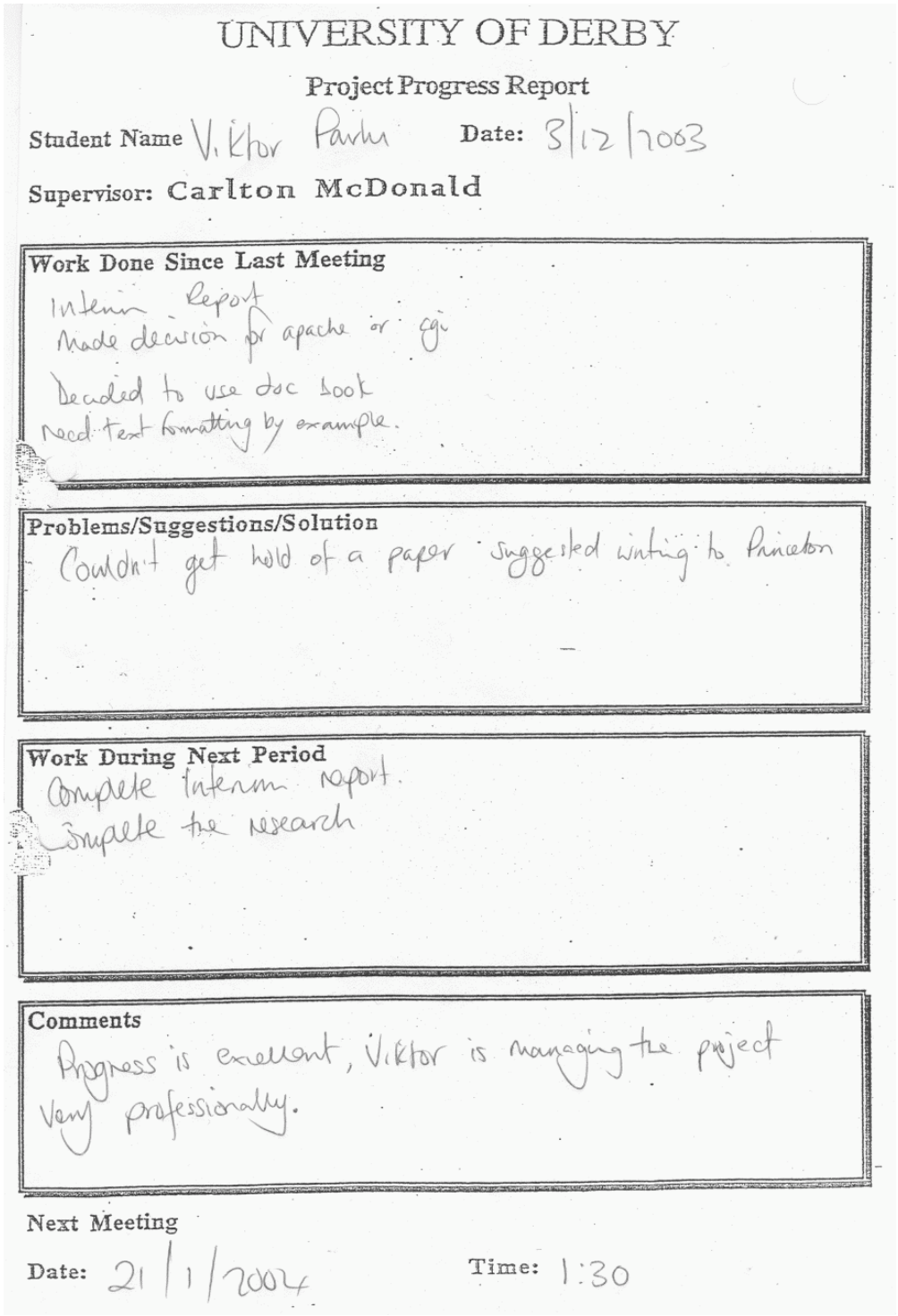
Comments

Excellent progress, documentation and review.

