A Formatter-Independent Structured Document Preparation System

G. de V. Smit

Research Report
CS-87-40

July 1987

A Formatter-Independent Structured Document Preparation System

by

G. de V. Smit

Department of Computer Science
University of Waterloo

Waterloo, Ontario, 1987

Abstract

This dissertation investigates the issues associated with making high quality document processing systems available to casual users. A prototype document preparation system is designed and implemented, which provides casual yet demanding users with simple, easy-to-use, interactive access to high quality (batch) document formatting systems. The system provides an interactive, syntax-controlled, structured what-you-see-is-almost-what-you get editing environment for the creation of formatter-independent documents. These documents can easily be transformed by the system into the appropriate input format for almost any target batch formatter.

A practical document model that describes the components of documents and their interrelationships is defined. A document class description language is defined with which the syntax (structure) and semantics (appearance) of classes of documents can be described. Such class descriptions are used by the system to aid the user in creating syntactically correct documents. Mechanisms are developed for dealing with incomplete documents in a syntax-controlled environment.

The dissertation further presents a user interface suitable for editing both the structure and the content of documents. An important aspect of the user interface is its ability to enforce correct document syntax, while still allowing reasonable and non-restrictive editing operations.

Acknowledgements

It is with deep gratitude that I acknowledge the direct and indirect contributions the following people made to this dissertation.

I owe much to my supervisor, Prof. Don Cowan. His encouragement, support, advice and guidance were invaluable. Without his tireless readings and constructive criticisms of my pitiful writings, there would have been no dissertation.

My external examiner, Prof. Alan Shaw from the University of Washington, and the other members of my thesis committee, Professors Randy Goebel, John North, and Frank Tompa deserve thanks for their constructive criticisms that contributed to the improvement of this dissertation.

Eric Mackie and John Wilson made major contributions in shaping my ideas through many (and sometimes heated) discussions. John must be credited with suggesting the idea of using finite automata as driving engines for the system. I also benefited much from discussions with Rob Black, Carl Durance and Martin Leistner.

To Pieter Kritzinger, who started it all and encouraged me along the way: "Baie dankie".

I thank the Computer Systems Group, the Waterloo Foundation for the Advancement of Computing, and Tek Logic (Pty) Ltd. for their very generous financial support. In particular I want to thank Eric Mackie, Prof. Wes Graham, Prof. Don Cowan, Llew Jones and Adriaan Joubert for their confidence in me.

The Computer Systems Group and its "members" provided me with an excellent work environment.

To my wife, Jeannine, thank you for your love, friendship and support, and the many sacrifices you have made through my years of study.

Finally, to my Creator, who formed me and sustains me, to Him be the glory.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, as more and more people have gained access to computer systems, there has been an increase in the number of people demanding access to systems for the creation of high quality documents. Many of these users have little or no computer training and might be termed casual users.

The objective of this dissertation is to investigate the issues associated with making high quality document processing systems available to the casual user. A prototype system is designed and implemented, which provides the user with the following features:

- an ability to view the document as it is being prepared,
- a method of constraining the document to ensure that it is "correct", that is, it conforms to some standard document format,
- a system with retargetable output (documents created on the system may be directed to different batch formatters and to different interactive displays), and
- a method of editing documents even though the editing process may make the documents temporarily incorrect.

In order to construct such a document preparation system the following topics are examined:

- a practical model for documents which describes the syntax (or structure) of correct documents and their associated appearances (or semantics) in terms of interactive display and target batch formatters,
- a language to implement the document model,
- a user interface consisting of a syntax-controlled editor which allows the user to create documents described by the model, and
- a mechanism for handling incomplete and incorrect documents which temporarily modifies the existing document model.

1.1 Motivation

As more people make use of computerised document preparation, a new type of computer user can be identified. These new users are not computer experts. In fact they do not have a computing background, are not familiar with programming languages or other esoterica of computing, and are often only casual users of the computer. They have a problem, namely to produce a neatly formatted document; they know the computer can help them in solving the problem, and they want the solution with a minimum of (perceived) fuss and bother. The terms in which they present the problem to the computer and in which they manipulate the results should be easy for a human to use and understand and should be familiar, that is, based on previous document experience.

The fact that these users might be novice or casual users does not necessarily say anything about their document processing needs. Most of the time they require access to sophisticated document preparation systems that are capable of producing high quality formatted documents. Many powerful document preparation tools exist that are capable of providing for their needs. However, these systems are often awkward and even difficult to use, especially for the novice and casual user. A substantial amount of training is often necessary before one can use these document systems intelligently, with the result that casual users can either not afford the time to get acquainted with the system, or if they do in fact manage to learn how to use it, they often forget what they have learned by the time they want to use the system again.

Related to the widespread use of computers for document preparation, is the fact that documents have become more "mobile". Documents are no longer created and remain in a single fixed form; to the contrary, copies of a document are often made in different forms. The form may differ depending on the output medium, or the intended recipient. For example, the same document may be formatted in different ways depending on whether it is viewed on a terminal, printed on a line printer, or reproduced on a typesetter. Similarly, a report that has initially been formatted according to internal company standards, might have to be reproduced in a different form to conform to a customer's standard.

Not only is it necessary to be able to change the form of documents, but it also is necessary to reproduce them on different output devices, often using different computers. We call such documents portable documents.

The need for portable documents resulted in the emergence of structured formatters [Furut82]. In these formatters documents are regarded as consisting of logical objects (sentences, paragraphs, sections, etc.), and the purpose of the formatter is to allow the manipulation of these objects. Generally speaking, the clearer the structure of a document is defined, the easier it is to reproduce the document in a different form. However, the rules for constructing a correct document[1] may become more complicated, with the result that the user has to know more in order to create a correct document.

It has been argued [Allen81] that batch formatting systems will always be able to do a better formatting job than interactive what-you-see-is-what-you-get (WYSIWYG) systems. We agree with this argument, but at the same time realise that the interactive approach provides a far better work environment, especially for non-expert users of a system.

This dissertation examines the issues associated with providing high quality document

[1]Throughout this dissertation we will use the term "correct document" to refer to a document that adheres to the structure (or syntax) that has been defined for that particular kind of document.

processing capabilities to the casual user in a changing environment. While this investigation is geared towards the casual and novice user, it is certainly the intention that expert users alike would find any system that is developed, practical and efficient to use. The major factors that influence the investigation are:

- Ease of document creation and editing,
- Flexibility in choosing the final appearance (layout) of the document, and
- Flexibility in choosing the type and quality of the output.

Each of these topics is discussed in more detail in the following sections.

1.1.1 Ease of Document Creation and Editing

As mentioned before, many sophisticated and powerful batch-oriented document formatters exist today, such as TEX [Knuth84], Scribe [Reid80], TROFF [Ossan76], GML-based formatters [IBM84b, Water86], and SCRIPT [IBM85, Water85]. These formatters are, however, often not straightforward and easy to use, especially for the novice or casual user, because:

- The formatting language is often complicated and a user unfamiliar with programming concepts may have trouble producing a quality document, or understanding some of the more advanced features of the language.
- The commands and/or declarations to the formatter tend to interfere with reading the document content, and the content in turn interferes with reading the commands/declarations.
- The editing systems used to prepare input for the formatters are usually general purpose editors and are not designed for the specific task of editing text documents.

For these reasons many authors of papers, articles and short documents who might wish to use these batch-oriented formatters, are reluctant to do so.

It is our aim to overcome these drawbacks, not by producing yet another batch formatter, but by providing a uniform, user-friendly and helpful environment in which to create documents that can be formatted by existing and future batch formatters. The environment should not just be easy to use, but the user must feel confident that the document being created in fact conforms to the requirements of the target batch formatter. The system must therefore ensure the creation of correct documents.

An important aspect of convenient document editing, is immediate feedback. Batch-oriented document processors can be very powerful and can afford to put a large amount of effort into producing high quality output. They can for example use global information to "optimise" the appearance of the final document [Knuth81]. Their major drawback, however, results exactly from this batch-orientation: the time lapse between entering a document description and seeing the result of that description is often too long. The user might want to experiment with various formats and layouts, but the delay in turnaround inhibits this and reduces productivity:

> The ability to experiment with different formats is clearly invaluable to both author and transcriber, providing that there are no serious restrictions resulting from this facility. Having to "program" formatting effects is a mental burden and requires sophisticated, complicated code all too often; debugging a sequence of formatting codes is difficult unless a formatted copy of the same document exists for comparison. [Meyro82]

Because of the necessity of embedding commands in the document description, it is much easier to make a mistake (and the elapsed time between making the mistake and discovering it is much longer) in a batch-oriented system than in an interactive one.

Few, if any, editors have sufficient knowledge of the formatting language of the formatter, to be able to tell the user immediately when a syntax error has been made. Even if such an editor would exist, it would not be able to detect semantic errors unless it had an intimate knowledge of the formatting process, in which case it might as well do the formatting itself.

Furthermore, in an interactive document processor it is possible to provide online, context sensitive help that can guide the inexperienced user to use the system effectively. In many of the current batch-oriented systems, error messages that are generated are often cryptic and far away from the actual error; hence they are often not of much help other than to indicate that some error has occurred.

1.1.2 Flexible Layout

As we have pointed out earlier, the use of computers for document preparation resulted in requirements for more flexibility in producing the same document in different forms. Obviously, the fewer changes that have to be made to a document description in order to have it formatted in a different style, the better.

The so-called declarative approach to document specification provides exactly this feature. In formatting languages such as Scribe [Reid80] and GML [IBM84b, Water86] only the logical components of the document (such as sections, headings, paragraphs, etc.) are specified in the document description. These components are then uniformly formatted according to a separate layout file or style sheet.

This approach greatly reduces the amount of necessary formatting detail that the user has to supply. It provides more structure and portability to the document and more things can be done with fewer commands. On the other hand, the user has less direct control over the final appearance of the document. A discerning user might also find that the components defined in a layout may not always meet the necessary requirements, and it is often not easy to define new components, or change the style of existing ones.

Our objective is to provide a declarative system in which the user specifies the logical components of the document, and the system provides the necessary formatting information. However, it must be possible for the more experienced user to define new classes of components and/or change the definition of existing components with relative ease.

1.1.3 Flexible Output

There are two aspects to output quality: the quality of the formatting process, and the quality of the reproduction produced by the output device.

We have already commented on the ability of batch formatters to produce well formatted documents. Some formatters are better at some tasks than others. It is therefore to be expected that users would want to use different target formatters at different times. Our system must be able to support different target formatters, but relieve the user from having to know the details of the target formatting language. In other words, once a document has been created using the system, the system must be able to generate the input for the target formatter with very little user intervention.

Some output devices are more expensive to use than others (e.g. a typesetting machine vs. a laser printer vs. a simple dot-matrix printer). It is not surprising therefore, that the choice of output device will differ depending on the intended use of the document. An early draft of a letter does not necessarily belong on a typesetting machine, just as the final copy of a report or paper does not belong on a dot-matrix printer.

A major advantage of current batch-oriented document processors is that they usually support a large variety of output devices. One can therefore use the same processor to produce documents for a number of diverse output devices. If, on top of this, one has a choice between a number of target formatters, the possibilities become even more attractive.

1.2 Outline

As we have already indicated, a number of issues are addressed in this dissertation, including

- The definition of a practical document model that describes the components of a document and their relationships to each other,
- the creation of an appropriate language with which to describe both the structure and appearance of classes of documents, and

- the design of a suitable user interface for editing the structure and content of documents. An important aspect of the user interface is its ability to enforce correct documents without being so restrictive that the user finds this enforcement of correct structure frustrating. In other words, incorrect documents can be produced during the editing and creation process, but the system ensures that these documents are correct before being translated to a specific target formatting language.

A document model is outlined and motivated in the next chapter, which also provides a framework within which our objectives are to be obtained. Chapter 3 defines and evaluates a language for the description of document classes. Chapter 4 provides some theoretical background relevant to the implementation of a prototype system which is described in Chapter 5. The dissertation is concluded in Chapter 6 by summarising the contributions that were made as well as pointing out some further areas of investigation. The remainder of this chapter describes related work by other researchers.

1.3 Related Work

Document processing systems and their history have been surveyed extensively in the literature [Reid80, Brade81, Furut82, Meyro82, Beach85]. We will refrain from doing so yet again, but will briefly mention those systems that have influenced our work.

1.3.1 Batch Formatters

Many batch formatters have been developed through the years, from RUNOFF [Saltz65] to TeX [Knuth84] and one of its macro packages LaTeX [Lampo86].

Among these systems Scribe [Reid80] and GML [IBM84b, Water86] are significant in that they were among the first to separate document content from document form or style explicitly. TeX is probably the most powerful formatter/typesetter that exists today that is generally available and affordable. However, it requires a significant amount of learning/training on the user's part for anything but the most trivial tasks. LaTeX, a macro package for TEX, is to some degree based on the Scribe model. For the most part it takes a declarative approach and provides the user with a number of both high and low level declarations. While it is considerably easier to use than TEX, there is still a lot of detail — detail that a casual user can easily forget.

1.3.2 Interactive Systems

A plethora of interactive word processors/document editors are available that range from the simple to the complex. See for example [Wiswe85] for a review of eighteen of the more popular ones available for personal computers. Most of them are to some or other degree WYSIWYG, but they rarely make use of document structure. They do not have the concept of document syntax, and very seldom perform document syntax checking.

1.3.3 Syntax-directed Programming Language Editors

Syntax-directed editors (also called language-oriented editors) and their generators such as the ALOE system [Medin82] and the Cornell Program Synthesizer [Teite81] and Synthesizer Generator [Reps84] grew out of the need for program editors that would facilitate the creation of syntactically correct programs. Other recent syntax-directed editors are described in [Campb84, Gansn83, Morri81].

The techniques used in these systems are to a large degree applicable to document editors, and often their authors mention that document editors have been or might be constructed, but do not go into any detail. This is probably because the systems mentioned are geared towards programming languages which are generally speaking more explicitly structured than documents with the result that the editors may seem to be too rigid for use with documents. Furthermore, the incremental parsing (or recogniser) approach taken by many syntax-directed editors is not easy to adapt for use with documents, as documents often have ambiguous grammars. For example, if a section in a document consists of an introductory paragraph followed by a number of subsections and possibly followed by a closing paragraph, it is often not possible to determine whether the last paragraph belongs to the subsection or the section.

Most syntax-directed editors have to be generated and compiled for a specific language environment. In other words, for text documents, one has to create a different editor for every document class or type such as business letters, technical document or thesis. Our system enables the user to edit different classes by merely issuing a command. This is achieved by interpreting document class descriptions, rather than having them pre-compiled and built into the system.

DOSE [Feile83, Feile86] is a multi-purpose display-oriented structure editor that makes use of techniques very similar to ours. It is not specifically geared towards document editing, but attempts to be an editor for anything for which an appropriate structural definition can be written. Unfortunately, because it is to become a commercial product of Siemens Corporate Research and Support Inc. in the near future, it was not possible to obtain enough information about the system in order to make a detailed comparison between DOSE and our system.

1.3.4 Structured Document Editors

Recently a number of structured document editors have been proposed and/or developed. These include Etude [Hamme81] and its derivative the Interleaf Publishing System (IPS) [Morri85], PEN [Allen81], W [King86], Grif [Quint86], and systems by Kimura [Kimur84a, Kimur84b], van Huu [vanHu85], Coray et al. [Coray86], and Furuta [Furut86]. In this discussion we will concentrate on the more recent ones.

All of the systems mentioned in the previous paragraph view documents as structured objects, and rely to various degrees on some kind of description of a document type or class to control the editing and formatting process.

IPS views the document as a simple sequence of components, where each component is assigned a fixed set of properties whose values may be set by the user (one of the properties is for example the object's name). The components do not form a hierarchy – they are a linearisation of the document components found in Etude. The other systems mostly use context-free grammars (usually a modified BNF notation) to describe document structure. In Grif the structure definition resembles a Pascal record declaration, while van Huu proposes using SGML in his system. Coray et al. use a BNF notation (but use Kleene closure rather than recursion to indicate repetition), and Kimura provides additional operators such as constructing sets (or bags), links and shared objects, all specified in a C-like language.

Some of the systems include formatting information in the class description in the form of attributes or properties of the objects (IPS, Coray et al.), while others (Kimura) provide a programming language with which to describe the formatting. Most of the authors were vague about the exact nature of the formatting information, merely making statements such as "...presentation schema ... contains presentation rules for each type of element defined in the structure schemas" [Quint86]. It is also not always clear how much of the formatting can be specified by the user, nor how difficult this would be.

Both IPS and Grif are WYSIWYG systems running on high powered workstations with high resolution bitmap screens. The other systems take a what-you-see-is-almost-what-you-get approach, with some (Kimura, Coray et al., W) providing a mode in which the document can be viewed in final form, provided the screen has the necessary capabilities. By nature the WYSIWYG systems cannot display the document structure, although IPS labels components in a special margin. Grif provides commands based on the document structure (especially in movement, selection and creation of objects), but the user interface does not stress the structural aspects because the system assumes the user is unfamiliar with such concepts. One result of this is that Grif has to make use of heuristics in deciding what action should be taken when the user deletes, copies or inserts within the document text. Grif's exact solution is not clear from the literature, nor the degree of success of their approach. The authors simply state that "Grif manages the right choice between manipulations done by structure and those done by text" [Quint86].

Kimura uses various nested and overlapping windows to indicate the document structure, while Furuta uses a combination of grammar productions and windows. Both these systems provide a generalised structure editor in combination with specialised editors for each type of (leaf-) component in the document. The W system uses embedded icons in the document text to delimit objects. Various display modes allow the user to view and edit them under different circumstances.

An important aspect of any syntax-directed editing system is the trade-off between the amount of checking done to ensure correct structure and the flexibility afforded the user when editing the structure. One approach is to use templates or place holders. Parts of the document that are required (or even optional), but still missing are represented by place holders ("ghost windows" or "blank slots" used by Kimura and "buttons" used by Furuta). A penalty paid for this is the extra storage space and extra screen space required for the place holders. Another approach is to allow free entry of text and mark-up (or keywords indicating objects) and to re-parse the input at appropriate times. While this approach works well for programming languages, it is less suitable for documents since that re-introduces the dilemma of the user having to know a "formatting language".