Next Meeting

Date: 3rd December Time: 11:15

Figure I.4. Progress Report: December 3, 2003



UNIVERSITY OF DERBY

Project Progress Report

Student Name Viktor Parka Date: 3/12/2003

Supervisor: Carlton McDonald

Work Done Since Last Meeting

Interim Report
Made decision for apache or cgi
Decided to use doc book
Need text formatting by example.

Problems/Suggestions/Solution

Couldn't get hold of a paper suggested writing to Princeton

Work During Next Period

Complete Interim report.
Complete the research

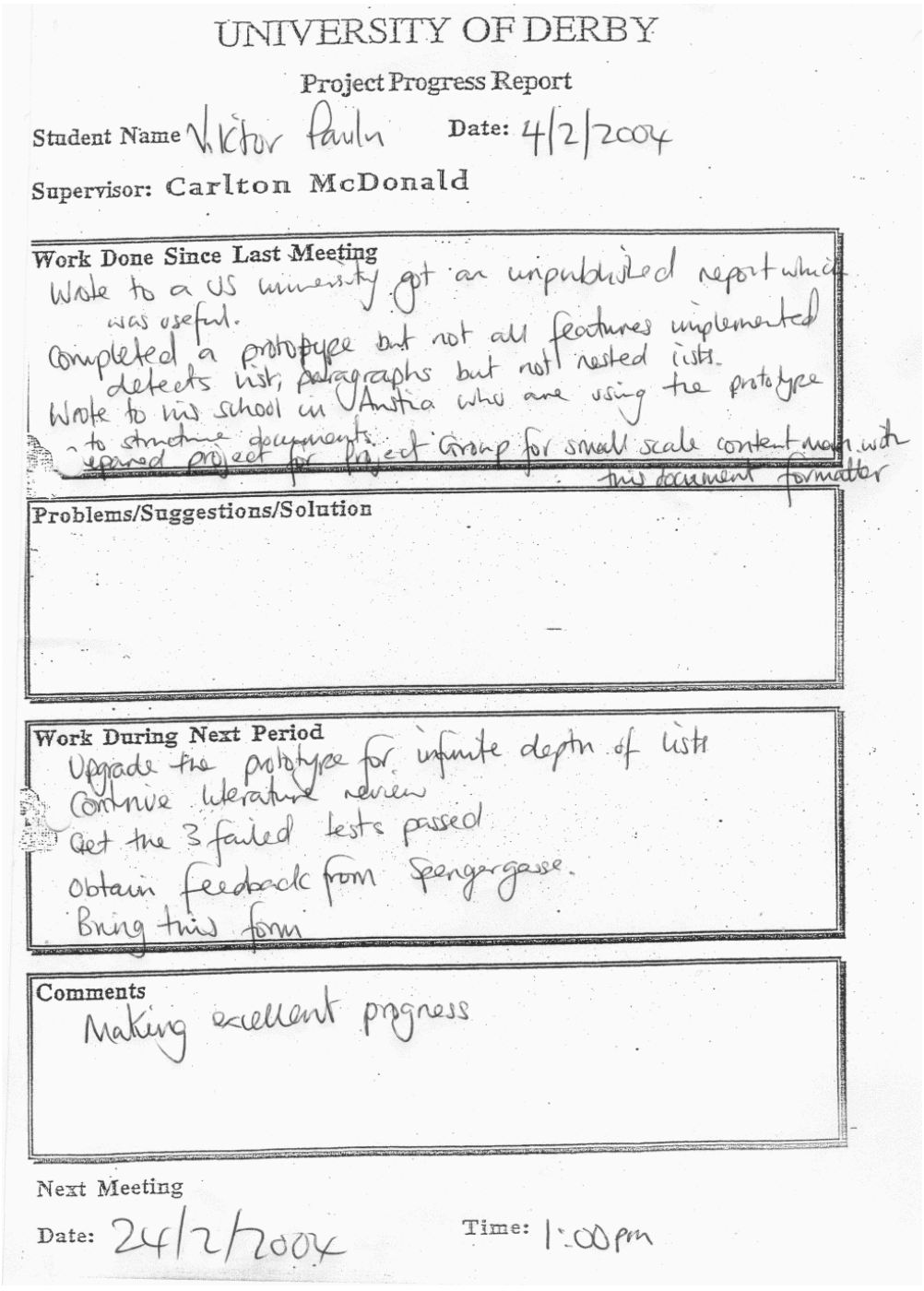
Comments

Progress is excellent, Viktor is managing the project very professionally.