The literature on the systems mentioned before that do not use the place holder approach is not very clear about how they attempt to provide the user with a flexible editing environment. The authors of Grif seem to indicate that it is the document class designer's responsibility to design a class description that is not restrictive. The author of W states that "W will ensure that the document's syntax definition is respected", but says in the same paragraph "W ... is not a syntax checker" [King86]. Coray et al. and van Huu do not mention the issue at all. Obviously it is a non-issue in IPS since the IPS document grammar accepts any object anywhere in the document.

As a final point we examine the output produced by the systems. IPS, W, Grif and Coray et al. all produce fully formatted, ready-to-print documents (W's author mentions that they are considering producing PostScript [Adobe85] and TeX at a future date). Kimura, Furuta and van Huu all produce output for batch formatters. It is doubtful that the scheme van Huu proposed (a mapping table) would be successful, since it does not provide enough power:

About the metalanguage contained in the map table, we must admit that it is more complicated than the elements described in the example of the previous section. In fact, the language must provide the person who is building the map table with instructions capable of describing all possible situations, according to whatever needs of the user, might arise during the description of the document for a formatting system. [vanHu85]

Chapter 2

Framework for a Solution

This chapter describes the outline of a solution to the problem identified in Section 1.2. We discuss the issues and problems associated with each part and indicate how we intend to solve them. Before the solution can be described, however, it is necessary to develop a conceptual model of a document that allows us to model a wide variety of documents in a natural and intuitive way.

2.1 A Document Model

Many document models have been proposed and are being used [Shaw80, Furut82, Horak84, Kimur84a, ISO85]. The model used in this dissertation is an adaptation of the one used by Kimura and Shaw [Kimur84b, Kimur86]. Near the end of this section we list and discuss the differences between the two models.

A document is viewed as an object composed of a hierarchy of *objects*. Each object is an instance of a *class* that defines the possible components and representations of its instances. Typical (low level) classes are document components such as sections, paragraphs, headings, footnotes, figures, and tables. Documents themselves are instances of document classes such as business letters, papers for a particular journal, or theses for a given university.

Objects are either composite or basic. *Composite objects* are composed of other objects.

The composition may be a mixture of ordered sequences and unordered sets of objects. *Basic objects* have no constituent objects. Each basic object has a (possibly empty) string of *non-structural data* associated with it that is referred to as the *content* of the object. The content of a composite object is the aggregate content of its constituent objects.

For example, a technical paper may consist of a sequence of objects: a header, body, appendix and bibliography. Each of these objects are in turn composed of other objects: the body may consist of a sequence of sections, each a sequence of subsections. A given subsection may again be a sequence of paragraphs and figures. Though the figures might be ordered in relation to each other (fig.1 first, then fig.2), they might not be ordered with respect to the paragraphs. The bibliography may be a set of reference items, the order of which will depend on how the document is formatted: in one instance they may be ordered alphabetically and in another, in order of reference.

We view a document as an abstract object that may have many concrete representations. *Abstract objects* are the logical entities comprising a document (such as those listed earlier), while *concrete objects* are the external representations of abstract objects and are usually two-dimensional formatted images.

Each object may have characteristics other than its composition, content and the class to which it belongs. These other characteristics are called *attributes*. All objects belonging to the same class have the same attributes, although the value of a particular attribute may differ from object to object (class instance to class instance).

For example, the object class `figure` may have the attribute `depth`, whose value indicates how much extra empty space should be left for material that is to be pasted in. Another attribute might be `frame` with possible values `box`, `rule`, or `none`, indicating which kind of frame, if any, should be drawn around a figure. Different figures may then have different frames depending on the value of their `frame` attribute.

Non-hierarchical relationships between objects are modelled by *references* between objects. Such references are independent of the composition structure of the objects. Each object class may define which classes of objects may be referenced. References are typically used to model such concepts as cross references, bibliographic references, index terms and footnote references.

As was pointed out earlier, the above model was adapted from that of [Kimur84b]. The differences between the two models are:

1. In our model the user-supplied text of a document is associated (as non-structural data) with basic objects only, while in the Kimura/Shaw model non-structural data may be associated with any object and not just basic ones.

   This is not a fundamental difference, because, as far as this aspect is concerned, either model can be simulated by the other. We did not find any need for non-structural data to be associated with composite objects. Our approach makes both the model and its implementation simpler, because of the clearer distinction between composite and basic objects.

2. Our model does not make provision for shared objects. A shared object is one that is an immediate child of more than one composite object. For example, if the same mathematical formula appears in different places in a document, it is possible to create a single abstract formula object that is shared by all objects that need it.

   We have limited our model to exclude sharing simply for convenience. Shared objects can certainly be included in the model similar to the way that Kimura has done. However, shared objects are not widely used in documents and introduce complexities in the implementation that can be avoided with a different approach as Furuta et al. has indicated:

   > Since trees are more comprehensible than general graphs, and since many documents are primarily tree structured, it may be more desirable to use a tree-structured model and to include general relations among objects as subsidiary information. [Furut82]

   We have taken this approach and have found that references can be used to model shared objects where necessary.

3. The Kimura/Shaw model does not include object attributes as described on page 15.

2.1.1 Document Descriptions

In order to describe a document fully, three components have to be specified: content, syntax, and semantics.

The *content* of the document is the text of the document, i.e. the string of characters that forms the visible part of the document. Ideally this should also include any drawings, figures or other graphical material.

The *syntax* of a document defines the structure of, and relationships between the various objects that constitute the document.

We define the *semantics* of a document to be a description of the visual form or layout into which the content would be formatted. It is therefore that which gives "meaning" to the (abstract) objects in the document.

2.2 Outline of the System

Figure 2.1 illustrates the framework for the solution that we propose. The user creates a document using an editor embedded in the system. Four pieces of information are needed: the class of the document, its content, syntax and semantics. The document class and content is determined by the user, while the syntax and semantics are determined by the document class. For example, the document class could be: business letter, technical paper, personnel form or legal document. General document classes have been given here as examples, obviously there could be many types within each class.

The document syntax and semantics are predefined in a file or data base of class descriptions. The data base may contain several semantic definitions (or layouts) for the same document class. This document class data base may be created by the user, but normally is composed by a "document designer".

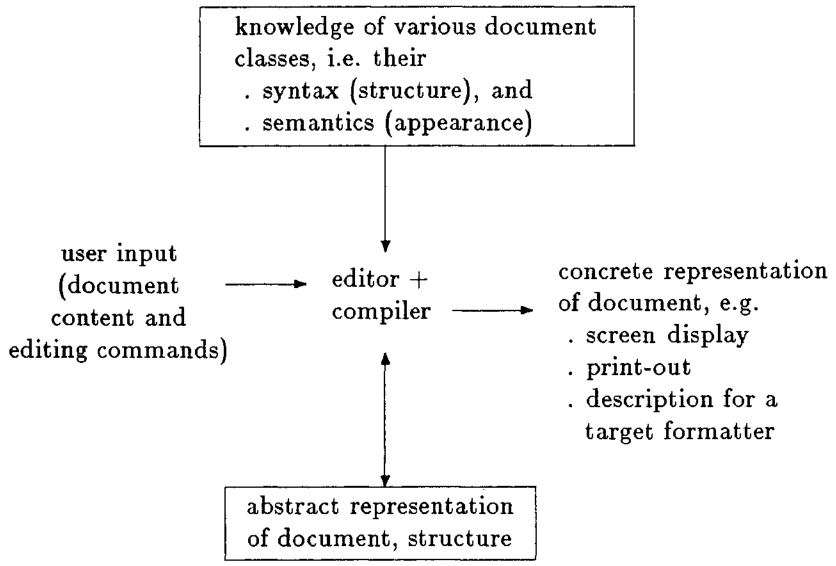

knowledge of various document classes, i.e. their
. syntax (structure), and
. semantics (appearance)
user input (document content and editing commands)
editor + compiler
concrete representation of document, e.g.
. screen display
. print-out
. description for a target formatter
abstract representation of document, structure

Figure 2.1: *Framework for a document preparation system*

While the document is being entered, the editor uses the information in the document class data base to assist the user in creating a correct document. The editor creates an abstract, internal representation of the document that conforms to the class syntax and contains the content of the document.

The abstract document can now be compiled into a concrete one by selecting a semantic definition from the document class description and generating the appropriate output. Two concrete representations are normally required: a screen display which approximates the final appearance of the document, and a description of the document in the input language of some target formatting system. It is highly desirable that the screen display be generated continuously as the document is being created, providing an interactive editing environment.

The individual parts of the system are described in more detail in subsequent sections.

2.3 Document Class Descriptions

The document class data base contains the definitions of various classes of documents. Since a class description is basically a document template (a document description without the content part), a class description consists of two parts: a syntactic and a semantic part.

2.3.1 Document Syntax

This part of a class description contains a description of the syntax or structure of any document belonging to the given class. The structural description includes definitions of the classes (types) of objects in the document class, their composition, and how they may be combined to form documents of the given class. For example, in a "business letter" class there could be such object classes as sender's address, receiver's address, salutation, body and close. The body might be composed of other classes such as paragraphs and lists. One of the syntax rules might state that the salutation must be preceded by the receiver's address and succeeded by the body.

Although not purely syntactic information, this part of an object class definition also includes a list of all attributes associated with that class of object, the possible values that each attribute may have, and the default value of each attribute.

The syntax description further defines which references may exist between objects.

Some of the questions that should be answerable by queries to the syntactic portion of the class data base are the following [IBM84a]:

1. What are legal objects for the given document class?
2. Does a given object have attributes and what are they? (This is not to be confused with determining the value of an attribute.)
3. Are the attributes of an object mandatory or optional?
4. What is the structure of the content of an object?
   (a) Which objects can occur in the content?
   (b) In what order can they occur?
   (c) Which content objects can occur more than once?
   (d) Which content objects are optional?

5. Are references allowed from this object to any other object and if so, to which other objects?

2.3.2 Document Semantics

This part of the data base contains layout information for each object in the class data base. It assigns "meaning" to the objects in a document class by specifying the form or layout in which each object would be presented to the user.

More than one layout may exist for the same document class. For example, a document belonging to the class "technical paper" may have layouts corresponding to the publication standards of various technical journals. There may therefore be a one-to-many relationship between object structure definitions and object layouts.

The information contained in the semantic definition is used at least twice: to determine how an object will be displayed on the screen, and to generate a target document description.

2.4 The Editor

The editor is an easy-to-use, interactive, formatter-independent document editor that allows the user to create documents in a consistent way, regardless of which formatting system is eventually used. The user specifies the document class and the editor obtains the class description from the class data base. The user then proceeds to give a declarative description of the document.

2.4.1 User Interface

Apart from the usual editing functions of copying, moving and deleting selected portions of text, a create function is provided through which the user creates specific document objects (such as headings, paragraphs, and list items). The attributes of each object are obtained from the syntax description of the document class, while the semantic description determines how each object is displayed (as it is being entered).

The editor is syntax-driven. It allows the user to create only legal objects (ones defined in the current document class) in their legal positions (determined by the syntax rules). For example, while entering the receiver's address in the business letter mentioned earlier, the user is not supplied with any other create function than that of creating the salutation. This implies a dynamic, state-driven user-interface: the functions/commands/menus available to the user change as the state of the document or edit-session changes. Specifying a new document class can cause a complete new menu-tree to be constructed.

The editor therefore takes the *generator approach* in language-oriented editor construction [Campb84]. This approach ensures that only acceptable constructs are created by the user.

2.4.2 Partial Documents

While the generator approach ensures that only correct documents are created, it can very easily become too restrictive to be useful. For example, a user might want to create the body of a letter before having to be concerned with the receiver's address. However, the document grammar may force the user to create the address before the body. Not only does the editor have to be flexible enough to allow the user to create objects in any order (but not necessarily at any position), but it also has to be flexible during the editing of an existing document.

Usually when something is modified, it is taken apart and then reconstructed. Similarly, while a document is being edited (modified), it is possible that one or more of the syntax rules must be violated in order to perform the edit effectively. The editor must allow the user freedom to violate the rules "temporarily" while still encouraging the correction of such a violation as soon as possible.

In addition to the standard syntax-driven editing environment, we make use of an unconstrained editing environment (called the *patch area*), as well as introduce the notion of partially correct documents to provide the required flexibility.

2.4.3 Structure Editing vs. Content Editing

One advantage of dealing with documents in a structured way, is the fact that the scope of operations can be specified in terms of the substructures of the document. For example, the sections of a chapter, or the items in a list may be rearranged, or the cursor may be moved to the start of the next chapter with a single command.

To be able to make use of the structure of the document, a user must be able to identify and indicate the various substructures clearly and unambiguously. However, not enough information is contained in the standard paper-like representation of a document for a user to do this. It is furthermore confusing for the user to have to operate on a hidden data structure, where the target of an operation is not directly connected to the context in which the operation is issued. A clear distinction must therefore be made in the editor between structure editing and content editing. At the same time, however, the transition between structure editing and content editing should be natural for the user and should not necessitate a change in the way which the user thinks about what he is doing.

2.5 Concrete Representations

A "compiler" embedded in the system generates formatter-dependent document descriptions. Using the abstract document object produced by the editor, together with a semantic description obtained from the class data base, it generates a document description in the target formatter's description language.

The display hardware is regarded as just another target formatter and the display instructions as its document description language. The only difference is that the system must be able to generate the document description for the display in an incremental manner. This is necessary because the user views and manipulates this representation interactively and it is not feasible, from a performance point of view, to regenerate the entire document each time that the user makes a change, or even just displays a different portion of the document.

The output produced for the display gives a visual feedback of the objects that are in the document. What the user sees on the screen may not necessarily be the exact form of the final output, since the display unit might not have the capability to display the object in its true form. However, the visual feedback enables the user to recognise each object clearly and determine that the display is approximately what is required. The user is not distracted by formatting commands as in document descriptions for batch-oriented formatters.

It is not always clear how much semantic information should be included in the document class definitions. If enough semantic information is included, the compiler should be able to generate a document description for any target formatter without making assumptions as to the availability of external layout or macro definition files. Since it has a complete description of every object (including the parameters that will determine its layout), the compiler can generate the appropriate tag and/or macro definitions for declarative target formatters, or sequences of formatting commands for procedural ones.

However, most declarative target formatters already have layout files for the various styles that are being used. To be able to use these layouts, they have to be "translated" and included in the semantic part of the class data base. It might be much better if the semantic part of the class data base only specifies the semantic class to which an object belongs. Additional semantic information necessary to format the object is then obtained by the target formatter from the external layout file.

Chapter 3

A Document Class Description Language

Document class descriptions lie at the heart of the system outlined in the previous chapter. These descriptions govern the editing process as well as the production of the correct concrete representation of the abstract document. This chapter describes the language with which a document class designer describes a document class. First, however, the design goals for the language are discussed.

3.1 Design Goals

The major design goal for the class description language can (somewhat obviously) be stated as follows:

> To provide a language through which the syntax and semantics of a document class can be defined as needed for the document preparation system outlined before.

The following sub-goals are identified:

1. The language must enable a class designer to describe the syntax of commonly used document classes.

2. It must be possible to describe several sets of semantic information for each document class. In particular, it must be possible to give semantic descriptions for rendering a document in the different input formats required by various batch formatters, as well as in a form suitable for human consumption, and display and manipulation on a terminal screen.

3. It must be easy to write a class description using the language. The language must be expressive, simple and concise.

4. As pointed out in item 2, one of the concrete representations of a document must be suitable for the interactive manipulation of the document. Class descriptions must therefore be amenable to incremental processing in order to produce the concrete representation interactively and incrementally as the document is changed by the user.

Further sub-goals may be defined for that part of class description language that defines the syntax of a document class:

1. The language must be able to express the complete syntax of commonly used document classes. The syntax definition of a document class includes:

    - Identifying classes of objects (such as sections, headings, lists, and paragraphs) that compose the document class,
    - Defining the composition of object classes from more elementary classes. This may be done by using such concepts as containment, sequencing, repetition (both bounded and unbounded), selection, optionality, and the interchange or shuffle of other objects,
    - Defining which cross references may exist between object classes (e.g. a sentence may refer to a section or an item in the bibliography) and under what circumstances, and
    - Defining attributes of object classes, such as those described on page 15.

2. Notational conveniences (succinctness) must be provided for the class designer while maintaining the readability and comprehensibility of the class description.

3. The syntax description must be implementable in the sense that it must be possible for the document preparation system to use the description as a guide (or template) for creating and editing documents of that class.

3.2 The Language

The class description language defines classes of objects. An object class definition consists of three parts: a structure definition, attribute definitions, and a semantic definition. Using a Backus-Naur-Form (BNF) notation we may write:

<class_definition> ::= <struct_def> <attr_defs> <semantic_def>;

In the subsections that follow, each of these class definition components will be discussed in more detail[1].

Note: When discussing class descriptions, and where the intent is clear, the term object is often used for object class.

3.2.1 Structure and Composition

The syntax of the structure definition part of an object class definition can be defined in BNF as follows[2]:

```
<struct_def>   ::= <class_name>
                 | <class_name> <composition>
<composition> ::= = <extended_reg_exp> <exceptions>
<exceptions>   ::= ε
```

[1]Appendix A contains the complete BNF definition for a class description, used in generating a class description parser.

[2]The symbol ε is used to denote the null expression.

```
                    | : <except_list>
<except_list>   ::= <except>
                    | <except>, <except_list>
<except>        ::= +<class_name>
                    | -<class_name>
```

In other words, the structure definition of a class consists of a class name (<*class_name*>), followed by an optional definition of the composition of the class (<*composition*>). The composition is defined with the use of extended regular expressions (<*extended_reg_exp*>) that may be modified by a list of exceptions (<*except_list*>).

If a composition is given in the structure definition of an object class, the objects in that class are composite objects. If no composition is given, the objects are basic objects (cf. Section 2.1).

Extended regular expressions are regular expressions [Hopcr79] extended with some notational conveniences. The simplest extended regular expressions are the null expression and an expression consisting of a single object class name. They respectively define an empty composite object and one composed of a single other object. For example,

```
heading = TEXT;
TEXT;
```

defines `heading` to be a class of composite objects composed of a single object of the class `TEXT`. `TEXT` in turn is a class of basic objects. Recall that a basic object has non-structural data (text, in this instance) associated with it.