Next Meeting

Date: 21/1/2004 Time: 1:30

Figure I.5. Progress Report: February 4, 2004



UNIVERSITY OF DERBY

Project Progress Report

Student Name Victor Paulu Date: 4/2/2004

Supervisor: Carlton McDonald

Work Done Since Last Meeting

Wrote to a US university got an unpublished report which was useful.
Completed a prototype but not all features implemented detects list, paragraphs but not nested lists.
Wrote to his school in Austria who are using the prototype to structure documents.
Joined project Group for small scale content work with this document formatter

Problems/Suggestions/Solution

Work During Next Period

Upgrade the prototype for infinite depth of lists
Continue literature review
Get the 3 failed tests passed
Obtain feedback from Spengergasse.
Bring this form

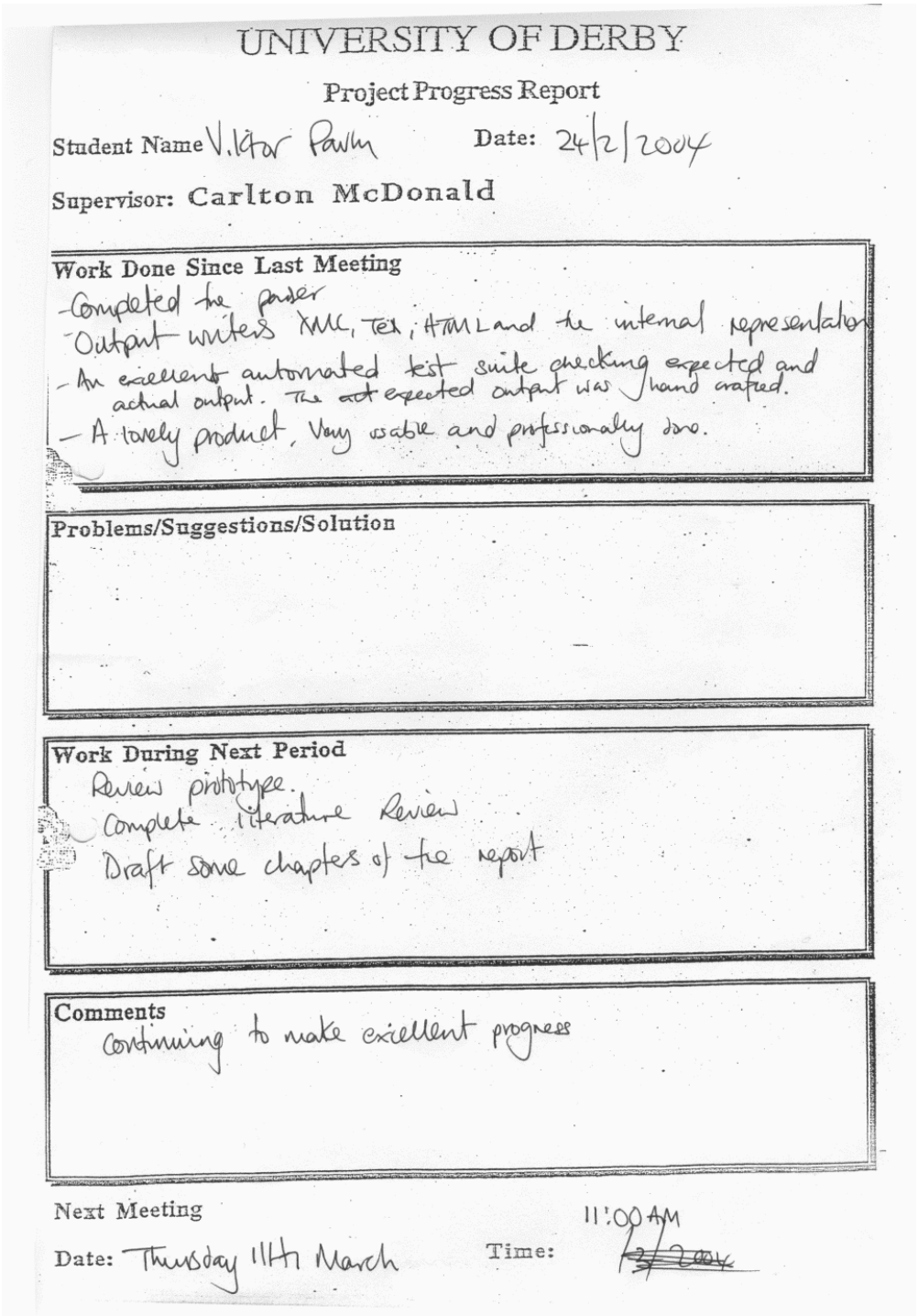
Comments

Making excellent progress

Next Meeting

Date: 24/2/200x Time: 1:00pm

Figure I.6. Progress Report: February 24, 2004



UNIVERSITY OF DERBY

Project Progress Report

Student Name Victor Pavlu Date: 24/2/2004

Supervisor: Carlton McDonald

Work Done Since Last Meeting

- Completed the parser
- Output writers XML, TEX, HTML and the internal representation
- An excellent automated test suite checking expected and actual output. The expected output was hand crafted.
- A lovely product. Very usable and professionally done.

Problems/Suggestions/Solution

Work During Next Period

- Review prototype.
- Complete literature Review
- Draft some chapters of the report