More complex expressions are constructed from symbols and operators. Symbols are just class names as defined by the user. The valid operators are defined in the next few paragraphs.

An extended regular expression defines a set of compositions, any one of which would be a legal composition of an instance of the object class that is being defined. For example, the expression

$a(b|c)d$

defines the two compositions $a\ b\ d$ and $a\ c\ d$. Writing

$x = a(b|c)d$

means that an object belonging to the class x must be composed of an a-object, followed by either a b-object or a c-object and followed by a d-object.

If **a** and **b** are extended regular expressions (not necessarily only symbols), then the following are also extended regular expressions and have the indicated meaning:

a b Sequencing. Any of the compositions defined by **a** followed by any of the compositions defined by **b**, e.g.

$x = c\ d$

defines a class x object to be composed of a class c object followed by a class d object.

a | b Choice. Any of the compositions defined by **a** or any of the compositions defined by **b**, e.g.

$x = c|d$

defines a class x object to be composed of either a class c object or a class d object.

a#b Shuffle [Shaw78]. All compositions defined by **a** are interleaved with all compositions defined by **b**. The relative order of the symbols in each composition remains the same, but the symbols of one composition may be interleaved with that of the other in any way. Formally (using the notation of [Hopcr79]), for a, $b \in S^*$, where S is some alphabet and * indicates Kleene closure,

$$a\#b = \{a_1b_1a_2b_2 ... a_kb_k \mid a_i, b_i \in S^* \ and \ a = a_1a_2 ... a_k, \ b = b_1b_2 ... b_k\}$$

For example, if w, x, y and z are symbols, then

$$wx\#yz = \{wxyz, wyxz, wyzx, ywxz, ywzx, yzwx\}$$

Table 3.1: *Operator Precedence*

highest	()
	?, *, +
	sequencing
	|
lowest	#

a? Optional. The compositions defined by **a** or the empty set.

a* Unbounded repetition, including none. Any of the compositions defined by **a** may occur in sequence any number of times, including not at all.

a+ Unbounded repetition, but at least once. Any of the compositions defined by **a** may occur in sequence any number of times, but it must occur at least once.

(a) Grouping. Brackets are used in the normal mathematical sense to impose precedence on operations.

a*<*n*> Bounded repetition, including none. <*n*> must be a number greater than 0. Any of the compositions defined by **a** may occur in sequence any number of times less than or equal to <*n*>, including not at all.

a+<*n*> Bounded repetition, but at least once. <*n*> is a number greater than 0. Any of the compositions defined by **a** must occur (in sequence) at least once, but not more than <*n*> times

Table 3.1 shows the precedence of the various operators.

When writing object definitions, spaces and newline characters are significant only in that they delimit symbols.

Figure 3.1 defines a document class `TechDoc` to be composed of a `titlePage`, a `body`, and an optional `appendices`. The `body` is composed of either one or more `parts` or one or more `sections`, etc. Only one basic object is defined, namely `TEXT`. Figure 3.1 also illustrates a

```
%basicObject = paragraph | ordList | unordList | example;

 TechDoc = titlePage body appendices?;

 titlePage = title author date abstract?;
 title = TEXT;
 author = TEXT;
 date = TEXT?;
 abstract = %basicObject+;

 body = part+ | section+;
 part = heading  %basicObject*  section*;
 section = heading  %basicObject*  section*;
 heading = TEXT;
 paragraph  = TEXT+;
 ordList = (listItem | listPart)+;
 unordList = (listItem | listPart)+;
 listItem = paragraph %basicObject*;
 listPart = paragraph  (paragraph | example)*;
 example = (aLine | %basicObject)+;
 aLine = TEXT;

 appendices = section+;
 TEXT;
```

Figure 3.1: *Class description for a general document*

further notational convenience, namely the fact that expressions (or sub-expressions) may be given symbolic names by preceding the name with a `%`. `%basicObject` has for example been defined as a symbolic name. All occurrences of such symbolic names in other expressions are treated as standing for the expression itself. In other words, they define parenthesized string substitutions and are similar to parameterless macros found in programming languages.

<*exceptions*> define an optional list of exceptions that modify the composition of an object and its constituents and are similar to the exceptions defined in SGML [ISO85]. It is often convenient to define one object in terms of another, for example "A is like B, except that it does not contain any Y objects" (e.g. an abstract is like a section, except that it does not contain any subsections), or "A is like B, except that object Y can be added at any point".

Two kinds of exceptions may be specified: inclusions and exclusions, indicated by "+" and "-" respectively.

An inclusion indicates that objects of a given class may appear anywhere in an object or its constituents. For example, index term definitions can usually appear anywhere in the body of a document. It is therefore convenient to write

```
body = part+ | section+ : +indexTerm
```

rather than having to include `indexTerm` in the definition of `body` as well as in the definition of a copy of every object class that form constituent objects of `body`.

Inclusions of the form $+R_1, +R_2, \ldots, +R_n$ modify the composition of the affected object classes by replacing each symbol and unary operator (if any) that applies to that symbol alone, with another expression. A sequence

$$S\eta$$

where S is a terminal or non-terminal symbol and η is either nothing or one of the unary operators ?, $*$, or $+$, is replaced with

$$(R_1^* \# R_2^* \# \cdots \# R_n^* \# S)\eta$$

The operators # and | remain unaffected. The class definitions that are modified in this way (the affected object classes referred to before), are the class definitions of all objects that may appear in the tree rooted by an object of the class being defined. For example, using the `indexTerm` example of before and referring to Figure 3.1, the class definitions for `body`, `heading`, `paragraph`, `ordlist`, `unordList`, `listItem`, `listPart`, `example`, and `aLine` will all be modified. In other words, the definitions of all the composite objects that may appear inside `body`, as well as that of `body` itself will be modified. In particular, the definition for `body` with `indexTerm` included, results in the production

```
body = (indexTerm* # part)+ | (indexTerm* # section)+
```

An exclusion has the effect of removing the excluded symbol, together with the operator(s) most tightly bound with it, from the expressions that define the composition of the affected objects. It is therefore as if the excluded symbols do not appear in the composition expressions at all.

For example, in the class description of Figure 3.1 `example` is defined as

```
example = (aLine | %basicObject)+;
```

which implies that examples may contain examples. If we wish to prohibit this, we could use an exclusion:

```
example = (aLine | %basicObject)+ : -example;
```

which effectively changes the composition of `example` to

```
example = (aLine | paragraph | ordList | unordList)+;
```

and also changes the composition of any `listItem` or `listPart` that appears inside an example. The composition for `listPart` is, for instance, changed from

```
listPart = paragraph (paragraph | example)*;
```

to

```
listPart = paragraph (paragraph)*;
```

If at any point a symbol is specified as both included and excluded, the exclusion takes precedence.

Finally, note the difference between basic objects and empty composite objects:

```
someObject;
```

defines `someObject` to be a basic object, and all basic objects have user text associated with them, even though the text might be the null string. On the other hand,

```
someObject = ;
```

defines `someObject` to be a composite object with no children or constituent objects. In this case `someObject` has no user text associated with it and, in terms of our model, has no content. The usefulness of this is illustrated in the footnote example given in the next section.

3.2.2 Attributes

In the previous section we have described the structure definition part (<*struct_def*>) of an object class definition

<*class_definition*> ::= <*struct_def*> <*attr_defs*> <*semantic_def*>;

In this section we describe the <*attr_defs*> part, namely the definition of any attributes that may be assigned to an object class. As pointed out in the previous chapter, attributes are characteristics of objects other than their composition and content. For example, we can assign the attributes `emphasis` and `quote` to the basic object TEXT of Figure 3.1 by writing

```
TEXT
  { emphasis: (some, more, most) = NONE,
    quote   : (yes, no) = no
  };
```

Attribute definitions are optional and, if present, appear in braces ({}) after the structure definition of an object class. An attribute definition has three components: the attribute name (e.g. `quote`), its type (or a list of possible values that it may have, e.g. `(yes, no)`), and the attribute's default value (`no`). The attribute type can be specified as an enumerated type (as in our example), a reference (or pointer) type, or as one of the built-in types `INTEGER`, `REAL`, `CHAR` or `STRING`.

The formal syntax of attribute definitions is given by the BNF definition:

$$
\begin{array}{lll}
<\!attr\_defs\!> & ::= & \varepsilon \\
 & | & \{\ <\!attr\_list\!>\ \} \\
<\!attr\_list\!> & ::= & <\!attr\_def\!> \\
 & | & <\!attr\_def\!>,\ <\!attr\_list\!> \\
<\!attr\_def\!> & ::= & <\!attr\_name\!>\ \mathtt{:}\ <\!attr\_val\_list\!>\ =\ <\!default\!> \\
 & | & <\!attr\_name\!>\ \hat{}\ <\!class\_name\!>\ =\ <\!default\!>
\end{array}
$$

For the sake of brevity we have not included the formal definition of *<attr_val_list>* as this may be obtained from Appendix A.

The default value of an attribute (*<default>*) can be any of the values given in the list of allowable values, or one of `NONE` or `REQUIRED`. `NONE` indicates that the attribute has no value and `REQUIRED` indicates that the user has to specify a value at the time that an instance of the object is created.

The second form of *<attr_def>*, the ^-form, defines an attribute that has as value a reference to any object of the indicated class. The only valid default values for such an attribute are `NONE` and `REQUIRED`.

As an example of the use of both forms of attributes, we have added emphasised and quoted text as well as footnote objects to the class description of Figure 3.1 as shown in Figure 3.2. Note that attributes have been defined for three object classes: `title`, `TEXT`, and `fnRef`. In the new description, paragraphs are defined as composed of a mixture of `TEXT` and `fnRef` objects, where the latter are empty composite objects (they cannot have any constituent objects and cannot contain any text). A `fnRef` object has an attribute that

```
%basicObject = paragraph | footnote | ordList | unordList | example;
%anyText = TEXT (TEXT | fnRef)*;

 TechDoc = titlePage body appendices?;

 titlePage = title author date abstract?;
 title = PLAIN_TEXT
   { footref ^ footnote = NONE };
 author = PLAIN_TEXT;
 date = PLAIN_TEXT?;
 abstract = %basicObject+;

 body = part+ | section+;
 part = heading  %basicObject*  section*;
 section = heading  %basicObject*  section*;
 heading = PLAIN_TEXT;
 paragraph = %anyText;
 footnote = %basicObject+ : -footnote, -fnRef;
 ordList = (listItem | listPart)+;
 unordList = (listItem | listPart)+;
 listItem = paragraph %basicObject*;
 listPart = paragraph  (paragraph | example)*;
 example = (aLine | %basicObject)+;
 aLine = %anyText;

 appendices = section+;

 PLAIN_TEXT;
 TEXT
   { emphasis: (some, more, most) = NONE,
     quote   : (yes, no) = no
   };
 fnRef =
   { footref ^ footnote = REQUIRED };
```

Figure 3.2: *General document class with footnotes*

must be initialised by the user and that points to a footnote object. Footnotes have been added as footnote objects (and included in the %basicObject symbolic definition) and are composed of one or more %basicObjects. However, neither footnotes, nor references to footnotes may appear within a footnote because of the exclusions specified in the footnote structure definition.

Notice that we have added a second basic class, PLAIN_TEXT, that, in contrast with the TEXT class, does not have any attributes. Attributes can always be set by the user, but in this document class definition there are instances where we do not want the user to be able to set the attributes of text (such as the emphasis attribute for example), because the attribute values are dictated by the environment (e.g. author names in the title page).

3.2.3 Semantic Definitions

The previous two sections described aspects of the class description language that relate to the syntax of a document class. This section describes that part of the language that defines the semantics of a document class, i.e. that part that defines how concrete representations of an abstract object are to be produced.

The concept that is used is that of "unparsing schemes" [Medin82, Reps84]. The term unparsing scheme refers to the fact that an abstract document can be represented by a parse tree of the concrete document. An unparsing scheme therefore specifies how the parse tree should be "unparsed" to obtain the concrete representation.

For example, the document fragment of Figure 3.3 can be regarded as having the structure depicted in Figure 3.4. An intelligent enough parser (e.g. a human being) could parse this fragment to produce the parse tree of Figure 3.5. This parse tree is a representation of the abstract document of which Figure 3.3 is a concrete representation.

In order to produce a paper-like document once again, the parse tree must be unparsed, i.e. each node must be visited in the correct order and the appropriate output must be generated at the appropriate time. Different unparsing schemes may be used to obtain different concrete representations.

Appendix H

Hyphenation

It is better to break a word with a hyphen than to stretch interword spaces too much. Therefore TEX tries to divide words into syllables when there's no good alternative available.

But computers are notoriously bad at hyphenation. When the typesetting of newspapers began to be fully automated, jokes about "the-rapists who pre-ached on wee-knights" soon began to circulate.

Figure 3.3: *A document fragment.* (Taken from [Knuth84])

An *unparsing scheme* is a set of unparsing methods. Each method describes one way of unparsing a class of objects (or class of nodes in the parse tree). More than one method may be given for a single class within one scheme. The two paragraphs in the document fragment in Figure 3.3 illustrate why this might be necessary. Both objects belong to the class paragraph, but they are unparsed differently (the first line of the second paragraph is indented) depending on whether they follow a heading or not. The unparsing scheme that produces the document in the form of Figure 3.3 therefore needs at least two methods for unparsing paragraphs.

An *unparsing method* is composed of a (possibly empty) set of variables and a set of unparsing actions. An *unparsing action* is an ordered pair (a transition and a program) that associates an unparsing program with a transition in a finite automaton constructed from one of the extended regular expressions that defines the class syntax. An *unparsing program* is a sequence of unparsing instructions that is executed when the particular transition is made during execution of the finite automaton[3]. *Unparsing instructions* produce output

[3]Every automaton is assumed to have at least two "transitions" that are always executed: one labelled `initial` that leads into the start state and is always executed to start the automaton, and one (or more)

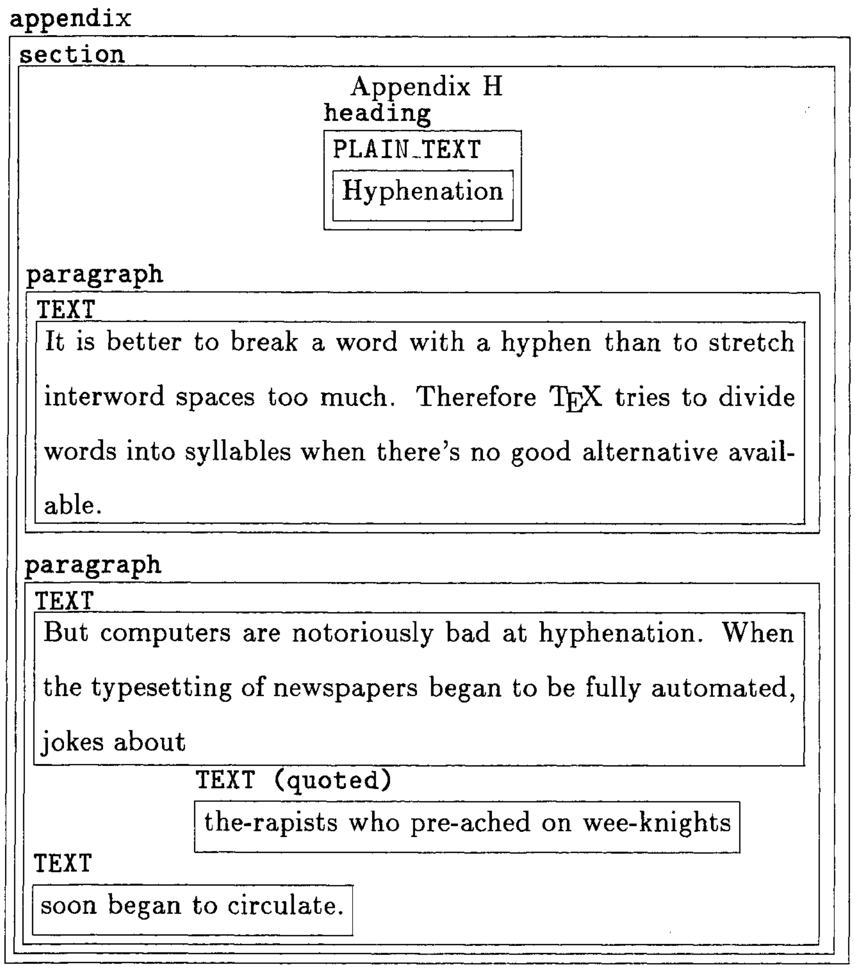

appendix
section
Appendix H
heading
PLAIN_TEXT
Hyphenation
paragraph
TEXT
It is better to break a word with a hyphen than to stretch interword spaces too much. Therefore TEX tries to divide words into syllables when there's no good alternative available.
paragraph
TEXT
But computers are notoriously bad at hyphenation. When the typesetting of newspapers began to be fully automated, jokes about
TEXT (quoted)
the-rapists who pre-ached on wee-knights
TEXT
soon began to circulate.

Figure 3.4: *Structure of a document fragment*

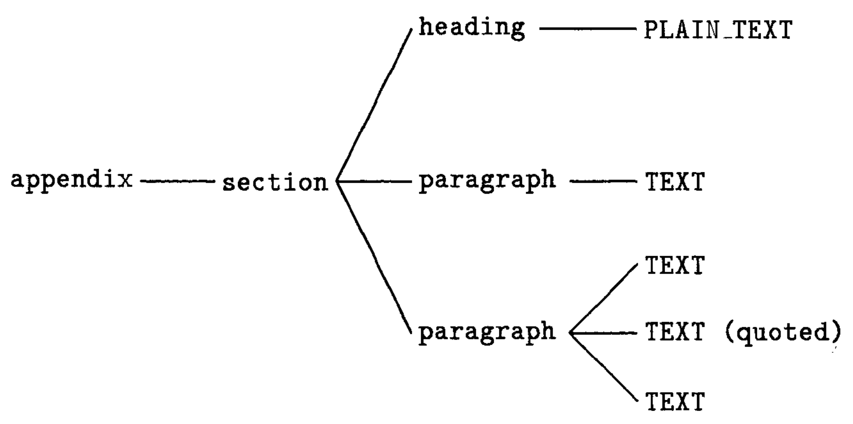


Figure 3.5: *Parse tree representing a fragment of an abstract document*

and/or change the value of the variables accessible to the method.