Comments

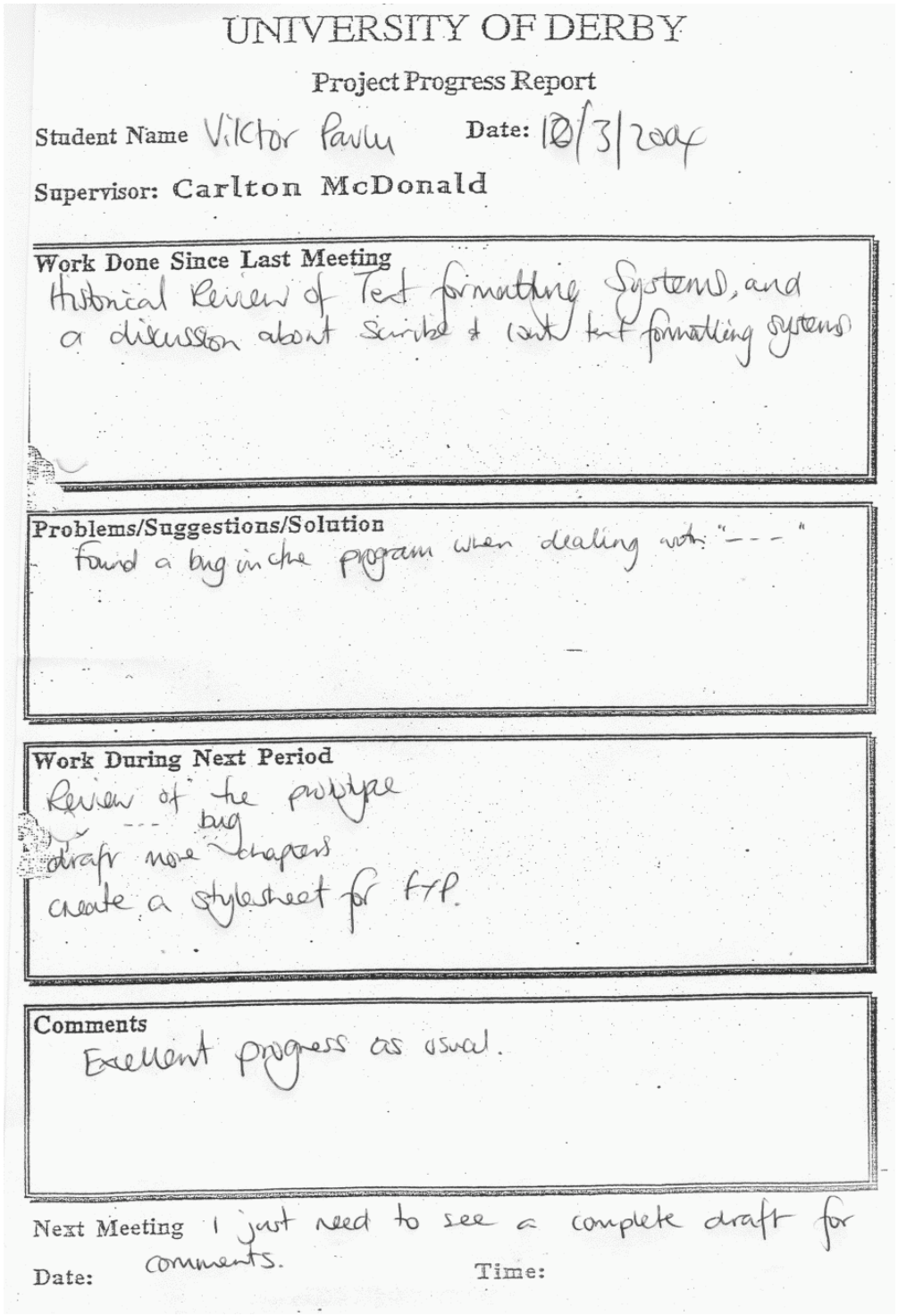
Continuing to make excellent progress

Next Meeting

Date: Thursday 11th March Time: 11:00 AM

~~3/2004~~

Figure I.7. Progress Report: March 10, 2004



UNIVERSITY OF DERBY

Project Progress Report

Student Name Viktor Pavlu Date: 10/3/2004

Supervisor: Carlton McDonald

Work Done Since Last Meeting

Historical Review of Text formatting Systems, and a discussion about Scribe & LaTeX text formatting systems

Problems/Suggestions/Solution

found a bug in the program when dealing with "---"

Work During Next Period

Review of the prototype

--- bug

draft more chapters

create a stylesheet for FYP.

Comments

Excellent progress as usual.

Next Meeting Date: I just need to see a complete draft for comments. Time:

APPENDIX J

INTERIM REPORT

The interim report was handed in on December 3, 2003. The following version is the report without the appendices. The appendices were removed to avoid duplication as the appendices of the interim report are available as appendices of the final report as well.

UNIVERSITY OF DERBY
Derbyshire Business School

Final Year Project Interim Report
as part of the
requirements for the

BSc (Hons) Computer Studies

entitled

Document Formatting Systems

by

Viktor C. Pavlu

in the years 2003 – 2004

Supervising Tutor: Carlton McDonald

Abstract

The Interim Report is part of the Final Year Project. It is intended to give you an introduction to the project and an overview of the progress being made so far. You will also find an updated version of the project proposal and a project overview diagram in the appendices. Approximately 1000 words.

Contents

List of Figures

1 Introduction

My project is to get knowledge of existing document formatting (DF) systems, their history and the techniques they use, in order to be able to implement a prototypical DF system for the web that combines the advantages of existing systems. The purpose of this DF system is the creation of technical documentation with a focus towards online publishing. The DF system prototype does not deal with the actual typesetting, creation of fonts, handling of multiple users trying to concurrently edit a document, versioning of documents or security concerns.