For example, Figure 3.6 shows a fragment of the syntax and semantics of a document class to which the document fragment of Figure 3.3 might belong. The semantic definition for each object (the part enclosed in square brackets [...]) defines a scheme named `gml` that produces a concrete representation of the object in a form suitable to serve as input to a GML formatter. Only one method has been defined for each object and in each case the method has been named **`Standard`**[4]. Notice that a method may have parameters. For example, the method defined for `section` has a single integer parameter. Parameter passing is always by value. The actions for each method are listed in the form

<transition> **:** *<program>*

e.g. `section:` `@Standard(1)` is one of the actions for the `appendices` object.

Figure 3.7 shows a finite automaton constructed for the `appendices` object defined in Figure 3.6. Each transition in the automaton is labeled with the symbol causing the

labelled `final` that leads from any final state to an imaginary "truly final" state.

[4]There is nothing special about the name `Standard` – any name could have been used, and each method could have had a different name. The only requirement is that different methods for unparsing the same object within a single scheme, have unique names.

```
appendices = section+
   [ /gml/
       Method Standard()
         initial:  @nl ":APPENDIX.",
         section:  @Standard(1),
         final:
       endMethod
   ];

section = heading  paragraph*  section*
   [ /gml/
       Method Standard( level:integer )
         initial:   @nl ":H" level ".",
         heading:   @Standard(),
         paragraph: @Standard(),
         section:   if level < 6 then
                       @Standard(level+1)
                    else
                       @Standard(level)
                    endif,
         final:
       endMethod
   ];

heading = PLAIN_TEXT
   [ /gml/
       Method Standard()
         initial:     ,
         PLAIN_TEXT: @Standard(),
         final:
       endMethod
   ];

paragraph  = TEXT+
   [ /gml/
       Method Standard()
         initial:  @nl ":P." @nl,
         TEXT:     @Standard(),
         final:
       endMethod
   ];
```

```
PLAIN_TEXT
   [ /gml/
       Method Standard()
         initial:  ,
         final:
       endMethod
   ];

TEXT
   { emphasis : (some, more, most) = NONE,
     quote    : (yes, no) = no
   }
   [ /gml/
       Method Standard()
         initial:  if quote = yes then
                      ":Q."
                   endif
                   if emphasis = some then
                      ":HP1."
                   elseif emphasis = more then
                      ":HP2."
                   elseif emphasis = most then
                      ":HP3."
                   endif ,
         final:    if quote = yes then
                      ":eQ."
                   endif
                   if emphasis = some then
                      ":eHP1."
                   elseif emphasis = more then
                      ":eHP2."
                   elseif emphasis = most then
                      ":eHP3."
                   endif
       endMethod
   ];
```

Figure 3.6: *A sample class definition.* The unparsing scheme generates a document description in GML.

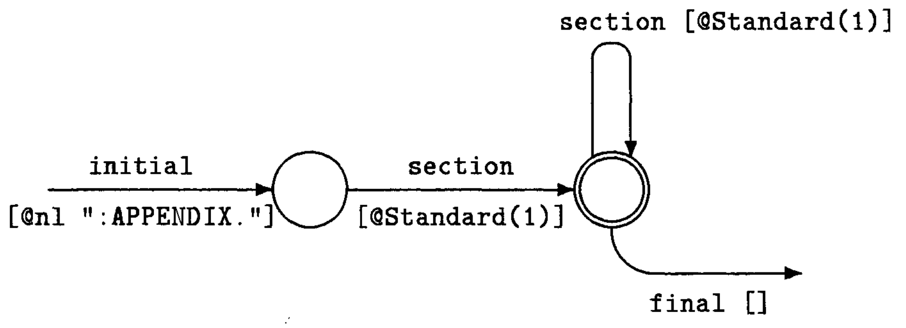


Figure 3.7: *Finite automaton and unparsing programs for* appendices *objects*

transition (e.g. section), as well as its unparsing program in square brackets. For simplicity we have assumed a single unparsing scheme with only one method.

The Standard method in the gml scheme for appendices states that upon visiting an appendices node during the unparsing process, the output :APPENDIX. (a GML formatting directive) must be generated on a new line. Immediate children of the appendices node belonging to the class section should then be visited and unparsed with the Standard method in the gml scheme defined for section. In other words, the method section.gml.Standard (using a Pascal-like notation) must be used to unparse section nodes when they are children of an appendices node. As we have mentioned before, section.gml.Standard expects a single integer parameter indicating the section level.

Unparsing methods for basic objects such as PLAIN_TEXT and TEXT can only have initial and final unparsing actions. It is implied that the text associated with a basic object is generated (emitted) between the initial and final unparsing actions for that object.

We now give a more formal definition of the syntax of the semantic definition of a given object class. Since the semantic definition contains many lists of things, some of which may be empty and others not, and some whose items must be separated by special delimiters and others not, and since standard BNF notation for lists are cumbersome, we use the following notation in describing the syntax of the semantic definition:

$\ldots <xxx\Theta>^{+} \ldots$

is used as shorthand for

$\ldots <xxx\_list> \ldots$

$<xxx\_list> ::= <xxx> \mid <xxx>\ \boldsymbol{\Theta}\ <xxx\_list>$

and

$\ldots <xxx\Theta>^{*} \ldots$

is used as shorthand for

$\ldots <n\_xxx\_list> \ldots$

$<n\_xxx\_list> ::= \varepsilon \mid <xxx\_list>$

$<xxx\_list> ::= <xxx> \mid <xxx>\ \boldsymbol{\Theta}\ <xxx\_list>$

In this notation, Θ can be any non-alphanumeric symbol or delimiter such as a comma, or a space. In the latter case it may be omitted altogether.

The syntax for the semantic definition of an object is then as follows:

$<semantic\_def> ::=$ [$<scheme>^{*}$]

$<scheme> ::= /<scheme\_name>/\ <method>^{*}$

$<method> ::=$ **Method** $<method\_name>\ <parameters>$

$<local\_vars>$

$<action,>^{+}$

endMethod

$<parameters> ::= (\ <var\_def,>^{*}\ )$

$<var\_def> ::= <var\_name>$ **:** $<var\_type>$

$<local\_vars> ::= \varepsilon \mid \{\ <var\_def,>^{+}\ \}$

$<action> ::= <trans\_name,>^{+}$ **:** $<program>$

A *<program>* is simply a sequence of unparsing instructions. Appendix B contains a description of all available unparsing instructions, while Appendix C.1 contains a complete class description with three unparsing schemes.

Note that in general there are three kinds of "variables" that are used in unparsing programs:

1. *Attributes* are associated with an object and are global to all unparsing methods in the object class. As far as the unparsing programs are concerned, attributes are read-only variables.

2. *Method parameters* are variables that are passed to a method when the latter is invoked (from within an unparsing program) to unparse an object. Method parameters are the same as call-by-value parameters in the C programming language.

3. *Local method variables* are variables that are local to a method and are similar to local variables in a C subroutine.

At least one unparsing scheme must be provided in a class description, namely one that generates a concrete representation for the display screen. The first scheme defined for the root object of the document class is always taken to be this scheme. For example, in the class description given in Appendix C.1, the object `TechDoc` is the root of the class, and the scheme `display`, the first scheme given in the root's semantic description, will be used for display purposes.

In general, the unparsing scheme used for display purposes differs from other unparsing schemes only in that the `@unparse` instruction may not be used in any display unparsing program. This restriction results from the fact that the display unparsing scheme must be executable in an incremental fashion and, as explained in Appendix B, the `@unparse` instruction allows for the unparsing of arbitrary subtrees in the document tree, which leads to difficulties in determining at which point to restart the unparsing process when doing incremental unparsing.

3.3 Adequacy of The Language

The question arises as to the completeness of the class description language. Is it possible to describe any class of document using the language? We will examine the question in two parts, namely whether the language is powerful enough to describe the syntax (structure) of any document, and whether the semantic descriptions are powerful enough to provide any concrete representation.

3.3.1 Syntax

We define a document class to be context-free if the class structure can be described by a context-free grammar (see for example [Hopcr79] for the definition and properties of context-free grammars). Our class description language is sufficient to describe any context-free class. This is true since every context-free grammar can trivially be represented in our language by simply using the sequencing and choice operators.

On the other hand, if we disregard attributes and references, our language is no stronger than a context-free grammar, and can therefore only describe context-free classes. In order to prove this, we show that for any class description written in our language, there is a context-free grammar that describes the class' structure (disregarding attributes and references).

First we show that the extended regular expressions that we use, are equivalent to regular expressions: The sequencing operator as well as choice (|), nullable repetition (*), and grouping are all the standard operators found in regular expressions. The operators optional (?), non-nullable repetition (+), and the bounded repetitions are regular because there are well-known ways in which they can be written in terms of the other regular operators, e.g.

$$a? = (a|)$$

and

$$a+ = aa*$$

In order to show that the shuffle operator (#) is regular, we construct a finite automaton that accepts the language described by $r_1\#r_2$, where r_1 and r_2 are regular expressions[5].

Let $M_1 = (P, \Sigma, \delta_1, p_0, F_1)$ be the deterministic finite automaton (DFA) accepting r_1, where P is a set of states, Σ an alphabet, δ_1 the transition function, p_0 the start state, and F_1 the set of final states. Let $M_2 = (Q, \Sigma, \delta_2, q_0, F_2)$ similarly be the DFA accepting r_2. The (non-deterministic) cross-product machine $M' = (P \times Q, \Sigma, \delta', [p_0, q_0], F_1 \times F_2)$ will accept $r_1\#r_2$, if δ' is defined as follows:

For $p, r \in P$, $q, s \in Q$, and $a \in \Sigma$,

$$\delta'([p,q],a) = [r,s]$$

if and only if

$$\delta_1(p,a) = r \text{ and } q = s$$

or

$$\delta_2(q,a) = s \text{ and } p = r$$

We have already indicated on page 31 how extended regular expressions with exceptions are re-written as extended regular expressions without exceptions. It is therefore now clear that our extended regular expressions are indeed equivalent to regular expressions.

Having shown the extended regular expressions to be regular, it is straightforward to show that for any class description written in our language, there is a context-free grammar that describes its structure (disregarding attributes and references): Since a context-free grammar can be constructed for every regular set [Hopcr79], it is possible to construct a context-free grammar describing the composition of every composite object class in a document class description. If we combine these grammars into a single one by repeatedly applying the substitution operation for context-free languages, we obtain a single grammar

[5]Ginsberg [Ginsb77] has shown shuffle to be regular using a non-constructive proof. The constructive proof given here (for which the help of J. H. Johnson is gratefully acknowledged) is only provided to give an indication of how this operator is implemented in the prototype of Chapter 5.

for the class description. The closure property of context-free languages under substitution [Hopcr79] ensures that this grammar is also context-free.

As we have seen, the description language makes use of an extended BNF notation to define a context-free grammar for each document class. The right hand side of each rule in the grammer is an extended regular expression, which we have shown to be equivalent to regular expressions. This is an important aspect of the description language and allows us to combine features of finite state automata (FSAs) with those of push-down automata (PDAs) [Hopcr79] in the construction of a syntax-controlled document preparation system.

It is well known that context-free languages are recognised (or generated) by PDAs. PDAs however, do not lend themselves easily to incremental execution. One automaton is constructed for the complete grammar, and it is a non-trivial task to reconstruct the status of the PDA if the user moves to an arbitrary point in the document and wishes to edit it. Things become much simpler, however, if methods similar to those of recursive transition networks (RTNs) [Woods70] are used to handle the grammar.

In our implementation the right-hand side of each rule in the class grammar is represented by a finite state automaton. In other words an FSA is used to recognise and/or generate the constituents of each object in the document class. The FSAs are used to determine the objects which can be added to the document, as well as the locations where the objects can be added. The legality of structural editing operations are determined by using the appropriate FSA to test the result. The alphabet of the FSA is simply all the symbols that appear on the right hand side of the rule. The FSA does not make a distinction between terminal and non-terminal symbols — non-terminals are not expanded, but are treated as terminals[6].

The push and pop operations of RTNs are manifested in the context (FSA) switching that occurs when the user traverses the structure of a document, as well as in the recursive calls to unparsing methods that are executed while a document is being unparsed.

[6]This contrasts with Johnson's INR system [Johns85] (a system for the manipulation of finite automata) in which non-terminals are always expanded and their FSAs embedded in their parent's FSA.

Since our language describes context-free classes, we must ask the question whether all document classes are context-free. This is certainly not the case, since one might want to construct a document class that corresponds to the context-sensitive language $\{a^i b^i c^i | i > 0\}$ — e.g. the class of rectangular tables having three rows. (See for example [Hopcr79] for more on this and other context-sensitive languages.) However, it is not clear that such document classes are all that useful. In fact, empirical evidence suggests that similar to programming languages, most useful document classes are indeed context-free, or can quite easily be modelled with context-free languages. For example Kazman [Kazma86] describes how the text of the Oxford English Dictionary (OED) is being structured using finite state transduction. That project strongly indicates that the structure of the OED can for the most part be described using only regular grammars, and the OED arguably belongs to one of the more complex document classes.

Further evidence in support of the adequacy of our language lies in the fact that our language closely resembles SGML [ISO85], which has recently been accepted as an international standard for document description by the International Organisation for Standardisation.

There are, however, two concepts that are not supported in our language:

1. The language does not support the idea that the order of the constituent objects of an instance of a particular class is imposed by its semantics (i.e. the unparsing scheme used to unparse the instance), rather than by the instance itself. In other words an instance of a class described in our language is always a sequence and never a set.

   While this is not a major deficiency, it could have been useful. For example, different publications may require the same components in a bibliography entry, however, the order in which they must appear might differ (e.g. author, title, publication, date as opposed to author, date, title, publication). If it had been possible to let the order be determined by the unparsing scheme, it would have been possible to define a single document class that catered for both cases.

   The reason for this deficiency is twofold: First, the language does not make a distinction between variable arity composite objects and fixed arity ones as in the ALOE system [Medin82]. It is therefore not possible to refer to the n^{th} child of an object in an unparsing scheme (specifying for example that it should be unparsed in the i^{th} position), and know beforehand which class that object would belong to. The second reason has to do with the mechanism with which an abstract object is unparsed, namely by executing a finite automaton. The automaton takes as input the class names of the children of the object in a left to right order. It is certainly possible to define a function that will permute the input as desired, given that certain limitations are introduced with regard to variable arity objects. However, this introduces further complications with respect to incremental unparsing, and the added capability was not deemed worth the added complexity.

2. Our language does not support shared objects, simply because, as explained in Section 2.1, the document model on which the language is based, does not support it. It is possible, however, to simulate shared objects through the judicious use of references.

3.3.2 Semantics

The language is certainly not complete as far as its semantic descriptions are concerned because it is, for example, not possible to describe a concrete representation that involves breaking the document into pages[7]. This in itself should not be of great concern because it is not the intention that the language be able to describe arbitrary concrete representations.

There are two major types of concrete representations that we wish to describe: those that are intended for human consumption (e.g. as displayed on the screen), and those that are intended as input to batch document formatters. For the former it is not necessary that the language cater to every conceivable representation of the document since the display representation is not the final one. These concrete representations only serve as ways to present still-to-be-formatted documents to the user in a way that is reasonably familiar to

[7]The language provides no mechanism for keeping track of the number of lines on the current page, nor is there a way of deferring the unparsing of objects such as footnotes to the appropriate time and place.

the user. It is sufficient that document objects only be displayed in distinctive ways so that the user can recognise them for what they are, and distinguish between them.

The second class of concrete representations are more demanding, however, because we would like to write unparsing schemes to produce input for any target formatter. Is this possible with our language?

There is no definitive answer. We have tried several experiments and have used the language to write unparsing schemes for a variety of formatting languages, such as GML, LaTeX, TROFF and Script, as well as for a class description of class descriptions and so far it has proven to be adequate (see the appendices for examples of class descriptions).

The ability to write unparsing schemes for any formatting language is not our only concern. One of the design goals for the class description language has been the ability to specify several sets of semantics for a single document class. The purpose behind this being the desire that once a document has been created, the system should be able to generate the appropriate input file for any batch formatter that is supported. While this is possible for some document classes and some batch formatters, it is not universally possible.

The input language for every batch formatter implies some document syntax. Sometimes the syntax for two different formatters are so different (because they are based on different document models) that it becomes extremely difficult, and sometimes even impossible to incorporate them into a single class description. As an example we have given a partial class description for business letters in Appendix C.2. It contains semantic definitions for GML and LaTeX. Even though the models for letters in GML and LaTeX are quite different, they could have been combined in a single class description (as in Appendix C.2) if it had not been for the incompatibility of their `close` objects. For GML the appropriate class description is

```
close = greeting name;
greeting;
name = aLine+;
aLine;
```

which corresponds to the GML fragment

```
:CLOSE.final greeting text
author's name
title, etc.
:eCLOSE.
```

In LaTeX on the other hand, it is necessary to define the `close` object as

```
close = name greeting;
greeting;
name = aLine+;
aLine;
```

in order to generate the LaTeX specification

```
\signature{author's name \\ title, etc. }
\closing{final greeting text}
```

The two models are incompatible in terms of our class description language because the LaTeX `close` requires its constituent objects in the opposite order as the GML `close`. The only way that this situation can be handled in our class description language is to incorporate the content of one of the objects as an attribute of `close` — for example as was done in Appendix C.2.

This example shows that it is necessary to first find a common model between formatters before writing the appropriate class description and unparsing schemes, rather than writing a class description and then adding unparsing schemes as the need arise. The latter approach might not always be successful.

Lastly it needs to be pointed out that the method currently being used to assign unparsing programs to transitions, namely by referring to terms in the regular expressions, does not exploit the full power of finite automata. A single term in a regular expression might correspond to more than one transition in a finite automaton (even in a deterministic finite automaton), but one might want to assign different unparsing programs to each transition. Consider for example the definition

```
dL = tH? dH? (t+ d?)+
```

TERMS	DEFINITIONS
termA1	
termA2	A block of text, called a definition, describing the term(s) listed next to and immediately above it.
termB1	Some terms, like the C group below might not have a definition.
termC1	
termC2	

Figure 3.8: *A sample definition list*

which can be interpreted as the structure of a definition list (`dL`) that consists of optional headings (`tH` and `dH`) for the terms (`t`) and term definitions (`d`) in the list. Such a definition list is illustrated in Figure 3.8.

Figure 3.9 shows a finite automaton for this definition list class. Notice that although each term (symbol) appears only once in the regular expression, both `dH` and `t` label more than one transition in the automaton. It is quite conceivable that one would like to assign different unparsing programs to the different `t`-transitions, or have different `final` unparsing programs, depending on which final state is reached. Unfortunately the language currently does not allow this.

This is a notational difficulty and not an inherent deficiency in the model. It is possible, for example, to design a class description editor that would allow the user to specify class structure with regular expressions or by building finite automata explicitly or a combination of both. By request the user can then view the definition in either form and can assign unparsing programs by indicating specific transitions or by using the transition labels as is currently the case. Although this difficulty is an inconvenience, it does not limit the unparsing schemes that the user can specify (even if the above interface is not supplied). The user can simulate the effect by using variables to record information that otherwise would have been implied by the transition.

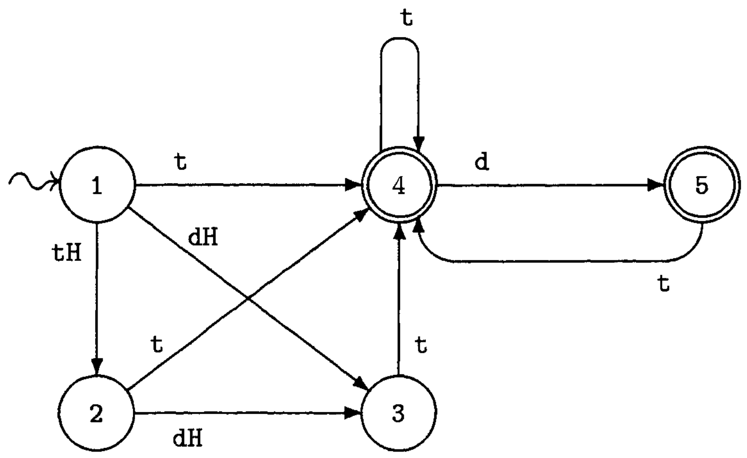


Figure 3.9: *DFA accepting definition list*

In summary: our class description language is sufficient to describe the syntax of any context-free document class. It provides additional descriptive power through the use of attributes, which includes references between objects. Empirical evidence suggests the semantic descriptions are sufficient to describe the input to many batch document formatters, provided that the batch input is modelled correctly by the syntax description. Finally we note that the mechanism used to associate semantics with syntax needs to be enhanced if full use is to be made of the power of the description language.

Chapter 4

Theoretical Foundations

In this chapter we discuss some of the theory and algorithms that are necessary for the construction of a prototype system based on the framework outlined in Chapter 2 and using the class description language described in Chapter 3. In particular, we address the issues of dynamic menu construction and the handling of incomplete documents. On first reading the reader might want to skim through this chapter and return to it after having read Chapter 5.

4.1 Menus and Their Construction

As will be seen in in Chapter 5, menus form a central part of the user interface of the prototype system.

There are two general kinds of menus in the system: static menus and dynamic menus. As their name implies, static menus remain the same throughout an editing session. Dynamic menus on the other hand, change depending on the current state of the editing session.

The create menu is a dynamic menu. This menu is usually displayed at the bottom of the screen while the document structure is being edited. It indicates which document objects, if any, can be legally inserted after the current object. Obviously, as the cursor is moved around the document, or as new objects are added to the document, the create menu will change.

The create menu is calculated dynamically from the finite automaton constructed for the object class of the parent of the current object:

Suppose a correct object X of class x currently consists of a sequence of objects $a_1...a_m$, and that the cursor is currently on a_k, $0 \leq k \leq m$ ($k = 0$ implies the cursor is on a phantom first child, in which case $a_1...a_k$ is the null sequence). We wish to construct a menu containing the names of all classes of objects that may be inserted after a_k without violating the document class syntax.

Let $\delta : (S, \Sigma) \mapsto S$ be the transition function of the finite automaton[1] that accepts all members of object class x, where S is the set of states and Σ the machine's input alphabet. Assume that the automaton is in state s_k after having read symbol a_k. The set of objects that should appear in the create menu is then

$$\{\ b \mid \delta(s_k, ba_{k+1}...a_m)\ \textit{is an accepting state}\ \}$$

Note that we have extended δ in the usual way (see for example [Hopcr79]) by writing $\delta(s, a_1...a_m)$ for $\delta(\delta(...\delta(\delta(s, a_1), a_2)..., a_{m-1}), a_m)$. This set can be calculated in a straightforward manner by executing the automaton with $a_1...a_kba_{k+1}...a_m$ as input string, using different values for b. The only values for b that have to be considered are those that label transitions from s_k. We will assume that the function *CONSTRUCT_MENU(x, X, k)*, whose parameters correspond to the symbols that we have used, exists and that it returns the create menu set as just described (i.e. the set of acceptable b's).

Example

Let x be the class defined by the DFA of Figure 3.9 on page 53, and let

$X = tH\ t$

[1] We have assumed here that the automaton is a deterministic one. This does not have to be the case, however, and a similar approach can be taken for a non-deterministic automaton.

with tH the current object (i.e. $k = 1$, and $s_k = 2$). *CONSTRUCT_MENU* will return the set $\{dH,\ t\}$, since both $\delta(2,$ *"dH t"*$)$ and $\delta(2,$ *"t t"*$)$ are final (accepting) states. Only dH and t will be considered since no other transitions are possible from state 2.

Note that if the current state of the automaton is stored with each symbol that is part of the document (e.g. s_k is stored with a_k), it is not necessary to re-start the automaton from its start state every time, but it can be started in state s_k with input $ba_{k+1}...a_m$. Nor is it always necessary to scan all of $ba_{k+1}...a_m$ for each b, since for a given b, scanning can be terminated as soon as an a_j has been read for which the corresponding stored state is the same as the current scan state. It is of course necessary that the state information saved with the document is updated whenever the document structure changes (e.g. when an object is added or deleted).

4.2 Partial Documents

As was pointed out in Section 2.4.2, the system must be able to accept partially complete (or incomplete) documents. There are at least two ways in which this goal may be achieved. The two methods differ with respect to the degree of incompleteness that is allowed in documents.

Tail incompleteness allows only the "tail" or "right end" of an object to be incomplete. This can be accomplished by regarding all the states in the automaton accepting a legal composite object to be final states. This approach allows for a front-to-back creation process, but leaves unresolved issues pertaining to the editing of a correct document. For example, suppose an object class x is defined as

```
x = (a|b) c*
```

and an instance of the class was created with composition

X_i = a c c c

If the user should now decide that a must be replaced with b, there is no easy and convenient way of accomplishing this while still allowing only tail incompleteness. One either has to define elaborate transformation rules, or use the mechanism of moving portions of the document to a non-syntax checking environment for editing and later insertion in the syntax checking environment.

Sub-sequence incompleteness on the other hand, allows an object to be accepted if it is composed of any sub-sequence of a legal composition, where a sub-sequence is defined as in [Aho74, page 361], namely

> A string $x = a_1a_2...a_n$ is a subsequence of string $y = b_1b_2...b_p$, if x is y with zero or more symbols deleted.

This approach allows the user to create sibling document components in any time order. Sub-sequence incompleteness also allows for much more flexibility when an existing document is edited. For example, the problematic case discussed under tail incompleteness can be handled in a natural way if sub-sequence incompleteness is allowed. In fact, tail incomplete documents form a subset of sub-sequence incomplete documents.

The prototype system described in Chapter 5 supports sub-sequence incomplete documents by modifying (at appropriate times) the finite state automaton constructed for each object class. The automata are modified by adding null (ϵ) transitions: If $\delta : (S, \Sigma) \mapsto S$ is the transition function for such an automaton, with S the set of states and Σ the input alphabet, then for every $r, s \in S$ and $a \in \Sigma$ such that $\delta(r, a) = s$, the transition $\delta(r, \epsilon) = s$ is added to the automaton. This is equivalent to changing the class grammar to one in which every symbol is an optional one.

Unfortunately supporting sub-sequence incompleteness introduces a potential complication for the user. It is conceivable that a user might create a document that is sub-sequence incomplete and then not know how to complete it. This is because the create menu contains the names of all classes that may be inserted at the current point, and not just those that will complete the document with the least number of insertions. In fact some items in the create menu might cause the document to be even more incomplete. One way to partially overcome this problem is for the system to mark those items in the menu that will complete the parent of the current object in the least number of insertions. If there are any required objects, at least one of them will always be marked.

For a given object class x, we use *Mx* to denote the original automaton, and Mx_ϵ to denote the one with the additional ϵ-transitions. Mx_ϵ is used to unparse (format) an object of class x and to construct a create menu as described in Section 4.1. *Mx* is used to check for document completeness (described in the next section) and to mark those items in the create menu that are necessary to complete the parent of the current object with the least possible number of insertions (see Section 4.2.2).

4.2.1 Checking for Document Correctness

A composite object is checked for correctness by executing the finite state automaton for that object, using the object's children as input symbols. If all the input symbols have been "read" and the automaton is in an accepting (or final) state, the object is correct and complete *as defined by that automaton.* In this way Mx_ϵ can be used to "accept" a sub-sequence incomplete object, while *Mx* can be used to check that the object is indeed complete before output is generated for a target formatter.

We will assume that the function *TEST_CORRECT(x, X)* exists that tests whether the sequence of symbols $X = a_1 ... a_m$ is accepted by the automaton for objects of class x. The function returns *TRUE* if the sequence is accepted, and *FALSE* if the automaton fails.

4.2.2 Marking Menu Items

The user must have some indication of how to complete a sub-sequence incomplete object using a minimal number of insertions. The system therefore marks these items in the menu. Note that once a user selects any item in the menu, the menu as well as the markings may change completely.

There are three steps in the marking process:

1. Construct a finite automaton (we will call it MX_{min}) that provides all possible legal ways of completing the incomplete object.

2. Determine the shortest path(s) from the initial to final state(s) in MX_{min}.

3. Use the set of shortest paths calculated in step 2 to mark the appropriate items in the create menu.

The details of these steps are given next.

Let $X_p = a_1 a_2 ... a_m$ be a sub-sequence incomplete object of class x, and let Σ_x be the set of all symbols in the regular expression defining the class x. Let MX_p be a finite automaton accepting the language

$$\Sigma_x^* a_1 \Sigma_x^* a_2 \Sigma_x^* ... \Sigma_x^* a_m \Sigma_x^*$$

where Σ_x^* is the Kleene closure of Σ_x. Construct MX_{min} to accept the intersection of the languages accepted by MX_p and Mx and find R, the set of shortest paths from the start to any final state of MX_{min} (Section 4.2.3 describes an algorithm that constructs MX_{min} and R simultaneously). All elements (paths) in R have the same length $|r|$, and each one defines one way of completing X_p using the least number of insertions. R is used in the marking algorithm we describe next.

Let *Menu* be the set of menu items (symbols) determined as described in Section 4.1. If a_c is the current object, i.e. the one after which a new object is to be inserted ($c = 0$ indicates the new object must be inserted in front of a_1, the first object), then the algorithm of Figure 4.1 can be used to mark the appropriate items in *Menu*. The functions *LEFTMOST_MATCH* and *RIGHTMOST_MATCH* are given in Figures 4.2 and 4.3.

In order to prove *MARK* correct, we need two lemmas:

Lemma 1 *LEFTMOST_MATCH produces the output as specified in at most $O(|r|)$ time if its input is as specified.*

Proof: If $c = 0$, the test of line 2 fails upon loop entry so that lines 3–5 are not executed and line 6 returns 0, which satisfies the output conditions.

For $1 \leq c \leq |a|$ the loop of line 2 is executed until $i > c$. i and j both start at 1, with j being incremented by 1 in every iteration. This ensures that a_i is compared with a

```
MARK( $X_p$, c, Menu, R )
INPUT:
    $X_p$     $= a_1 ... a_m$ a sub-sequence incomplete object of class $x$.
    $c$       - the index (in $X_p$) of the current object.
    Menu  - the set of names of objects that may be inserted after $a_c$.
    $R$       - the set of shortest paths from the start to any final state of $MX_{min}$.
OUTPUT:
    Menu, with those symbols marked that will allow $X_p$ to be completed in the
    least number of insertions.
begin
1.     $n := |r|$, the length of any element of $R$
2.     if( $|X_p| \geq n$ )then return Menu with no items marked
3.     for all $r = r_1 r_2 ... r_{|r|} \in R$ do
4.         $j$ := LEFTMOST_MATCH( $X_p$, c, r )
5.         $k$ := RIGHTMOST_MATCH( $X_p$, c+1, r )
6.         $i := j + 1$
7.         while( $i < k$ )do
8.             mark $r_j$ in Menu
9.             $i := i + 1$
           endwhile
       endfor
end
```

Figure 4.1: *The marking algorithm*

```
    LEFTMOST_MATCH( a, c, r )
    INPUT:
        strings $a$ and $r$, with $a$ a sub-sequence of $r$.
        $0 \leq c \leq |a|$
    OUTPUT:
        the smallest $j \geq 0$ such that $a_1 ... a_c$ is a sub-sequence of $r_1 ... r_j$
    begin
1.      $i := j := 1$
2.      while( $i \leq c$ )do
3.          if( $a_i = r_j$ )then
4.              $i := i + 1$
            endif
5.          $j := j + 1$
        endwhile
6.      return $j - 1$
    end
```

Figure 4.2: *LEFTMOST_MATCH*

```
    RIGHTMOST_MATCH( a, c, r )
    INPUT:
        strings $a$ and $r$, with $a$ a sub-sequence of $r$.
        $0 \leq c \leq |a|$
    OUTPUT:
        the largest $j \leq |r| + 1$ such that $a_c ... a_{|a|}$ is a sub-sequence of $r_j ... r_{|r|}$
    begin
1.      $i := |a|$
2.      $j := |r|$
3.      while( $i \geq c$ )do
4.          if( $a_i = r_j$ )then
5.              $i := i - 1$
            endif
6.          $j := j - 1$
        endwhile
7.      return $j + 1$
    end
```

Figure 4.3: *RIGHTMOST_MATCH*

different r_j (in line 3) in each iteration. Since i is incremented only if $a_i = r_j$, and since a is a sub-sequence of r (in other words every a_i is equal to at least one unique r_j), the loop of line 2 will terminate after at most $|r|$ iterations with $i = c + 1$, $a_c = r_{j-1}$ and $a_1 \ldots a_c$ a sub-sequence of $r_1 \ldots r_{j-1}$. Line 15 returns $j - 1$, which proves the lemma.

Lemma 2 *RIGHTMOST_MATCH produces the output as specified in at most $O(|r|)$ time if its input is as specified.*

Proof. Similar to Lemma 1.

We can now prove that *MARK* is correct: If X_p is complete, it will be as long as any element in R and no items have to be marked, in which case line 1 returns the correct answer.

Since each element of R defines one way of completing X_p using the least number of insertions, and since the loop of line 3 operates on each element of R, we only have to show that lines 4–9 mark all the required items defined by a given $r \in R$. In other words, we have to show that all symbols (in a given r) that can be inserted between a_c and a_{c+1} such that the new $a_1 \ldots a_{|a|}$ is still a sub-sequence of r, are indeed marked by lines 4–9.

By Lemma 1, j of line 4 is the smallest j such that $a_1 \ldots a_c$ is a sub-sequence of $r_1 \ldots r_j$. Similarly, by Lemma 2, k of line 5 is the largest k such that $a_{c+1} \ldots a_{|a|}$ is a sub-sequence of $r_k \ldots r_{|r|}$. There are therefore no symbols in r other than $r_{j+1} \ldots r_{k-1}$ that can be inserted between a_c and a_{c+1} while still satisfying the condition that $a_1 \ldots a_c$ and $a_{c+1} \ldots a_{|a|}$ are sub-sequences of the remaining left and right portions of r respectively. Lines 6–9 mark exactly these symbols in *Menu*. We therefore conclude that the marking algorithm is correct.

If a straight linear search is used in line 8, the worst case time complexity of the marking algorithm is $O(|R||r||Menu|)$.

4.2.3 Constructing MX_{min} and R

The set R of shortest paths from the start to any final state of MX_{min}, can be constructed at the same time that MX_{min} is being constructed. In fact, it might not be necessary to construct the complete MX_{min} as it is sufficient to find only the final state nearest to the start state in order to completely construct R. The construction algorithm for R is as follows:

1. Form the state $s_1' = [s_1 s_2]$, where s_1 is the start state of Mx and s_2 the start state of MX_p. Set $level(s_1')$ to 0 and add it to a (hereto empty) list of states L.

2. Start at the front of the list L, and for every state $s' = [s_1 s_2]$ in the list do the following:

   (a) For every symbol $a \in \Sigma_x$, form the state $q = [\delta_1(s_1, a)\delta_2(s_2, a)]$, where δ_1 and δ_2 are the transition functions of Mx and MX_p respectively. If q is well defined and is not in the list L, set $level(q)$ to $level(s') + 1$ and append q to the end of L.

   Note: The states that are created are states in MX_{min} and the level of such a state indicates the shortest distance that it is from MX_{min}'s start state.

   (b) For every new state s' added to L, keep a "from"-list of state-symbol pairs. A pair (q, b) is added to the list for s' only if s' can be reached from state q with a single transition labeled b, and s' has a higher level than q.

   (c) Stop scanning and adding new states to L as soon as the first level-group that contains at least one final state is completed. A final state has the form $[f_1 f_2]$, where f_1 is the final state of Mx and f_2 is the final state of MX_p.

3. The shortest paths that we are looking for are defined (in reverse order) by following the "from"-links starting at each final state in L.

This algorithm constructs the states of MX_{min} in a breadth-first order. This follows from the fact that the standard breadth-first traversal technique is used in step 2(a), namely keeping a queue of states (nodes) still to be "visited" and "visiting" them in the order they were added to the queue [Stand80]. That states added to this queue are indeed states in MX_{min} follows directly from the definition of an "intersection machine" [Hopcr79] and the state construction of step 2(a).

The level number assigned to each state in step 2(a) is the state's distance from the start state since the start state is assigned a level of 0, every other state is assigned a level one greater than that of its predecessor, and the states are visited (created) once only and in breadth-first order.