This report summarizes the background of the Final Year Project "Document Formatting Systems" (Sect. 2), describes its current status (Sect. 3), changes to the original proposal (Sect. 4) and future perspectives (Sect. 5) of the project in brief detail.

2 Project Background

Document formatting is the process of mapping information to layout. Since its beginnings in 1961, the use of computerized document formatting systems has steadily increased. Today it is one of the most widespread applications of computers.

There are two approaches towards formatting a document: the first and today the more popular one (as it seems) is to use a graphical editor to apply physical formatting to a document. The second approach is to manually insert logical formatting instructions into a document.

While the latter way seems obsolete and tedious, it has many advantages over a WYSIWYG application: By using generic or logical formatting ('format this as a level 1 heading") instead of using physical formatting ("make this bold and 14pt in size"), a logical structure is given to the document. Changing the appearance of all headings is very easy in a generic coded document, whereas in a document with physical formatting, inconsistent layout could result.

This project gives an overview of the historical development of important document formatting systems, studies and reviews the most important approaches to document formatting, points out the problems they pose and discusses current publications that try to find solutions, with a focus on non-interactive document formatting systems.

For *classic* publishing there exists a wide range of sophisticated DF systems (troff, SCRIBE, TEX, lout), for online publishing, however, there are no such well-established systems. Creating the markup by hand seems the only promising way to create consistent, standards-compliant and irreducible XTHML.

Biloba STX, a small-scale document formatting system prototype, is going to be developed in order to gain first-hand experience of the practical aspects of implementing and deploying a web-based document formatting system that supports generic coding.

The outcomes of this project are a thorough overview of DF systems and their historical development, and a DF system prototype developed with a focus on online publishing that tries to infer the appropriate formatting from the spacing and contents of the plain text source. The output format is generic coded markup for in-browser display or further processing (XTHML, DocBook).

There will also be recommendations for further development of the DF prototype.

3 Objectives and Achievements

To get a better understanding of the problems involved in implementing a DF system I had to research not only on papers published in this field of research but I also had to inspect DF systems currently in use.

In my original project proposal I planned to study the algorithms and data structures used in DF systems by reviewing the source code, where available. This turned out to be inappropriate as those systems are mainly concerned with typesetting and graphical positioning of objects on a page. These are tasks the DF system for the web leaves to the browser. The proposal was changed accordingly.

I did research into the field of well-established file formats used in combination with DF systems and compared them towards their suitability as output format for the DF prototype. Formats under consideration were PostScript, PDF, HTML, M-XML, XHTML, DocBook, DVI and a author-defined XML dialect. I decided to use the W3C recommendations for XHTML in combination with Cascading Style-sheets for displaying and the DocBook standard as output format for further processing.

The report summarizes the history of DF systems starting with runoff in 1961 up to current papers about what their authors think to be the future of DF systems.

A first draft of parsing rules the DF system is going to incorporate was created and the development environment to be used for the prototype was evaluated. The decision was made in favor of the REBOL programming language in combination with CGI and against an Apache 2.0 module written in C++ for reasons involving compatibility with other webservers, built-in support of parsing functions, and speed of prototyping cycle.

During the research — which is not completed yet — I found out that there is at least one DF system that follows the same ideas that I want to imple-

ment. “NOTECH: Typesetting without Formatting”, a paper published in 1990 at Princeton University, describes the ideas of inferring appropriate formatting from spacing and contents of the document. Unfortunately I was not able to get a copy of the paper (currently I am waiting for a reply from the University of Princeton). Another System, Zope STX, developed as part of the Zope project follows this approach, too. Both systems are going to play a major role in the remaining part of the research phase and during the critical review of the prototype as well.

4 Changes to Original Proposal

The title of the project had to be changed from “Text Formatting Systems” to the more appropriate title “Document Formatting Systems” as the systems being observed and the prototype being implemented are not only concerned with the layout of text, but all elements that constitute to a document (tables, figures, mathematical formulae, images, . . .)

Algorithms and data formats used in existing DF systems play a minor role in the development of the DF prototype as these systems try to meet different needs. The proposal was changed to reflect this. See section 3 for details.