Since we are only interested in the shortest path(s) from the start to the nearest final state(s), the algorithm terminates correctly in step 2(c) once all states at the same level of the first final state have been created. It is necessary that all states at that level be created to ensure that all shortest paths are found; for example more than one of them may be a final state. By nature of their construction, the "from"-links define all these shortest paths.

Steps 2(a) and 2(b) and the test in step 2(c) are executed at most $|Q_1 \times Q_2| =$ *maximum number of states in* MX_{min} times. In step 2(a) every symbol in Σ_x has to be examined once for a cost of $O(|\Sigma_x|)$. Determining that q is not already in L has a worst-case cost of $|Q_1 \times Q_2|$ assuming a simple linear search is used. Adding a "from"-link to a state can be done in constant time so that the total cost of step 2 is $O((|\Sigma_x| + |Q_1 \times Q_2|)|Q_1 \times Q_2|)$. The set of shortest paths can be obtained in step 3 in $O(|r||R|)$ time, which gives the complete algorithm a time complexity of $O((|\Sigma_x| + |Q_1 \times Q_2|)|Q_1 \times Q_2| + |r||R|)$. While this might seem excessive because of the $|Q_1 \times Q_2|^2$ factor, it should be pointed out that on average the intersection machine has less than $|Q_1 \times Q_2|$ states and furthermore, the algorithm might not have to create all of the states in the machine. In addition it seems that the automata for objects in most practical document models tend to have only a few states.

Example

Let

$$x = a?((a|b)c(a|b)?)*$$

and

$$X_p = aaba$$

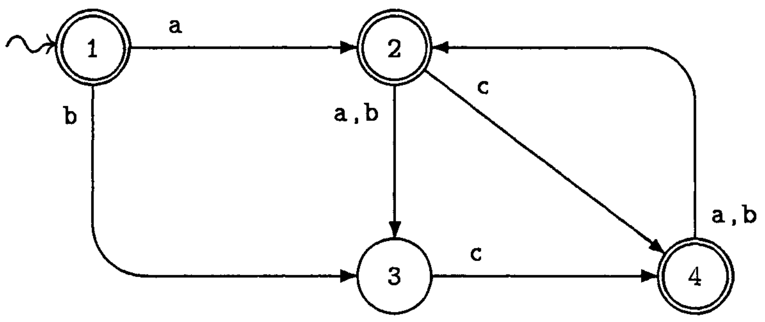


Figure 4.4: *The minimised finite automaton Mx that accepts $a?((a|b)c(a|b)?)*$*

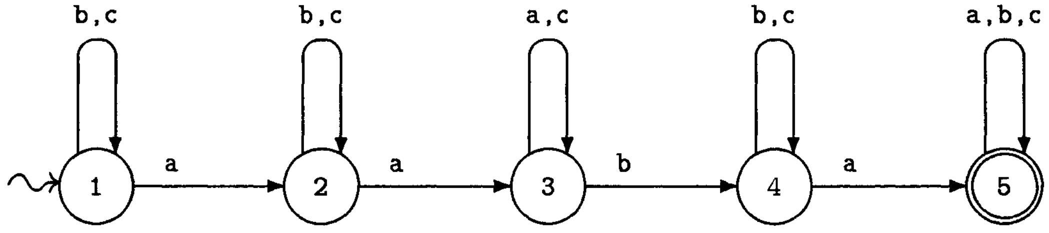


Figure 4.5: *The minimised finite automaton MX_p*

Mx is given by Figure 4.4 and MX_p by Figure 4.5. Using the method outlined in Section 4.1, it is straightforward to determine that the create menus for each of the five insert positions in X_p each contain the items a, b and c. Table 4.1 illustrates the construction of MX_{min} and R. The first two columns show the state list L, while the last column shows the from-list for each state. The other three columns show the transition functions for MX_{min}. Note that Table 4.1 does not define the complete MX_{min}, but only those states and transitions used to construct R.

Figure 4.6 shows the "shortest path machine" constructed from the from-lists. In this DFA each path from the start state to a final state defines one of the paths in R. In other words, there are three ways of completing X_p with only two insertions, namely *aacbac*, *aacbca*, and *acabca*. If we perform the marking algorithm for each of the five insert positions,

Table 4.1: *States and Transitions in MX_{min}*

		transition on			
Level	State	a	b	c	from-list
0	[1 1]	[2 2]	[3 1]	-	-
1	[2 2]	[3 3]	[3 2]	[4 2]	([1 1],a)
	[3 1]	-	-	[4 1]	([1 1],b)
2	[3 3]	-	-	[4 3]	([2 2],a)
	[3 2]	-	-	[4 2]	([2 2],b)
	[4 2]	[2 3]	[2 2]	-	([2 2],c)
	[4 1]	[2 2]	[2 1]	-	([3 1],c)
3	[4 3]	[2 3]	[2 4]	-	([3 3],c)
	[2 3]	[3 3]	[3 4]	[4 3]	([4 2],a)
	[2 1]	[3 2]	[3 1]	[4 1]	([4 1],b)
4	[2 4]	[3 5]	[3 4]	[4 4]	([4 3],b)
	[3 4]	-	-	[4 4]	([2 3],b)
5	[3 5]	-	-	[4 5]	([2 4],a)
	[4 4]	[2 5]	[2 4]	-	([2 4],c) ([3 4],c)
6	[4 5]	[2 5]	[2 5]	-	([3 5],c)
	[2 5]	[3 5]	[3 5]	[4 5]	([4 4],a)

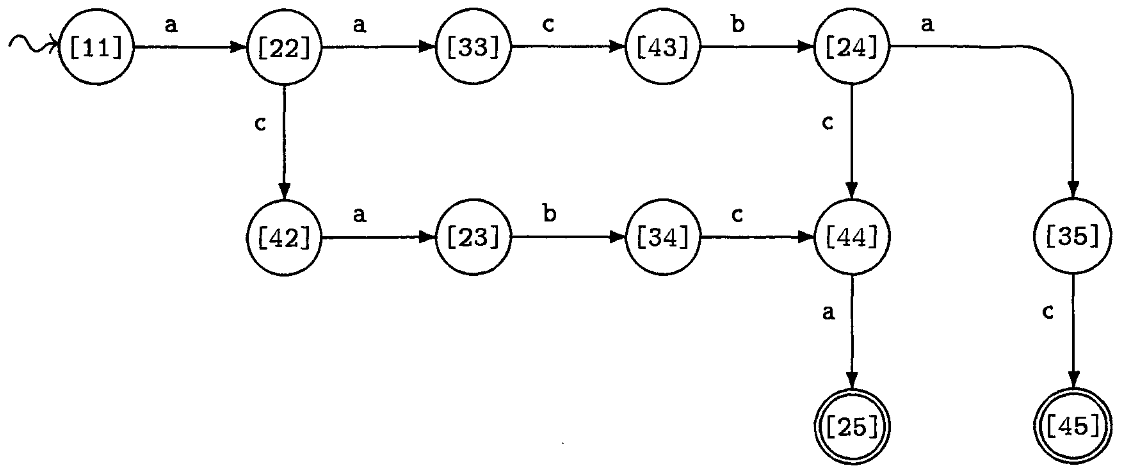


Figure 4.6: *DFA accepting all shortest paths from start to final states in* $\mathcal{M}X_{min}$

we obtain the following:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \mathrm{a} \begin{bmatrix} a \\ b \\ \underline{\mathrm{c}} \end{bmatrix} \mathrm{a} \begin{bmatrix} a \\ b \\ \underline{\mathrm{c}} \end{bmatrix} \mathrm{b} \begin{bmatrix} a \\ b \\ \underline{\mathrm{c}} \end{bmatrix} \mathrm{a} \begin{bmatrix} a \\ b \\ \underline{\mathrm{c}} \end{bmatrix}$$

The menu at each insert position is shown between square brackets ([]), while marked menu items are underlined. Note that we have constructed all five menus merely as an illustration. The system constructs only the menu for the current position.

Chapter 5

A Prototype

This chapter describes the design and implementation of a prototype system built within the framework presented earlier[1].

We provide a syntax-controlled structured document editor that uses the information given in document class descriptions to guide the user in creating documents that conform to the particular document classes. We use the term syntax-controlled simply to distinguish it from the normal syntax-directed programming language editors that can handle only one set of syntax rules. Our system can switch dynamically between different sets of syntax rules (class descriptions).

The system uses the dynamic menu construction algorithm described in the previous chapter to present the user with menus that only allow the creation of correct documents (in the sub-sequence incomplete sense). The system is also capable of producing different output files for the same document, each output file being suitable as input for a different target batch formatting system.

The driving engine of the system is a set of finite automata; the automata accept the languages defined by the regular expressions that in turn define the object classes in the class description. A finite automaton is built for each object class using a construction

[1]This dissertation, excluding the figures and tables, was written using an early prototype of the system.

algorithm based on the one given in [Aho86]. These automata are used to determine the objects which can be added to the document, as well as the locations where the objects can be added. The automata are also used to drive the unparsing (or formatting) process that produces the output file(s).

The automata that are built are "minimised" non-deterministic finite automata (NFAs). The automata are minimised in the sense that they do not contain any epsilon (null) transitions, equivalent states are combined, and unreachable states are removed.

5.1 The User Interface

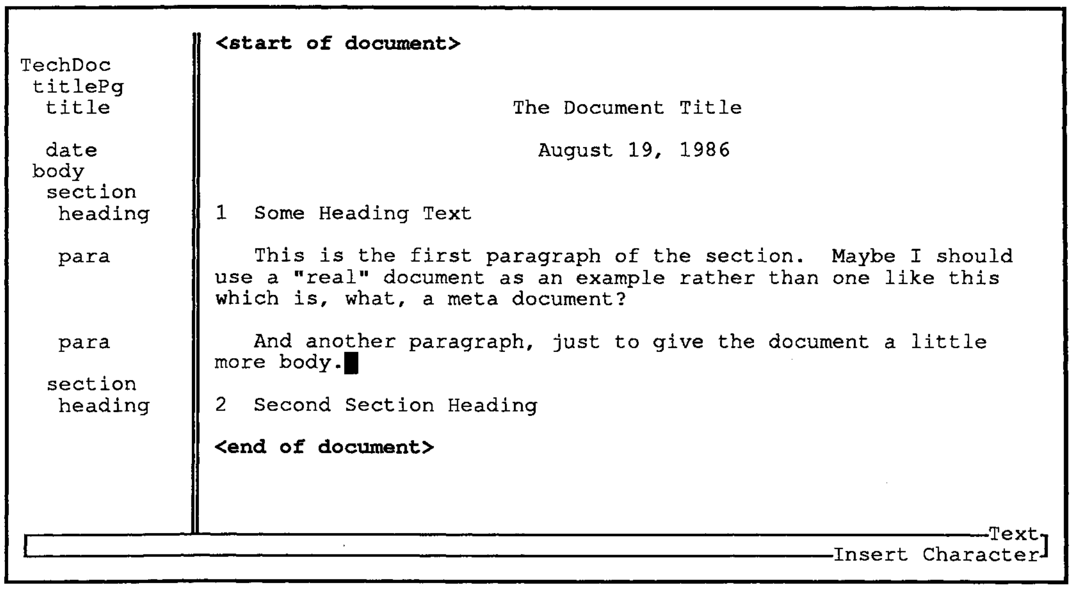
In order to attain the system goals that have been outlined in Chapter 1 the following design decisions were made:

1. It must be possible to present a document to the user in a form that is reasonably close to the final form that the user wants.
2. Not only the document content should be presented to the user, but also the document structure.
3. The user must be allowed to edit the document structure explicitly.

The approach that has been taken in the prototype is one that could be termed an annotated what-you-see-is-almost-what-you-get (WYSI-A-WYG) approach. A WYSI-A-WYG display of the document is annotated with labels that indicate the structural aspects of the document. For this purpose the display screen is divided into two parallel windows. The formatted document content is displayed in a text window and the structure labels in a structure window as illustrated in Figure 5.1[2]. A third window at the bottom of the screen is used for status and menu display.

Both the structure and text windows are used to edit the document. The structure window is used to edit the document structure. Structure editing refers to the editing

[2]Unless otherwise stated, all screen examples are based on the class description in Appendix C.1.



```
               <start of document>
TechDoc
 titlePg
  title                          The Document Title

  date                             August 19, 1986
 body
  section
   heading     1  Some Heading Text

   para           This is the first paragraph of the section.  Maybe I should
               use a "real" document as an example rather than one like this
               which is, what, a meta document?

   para           And another paragraph, just to give the document a little
               more body.█
  section
   heading     2  Second Section Heading

               <end of document>

                                                                         Text
                                                             Insert Character
```

Figure 5.1: *Annotated WYSI-A-WYG display showing document structure and content*

The labels in the structure window on the left are the class names of the objects in the document. The indentation of the labels indicates the hierarchical levels.

The text window on the right shows the document content, formatted to have an appearance similar to the final document.

The (empty) window at the bottom is a menu window that may contain a create menu or other command menus at appropriate times. Its upper right corner indicates the class name of the current object (the smallest object containing the cursor), and the lower right corner indicates the current edit mode.

The block of reverse video at the end of the second paragraph is the cursor. Here the size of the cursor is a single character. The cursor always indicates the scope of editing operations, and its size is therefore adjustable by the user.

operations that add, delete, copy, or move objects (instances of object classes) to, from, or in the document, consistent with the document class description. For example, to add a third paragraph to the first section of the document in Figure 5.1, we have to edit the structure of the section, giving it a fourth child. Structural editing only influences the content of the document in so far that the objects being created, deleted, copied, or moved, "take their content with them".

The text window is used to edit the document content. The content editing operations do not influence the structure of the document.

5.1.1 Structure Editing

While the user is editing the document structure in the structure window, the cursor always covers the current object and its constituents, as illustrated in Figure 5.2. This ensures that the user can clearly see which part of the document is involved in the edit.

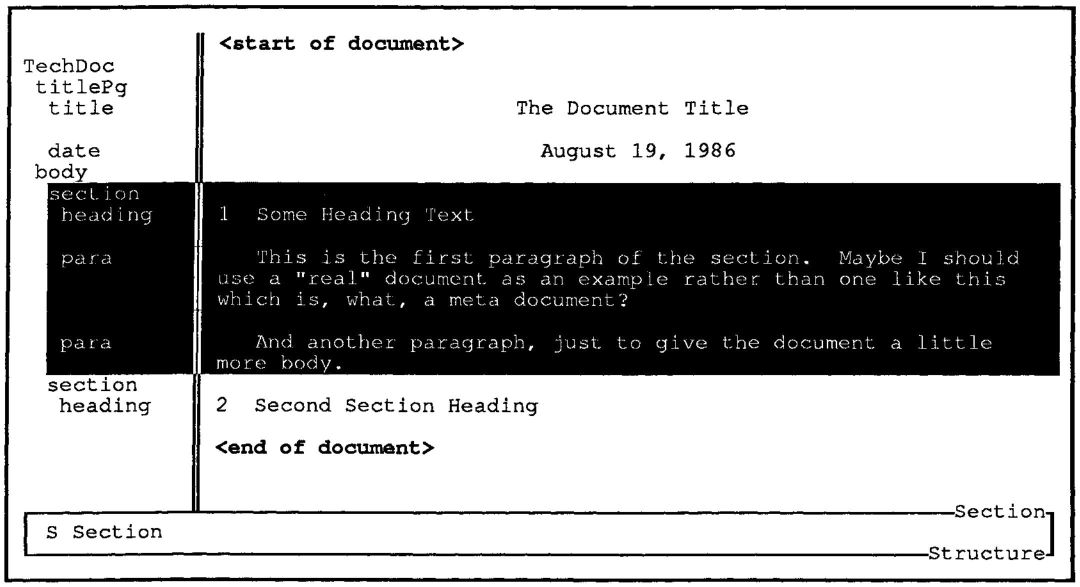
During structural editing, the system ensures that the document's structure conforms to the structure of its class as described in the class description. In other words, the system ensures that the document is a valid instance of its class. This is mostly done by not providing the user with an opportunity to create an incorrect document. At any point in time the system presents the user only with a legal set of commands.

Figure 5.3 and Figure 5.4 illustrate how objects are added to a document. In Figure 5.3 the structure cursor is positioned[3] on the `title` object inside the title page, and the create menu indicates that an object author can be created and inserted after the current object by pressing the U key.

The way in which an item is displayed in the menu indicates whether it is a required item or an optional one: required items are displayed in bold typeface while optional ones (like author) are displayed in normal typeface.

Figure 5.4 shows the result after the user has typed a U. The **author** syntax requires a

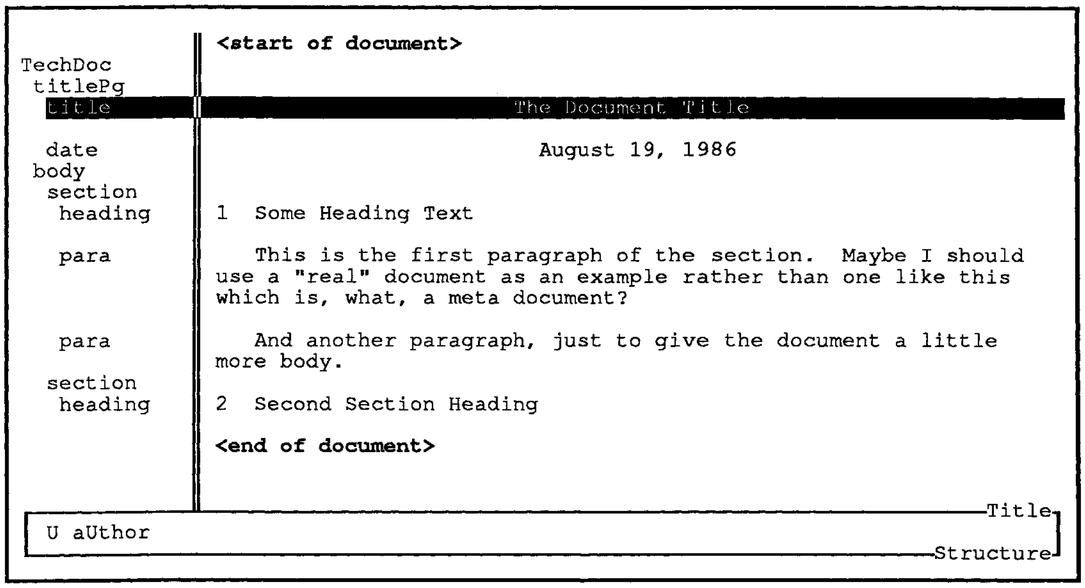
[3]The complete list of commands (such as cursor movement commands) that are provided by the system, is given in Appendix E.



```
               <start of document>
TechDoc
 titlePg
  title                         The Document Title

  date                            August 19, 1986
 body
  section
   heading      1  Some Heading Text

   para            This is the first paragraph of the section.  Maybe I should
                use a "real" document as an example rather than one like this
                which is, what, a meta document?

   para            And another paragraph, just to give the document a little
                more body.
  section
   heading      2  Second Section Heading

                <end of document>

 S Section                                                              Section
                                                                      Structure
```

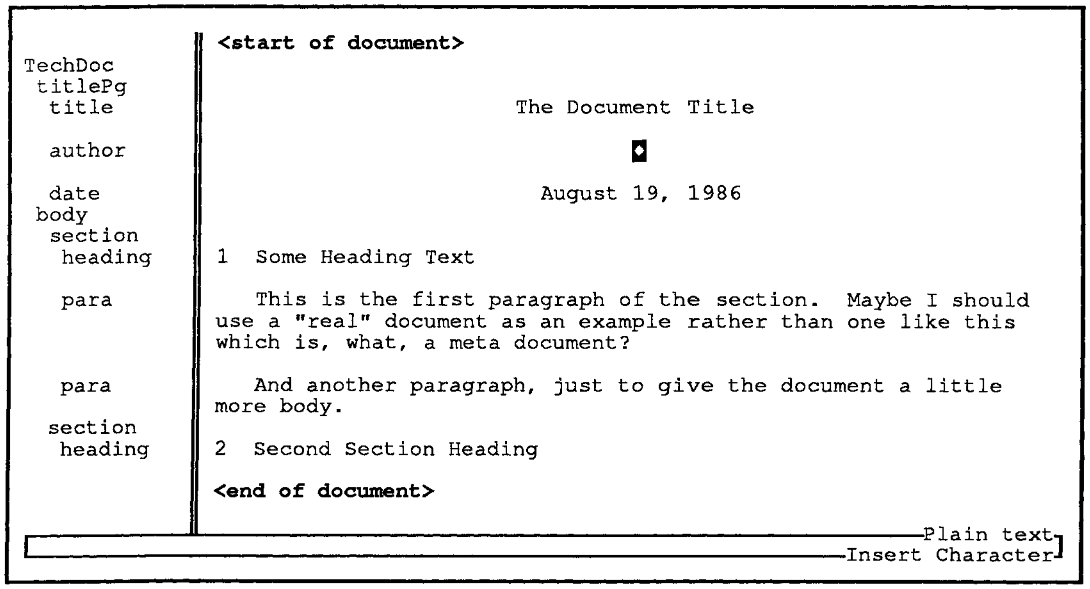
Figure 5.2: *The cursor during structure editing*

When editing the document structure, the cursor always covers the current object (and its constituents). The menu window contains a list of objects that may be inserted after the current object.



```
                 <start of document>
TechDoc
 titlePg
  title                          The Document Title

  date                          August 19, 1986
 body
  section
   heading       1  Some Heading Text

   para             This is the first paragraph of the section.  Maybe I should
                 use a "real" document as an example rather than one like this
                 which is, what, a meta document?

   para             And another paragraph, just to give the document a little
                 more body.
  section
   heading       2  Second Section Heading

                 <end of document>

                                                                          Title
  U aUthor
                                                                      Structure
```

Figure 5.3: *Ready to add an* author *object by typing* U



```
                 <start of document>
TechDoc
 titlePg
  title                         The Document Title

  author                                ◘

  date                          August 19, 1986
 body
  section
   heading       1  Some Heading Text

   para             This is the first paragraph of the section.  Maybe I should
                 use a "real" document as an example rather than one like this
                 which is, what, a meta document?

   para             And another paragraph, just to give the document a little
                 more body.
  section
   heading       2  Second Section Heading

                 <end of document>
                                                                     Plain text
                                                               Insert Character
```

Figure 5.4: *The* author *object has been added.* The user may now enter text.

single child of class PLAIN_TEXT which is automatically created by the system. Furthermore, since PLAIN_TEXT is a basic object (and therefore has user-supplied text associated with it), the system automatically goes into text edit-mode in anticipation of the user's text entry. The diamond that is displayed is the null string placeholder and is displayed when the text associated with a basic object is the null string. It is replaced by whatever text is entered by the user.

Note that the "current object", as indicated by the name in the top right corner of the menu window, is not author but its first child, a PLAIN_TEXT object. By design, basic objects are not annotated with their class names in the structure window.

The preceding example shows how new objects are only added after the current object. The use of *phantom first children* allows this approach to be valid even for adding an object as an only child or inserting it at the start of a list of objects. If the class syntax and current composition of a composite object permits new objects to be inserted in front of the first child of the composite object, the system conceptually inserts a phantom child at that position. A phantom child is classless and is only visible if the cursor is positioned on it. It acts as a placeholder for objects that may be inserted at that position. We will show two examples of its use.

After the user has entered the author's name in Figure 5.4, and issued the "next" command to position the cursor on the date object, the screen will be as shown in Figure 5.5. Note that the command menu shows that an abstract can be inserted after the date in the title page by pressing the A key. Figure 5.6 shows the result after such an operation.

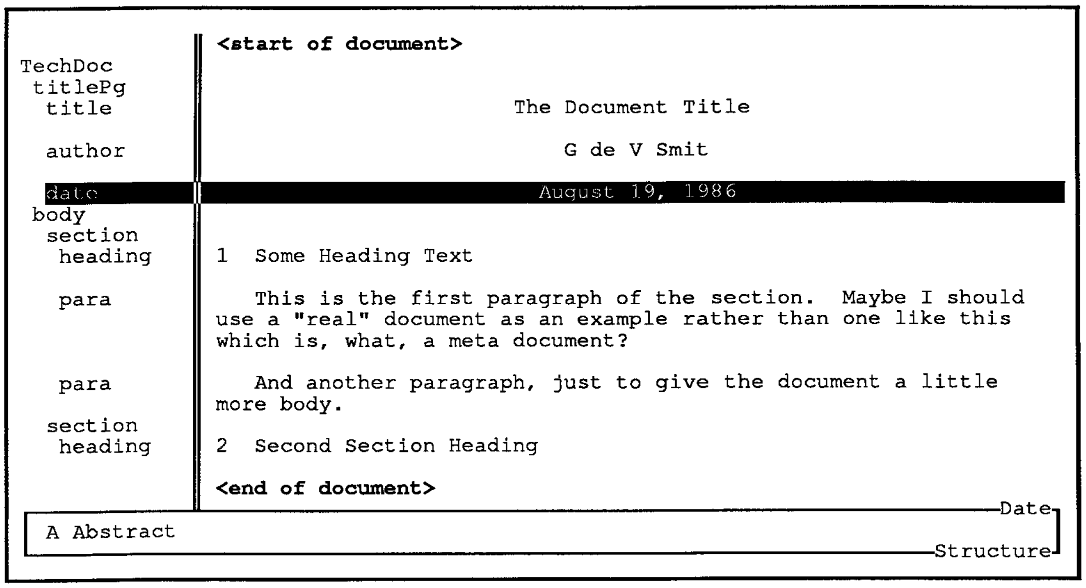
The syntax for abstract, defined as

```
    abstract = %basicObj+ : -footnote, -fnRef
```

which is equivalent to

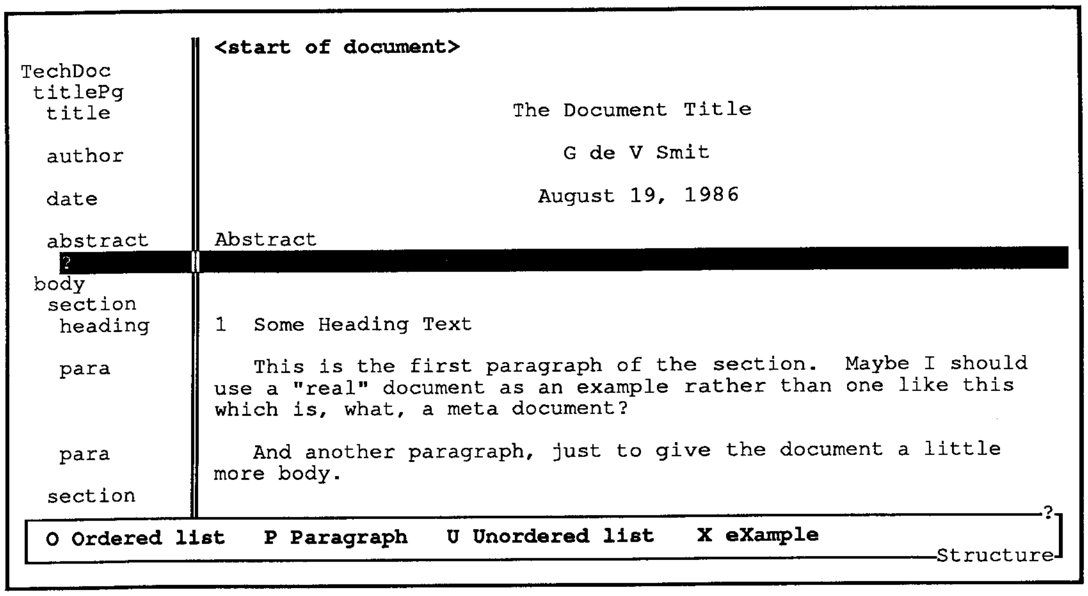
```
    abstract = (paragraph | ordList | unordList | example)+
```

requires that abstract has at least one child. The system therefore wants to create it automatically. However, since the system cannot determine to which class the child should



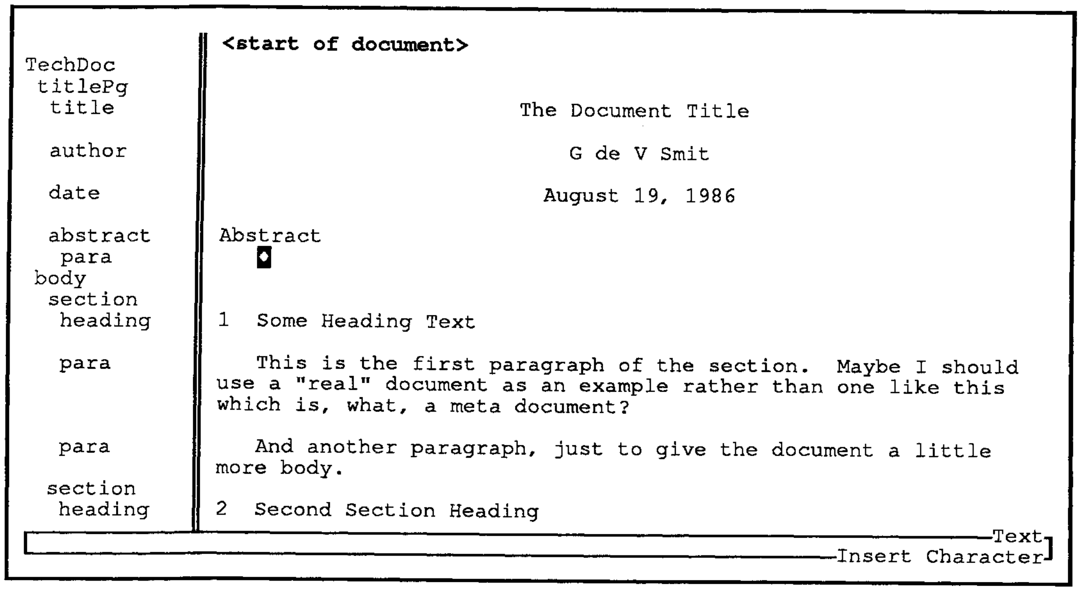
```
                  <start of document>
TechDoc
 titlePg
  title                          The Document Title

  author                             G de V Smit

  date                            August 19, 1986
 body
  section
   heading        1  Some Heading Text

   para              This is the first paragraph of the section.  Maybe I should
                  use a "real" document as an example rather than one like this
                  which is, what, a meta document?

   para              And another paragraph, just to give the document a little
                  more body.
  section
   heading        2  Second Section Heading

                  <end of document>
                                                                        Date
 A Abstract
                                                                   Structure
```

Figure 5.5: *Ready to add* abstract *after* date



```
                  <start of document>
TechDoc
 titlePg
  title                          The Document Title

  author                             G de V Smit

  date                            August 19, 1986

  abstract        Abstract
   ?
 body
  section
   heading        1  Some Heading Text

   para              This is the first paragraph of the section.  Maybe I should
                  use a "real" document as an example rather than one like this
                  which is, what, a meta document?

   para              And another paragraph, just to give the document a little
                  more body.

  section
                                                                           ?
 O Ordered list   P Paragraph   U Unordered list   X eXample
                                                                   Structure
```

Figure 5.6: *Adding the first child of an object*



```
                    <start of document>
TechDoc
 titlePg
  title                              The Document Title

  author                               G de V Smit

  date                               August 19, 1986

  abstract         Abstract
   para               ■
 body
  section
   heading         1  Some Heading Text

   para               This is the first paragraph of the section.  Maybe I should
                   use a "real" document as an example rather than one like this
                   which is, what, a meta document?

   para               And another paragraph, just to give the document a little
                   more body.

  section
   heading         2  Second Section Heading
                                                                         Text
                                                             Insert Character
```

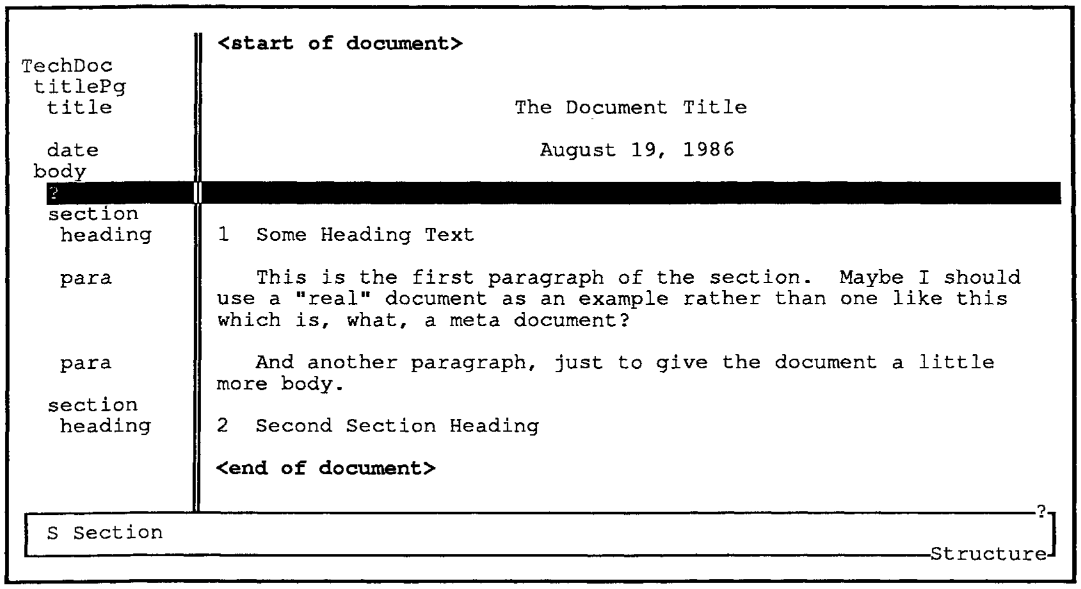
Figure 5.7: *The abstract has been added*

belong, it positions the cursor on the phantom first child of **abstract**, and provides the user with the opportunity to specify the class of the real first child. Once again the user only has to type the appropriate letter. Figure 5.7 shows the result after a P has been typed.

The fact that all the object names in the create menu of Figure 5.6 are displayed in bold, does not mean that they are all required objects, but rather that the required object can belong to any of the indicated classes.

A second example of the use of a phantom first child is shown in Figure 5.8, which shows the screen after the command "previous" has been executed in the situation depicted by Figure 5.2.

Structural editing operations other than the insertion of new objects (e.g. deletion, nesting, extraction, and changing objects' classes — cf. Appendix E.5) are usually performed by first marking sub-trees in the document tree and then specifying the operation to be performed. The operation is only executed if the result will be a correct document in the sub-sequence incomplete sense. The marking operation (at least during structure editing) is such that only complete sub-trees can be marked. For example it is possible to mark a



```
                 <start of document>
TechDoc
 titlePg
  title                            The Document Title

  date                               August 19, 1986
 body
  ?
  section
   heading       1  Some Heading Text

   para             This is the first paragraph of the section.  Maybe I should
                 use a "real" document as an example rather than one like this
                 which is, what, a meta document?

   para             And another paragraph, just to give the document a little
                 more body.
  section
   heading       2  Second Section Heading

                 <end of document>

 S Section                                                              ?
                                                                Structure
```

Figure 5.8: *The cursor positioned on a phantom first child*

This is the result of issuing the command “previous” in the situation depicted by Figure 5.2.

sequence of sibling sub-trees such as three consecutive sections in a chapter; but it is not possible to mark some children of a node together with a sibling of that node, such as the last few paragraphs of a section together with the section that follows it. Editing operations that require such marking have to be performed in separate steps.

The algorithms used by the structure editing commands of Appendix E.5 are given in Appendix F.

5.1.2 Content Editing

Content editing operations are performed in the text window on the right hand side of the screen and do not influence the structure of the document. All standard text editing operations are provided: insert, overwrite, search, replace, copy and move.

The cursor always serves as a text selector, selecting the text that will be deleted, copied or moved. The size of the cursor can be set to that of a character, a word, a sentence, a text block[4], or an arbitrary portion of text. Only text that the user has entered can be selected, and not any text generated by system. For example, the cursor cannot be positioned on the word `Abstract` just above the cursor in Figure 5.7. Issuing the cursor up command will move the cursor to the text on the `date` line.

Deletion of content does not change the document structure. The user may select a portion of text that spans more than one object and delete the text. This may result in one or more empty basic objects in the document, but the document structure would remain unchanged.

Text that is deleted is copied to a patch area that is described in the next section.