Both the original and the updated project proposal can be found in the appendices.

5 Further Plan

Up to now the project plan was followed accurately and everthing was completed in time. However, I am afraid that this is going to change in the next few weeks when the literature review was planned to be completed. This will most likely take more time than originally assessed as I underestimated the number of DF systems involved in the historical development of today’s important DF systems. I also underestimated the size and range of influential papers published in this field.

Therefore I will most likely have to continue work during the winter holidays in order to keep up with the planned progress. This is not going to delay the completion of the project as the holidays were not part of the original estimation and I have arranged an extra buffer of three weeks at the very end of the project to handle such difficulties.

An overview project plan can be found in the appendix.

References

Adobe Systems, I., ed. (1999), *PostScript Language Reference*, third edn, Addison-Wesley.

Barron, D. W. (1989), 'Why use SGML?', *Electronic Publishing - Origination, Dissemination, and Design* **2**(1), 3–24.

Bienz, T. & Cohn, R. (1993), *Portable Document Format Reference Manual*, Addison-Wesley.

Bos, B., Lie, H. W., Lilley, C. & Jacobs, I. (1998), 'CSS layer 2'.
URL: *http://www.w3.org/TR/1998/REC-CSS2-19980512* *[accessed 4 October, 2003]*

Bray, T., Paoli, J., Sperberg-McQueen, C. M. & Maler, E. (2000), 'Extensible markup language (XML) 1.0'.
URL: *http://www.w3.org/TR/2000/REC-xml-20001006.pdf* *[accessed 4 October, 2003]*

CTSS, the Compatible Time-Sharing System (n.d.). From Wikipedia, the free encyclopedia.
URL: *http://en.wikipedia.org/wiki/CTSS* *[accessed 20 October, 2003]*

Furuta, R. K. (1992), 'Important papers in the history of document preparation systems: basic sources', *Electronic Publishing – Origination, Dissemination, and Design* **5**(1), 19–44.
URL: *citeseer.nj.nec.com/furuta92important.html* *[accessed 19 October, 2003]*

Jacobsen, D. (1996), 'The BibTEX format'.
URL: *http://www.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html* *[accessed 19 October, 2003]*

Kingston, J. H. (n.d.), 'The future of document formatting'.

Knuth, D. E. (1984), *The TEXbook*, Addison-Wesley.

Lamport, L. (1986), *LATEX: A Document Preparation System*, Addison-Wesley.

Minimal XML 1.0 (2000).
URL: *http://www.docuverse.com/smldev/minxmlspec.html* *[accessed 8 October, 2003]*

Myers, B. A. (1991), Text formatting by demonstration, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 251–256.

Patashnik, O. (1988), BibTEXing. Documentation for general BibTEX users, version 0.99b.

Pemberton, S. e. a. (2001), 'XHTML 1.1 - module-based XHTML'.
URL: *http://www.w3.org/TR/xhtml11/xhtml11.pdf [accessed 4 October, 2003]*

Saltzer, J. H. (1964), *TYPSET and RUNOFF, Memorandum editor and type-out commands.*
URL: *http://web.mit.edu/Saltzer/www/publications/AH.9.01.html [accessed 20 October, 2003]*

Saltzer, J. H. (1966), *Manusrcipt typing and editing.*
URL: *http://web.mit.edu/Saltzer/www/publications/AH.9.01.html [accessed 20 October, 2003]*

Tobin, G. (1994), Metafont for beginners, third draft.
URL: *http://www.ntg.nl/doc/tobin/mf4begin.pdf [accessed 20 October, 2003]*

van Fleck, T. (1995), 'The IBM 7094 and CTSS'. updated 2003.
URL: *http://www.multicians.org/thvv/7094.html [accessed 20 October, 2003]*

Walsh, N. & Muellner, L. (2002), *DocBook: The Definitive Guide*, 2 edn, O'Reilly & Associates, Inc.