5.1.3 The Patch Area

The *patch area* is a special, non-restrictive editing environment that facilitates the moving and copying of portions of the document, as well as making drastic changes to a document. Anything that is deleted from the document currently being edited, is automatically copied

[4]A text block is the text forming the content of the current basic object.

to the patch area, from where the user can copy it again to any place in the document. The user can edit the patch area by issuing the "Patch" command.

There is no difference between editing the patch area and editing a normal document, except that the system uses a standard, non-restrictive class grammar for the patch area. The syntax for this grammar is

```
%Sigma = ob_1 | ob_2 | ob_3 ...  ob_n
patch = %Sigma*
```

where the $\mathtt{ob}_i$'s are all the object classes (basic and composite) that are defined in the current document class grammar. Furthermore, the syntax for each composite object class $\mathtt{ob}_c$ is re-written as

```
ob_c = %Sigma*
```

The result is that any combination and/or composition of any object is accepted in the patch area.

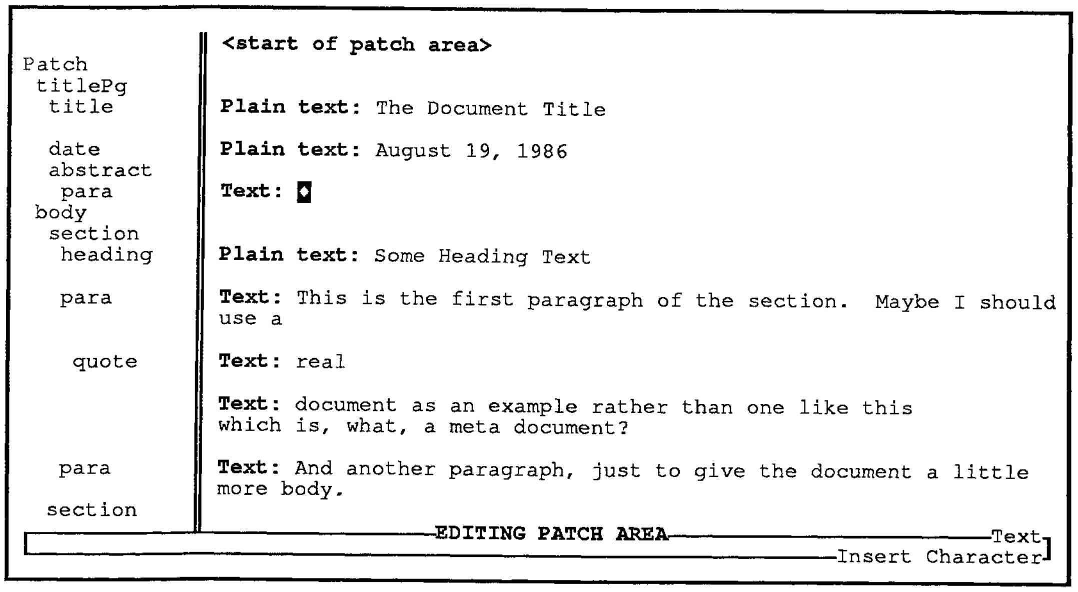
A standard, very simple unparsing scheme is used for the patch area. There is only one scheme, namely for the screen display, and each object class has only one method. Composite objects are unparsed with the method

```
Method Patch
  init:   @vs(2),
  Sigma:  @Patch(),
  final:
endMethod
```

and basic objects with

```
Method Patch
  init:   @vs(1) @bold "<class>:  " @ebold,
  final:
endMethod
```

where <*class*> is the class name of the basic object as it would appear in the create menu. Furthermore, all composite objects are labeled in the structure window (if its start label is



```
               <start of patch area>
Patch
 titlePg
  title        Plain text: The Document Title

  date         Plain text: August 19, 1986
  abstract
   para        Text: ▮
 body
  section
   heading     Plain text: Some Heading Text

   para        Text: This is the first paragraph of the section.  Maybe I should
               use a

    quote      Text: real

               Text: document as an example rather than one like this
               which is, what, a meta document?

   para        Text: And another paragraph, just to give the document a little
               more body.

  section
───────────────────EDITING PATCH AREA──────────────────────────────Text
───────────────────────────────────────────────────────Insert Character
```

Figure 5.9: *Editing the patch area*

null, the menu name is used – cf. the next section). Figure 5.9 shows the patch area if the complete document of Figure 5.7 were copied into it.

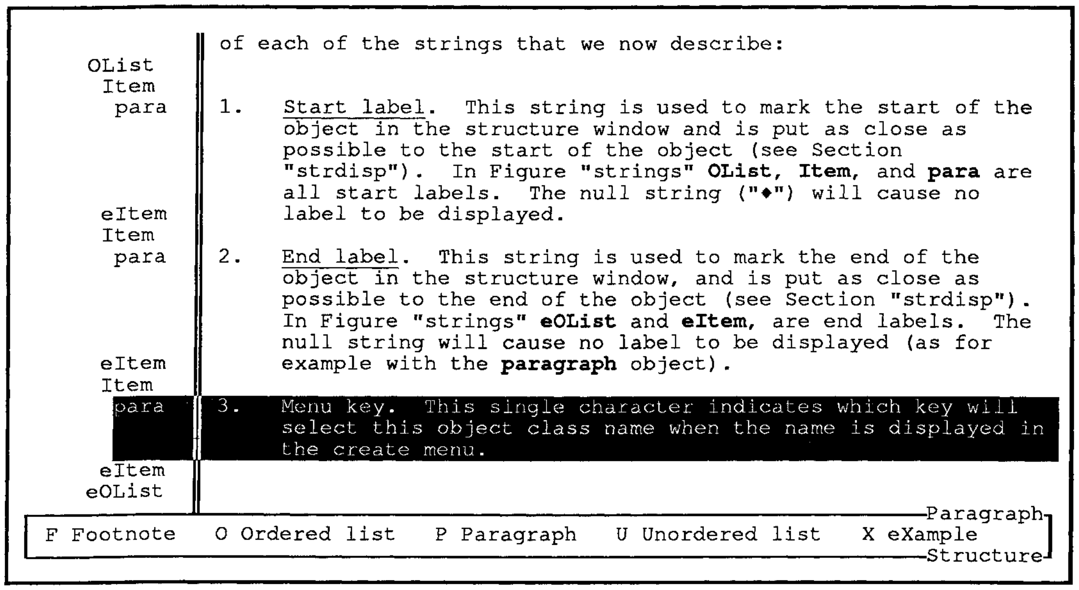
The contents of the patch area can be inserted in the document only if the result will be a correct document (in the sub-sequence incomplete sense of Section 4.2). An incorrect insert operation is rejected with a message indicating the reason for the rejection.

The patch area has its own patch area that behaves exactly like the normal patch area, except that the user cannot edit it.

5.1.4 Menus

Menus form a central part of the user interface. They are implemented in such a way that an item can be selected either by moving a cursor or pointer to the appropriate item, or by a single keystroke. In this way provision is made for both the novice and experienced user. The novice does not need to remember a variety of commands, while the expert is not held up by having to point to menu items.

The dynamic construction of the create menu described in Section 4.1 requires that the



```
           ‖ of each of the strings that we now describe:
 OList     ‖
  Item     ‖
   para    ‖ 1.   Start label.  This string is used to mark the start of the
           ‖      object in the structure window and is put as close as
           ‖      possible to the start of the object (see Section
           ‖      "strdisp").  In Figure "strings" OList, Item, and para are
           ‖      all start labels.  The null string ("♦") will cause no
  eItem    ‖      label to be displayed.
  Item     ‖
   para    ‖ 2.   End label.  This string is used to mark the end of the
           ‖      object in the structure window, and is put as close as
           ‖      possible to the end of the object (see Section "strdisp").
           ‖      In Figure "strings" eOList and eItem, are end labels.  The
           ‖      null string will cause no label to be displayed (as for
  eItem    ‖      example with the paragraph object).
  Item     ‖
   para    ‖ 3.   Menu key.  This single character indicates which key will
           ‖      select this object class name when the name is displayed in
           ‖      the create menu.
  eItem    ‖
 eOList    ‖
                                                                   Paragraph
 F Footnote   O Ordered list   P Paragraph   U Unordered list   X eXample
                                                                   Structure
```

Figure 5.10: *Start and End labels; Menu key and name*

document class designer provides some extra information in the document class description. Four strings are required for each object class defined. Two of the strings are used in menu construction and the other two when generating the structure display of an object. Figure 5.10 illustrates the use of each of the strings that we now describe:

1. *Start label.* This string is used to mark the start of the object in the structure window and is put as close as possible to the start of the object (see Section 5.2.2). In Figure 5.10 OList, Item, and para are all start labels. The null string ("") will cause no label to be displayed.

2. *End label.* This string is used to mark the end of the object in the structure window, and is put as close as possible to the end of the object (see Section 5.2.2). In Figure 5.10 eOList and eItem, are end labels. The null string will cause no label to be displayed (as for example with the paragraph object).

3. *Menu key.* This single character indicates which key will select this object class name when the name is displayed in the create menu. It is displayed next to the object's

menu name in the create menu (e.g. `O`, `P`, `U`, and `X`) and may not be null.

4. *Menu name.* The object class name as displayed in the create menu (e.g. **`Ordered list`**, **`Paragraph`**, etc.). It is also used to indicate the current object type in the top right hand corner of the menu window and therefore may not be null.

The above strings are defined (in quotes) in the order listed and delimited by "<" and ">", immediately following the unparsing declarations for each object class. See for example the class descriptions in Appendix C.

5.1.5 Attributes

Values are assigned to attributes through the "Edit Attributes" command (which may be tied to a function key). This command opens an attribute window in which all the attributes of the smallest object containing the cursor are displayed together with their current values as illustrated in Figure 5.11.

The user is allowed to edit the attribute values through one of three methods, depending on the type of attribute.

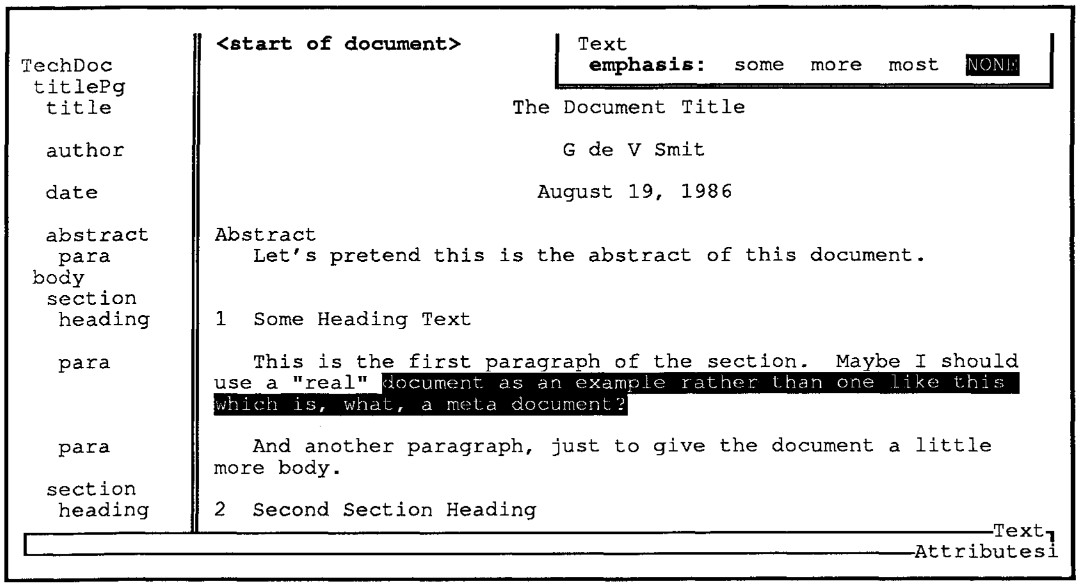
1. If the attribute is of one of the built-in types, the user must type in a value.

2. If the attribute is a user-defined type, an option list of possible values is displayed and the user must select one.

3. If the attribute is a reference to another object, the user has to type in the name (id) of the other object. This other object does not have to exist yet.

5.2 Concrete Representations

Once an abstract document object has been created by the editor, it is possible to generate a concrete representation of the object using the unparsing schemes defined in the class description. This section describes how this is accomplished.



```
                    <start of document>     | Text
TechDoc                                     |  emphasis:  some  more  most  NONE
 titlePg
  title                        The Document Title

  author                         G de V Smit

  date                          August 19, 1986

  abstract         Abstract
   para               Let's pretend this is the abstract of this document.
 body
  section
   heading         1  Some Heading Text

   para               This is the first paragraph of the section.  Maybe I should
                   use a "real" document as an example rather than one like this
                   which is, what, a meta document?

   para               And another paragraph, just to give the document a little
                   more body.

  section
   heading         2  Second Section Heading
                                                                            Text
                                                                      Attributes
```

Figure 5.11: *Editing object attributes*

The attributes of the smallest object that contains the cursor are displayed in the attribute window and may be edited by the user.

Initially, we will assume that the finite automata constructed from the class description are deterministic ones (DFAs), and that the document being unparsed is a complete one. We will then show how both these assumptions can be relaxed to allow for non-deterministic finite automata as well as incomplete documents.

We represent a DFA by an array of states, where each state consists of an array of transitions and a flag indicating whether it is an accepting state or not. Transitions are represented by ordered triples of the form *(symbol, next_state, program_list)*. A *program_list* is a list of unparsing programs identified by a *(scheme, method)* ordered pair.

We use the term *environment* to refer to a set of variables and their current values. The environment of an unparsing method consists of the parameters passed to the method, any local variables declared in the method, as well as a copy of a standard set of variables defining the following:

left margin

right margin

current line length

fill mode (character or word)

justification (left, right, or centre)

underlining (on or off)

bolding (on or off)

reverse display (on or off)

One could regard the above set of variables as a standard set of parameters that are passed to all methods. Any changes made to them by an unparsing program in a method only affects that method and any methods that it calls. Notice the correspondence between an environment and the stack frame in some implementations of programming languages such as Pascal.

An abstract document object is represented as a tree (called the document tree), with each constituent object in the document a node in the tree. The procedure to unparse any subtree of the document tree is given by the following pseudo-code:

```
dfa := the DFA describing this node's class;
Determine the current unparsing scheme and method;
Allocate space for, and initialise the environment;
Execute the method's "initial" unparsing program
  (note that the program may reference and change the
  values of the environment variables);
if the current node is a primitive node then
    Output the text associated with the node;
else
    state := dfa's start state;
    for every child of the current node do
        Using the child's class as a symbol, and state as
          current state, find the next transition in dfa;
        Execute the unparsing program in the current
          method that is associated with the next
          transition (it may reference and change the
          values of the environment variables and may
          include a recursive call to this procedure);
        state := next state as defined by the transition;
    endfor
endif
Execute the method's "final" unparsing program;
De-allocate the environment;
```

No reference is made in this procedure to the final states of the DFA, and no provision is made for not finding the next transition because, as we have stated, it is assumed that documents are complete and therefore syntactically correct. This is not an unreasonable assumption because:

1. It is possible and quite reasonable[5] to disallow the unparsing of an incomplete document for all unparsing schemes except the one that generates a concrete representation for the display device.

2. By using the appropriate finite automata (Mx_ϵ as described in Section 4.2), subsequence incomplete documents are accepted as complete documents. By definition these automata ensure that any document being created is always "complete" and a concrete representation can therefore be generated for the display device.

The unparsing procedure outlined previously assumes that deterministic finite automata are being used. It is simple, however, to adapt the procedure for non-deterministic finite automata (NFAs):

NFAs can easily be simulated by deterministic machines using an algorithm based on the subset construction technique [Aho86]. The only complication lies with the unparsing programs. When simulating an NFA, it is possible that at a given moment, more than one transition must be made, each one labelled with the same symbol, but having different unparsing programs. The simulation algorithm will make all the transitions, but which unparsing program should be executed?

For example, suppose an object x is defined as

```
x = a a
```

and that the unparsing programs $\mathcal{P}$ and $\mathcal{Q}$ are associated with the first and second a-transitions respectively. The class DFA for x is the simple one depicted in Figure 5.12 (the notation a[$\mathcal{P}$] is used to indicate that unparsing program $\mathcal{P}$ is associated with the transition labelled a). In order to handle a subsequence incomplete x, the DFA is changed into the NFA of Figure 5.13.

When using the NFA to unparse an instance of x that consists of a single a, both a-transitions are made[6], but only one of the unparsing programs should be executed.

[5]The incomplete document would be rejected by the batch formatter anyway.

[6]Both transitions are made because they result from the NFA accepting the "strings" aϵ and ϵa.

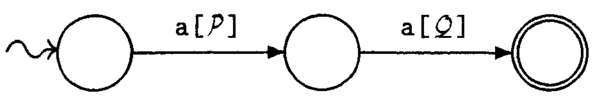


Figure 5.12: *DFA accepting* a a

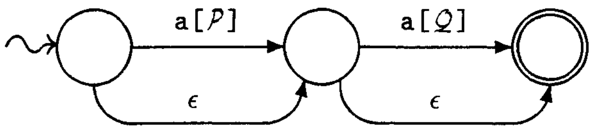


Figure 5.13: *NFA accepting* a? a?

Determining which unparsing program to execute is similar to disambiguating an ambiguous grammar. Since documents are usually created in a front-to-back order, it is natural to always choose the "earliest" unparsing program. The latter is determined as follows:

The states of the finite automaton are numbered in a breadth-first manner and the program that is associated with the transition that originates from the lowest numbered state is selected. If more than one transition has the same originating state, the tie is broken in favour of the one with the lowest numbered destination state.

According to this method, the program $\mathcal{P}$ in our previous example would be executed.

5.2.1 Incremental Formatting

The unparsing scheme used to generate the concrete representation for the display device differs from the other schemes in that it must be possible to execute it in an incremental manner. As the user makes changes to the document, or as different parts of the document are viewed, the concrete representation must be generated for a portion of the document around the current point of interest.

Conceivably the unparsing process could be restarted from the beginning of the document and terminated when enough of the document has been formatted. This, however is unacceptable from a performance point of view. It is therefore necessary to be able to restart the unparsing process almost anywhere in the document.

Restarting points are determined from the class description: the start of any unparsing program in a method is a restart point if the program contains a "start a new line" instruction (e.g. `@nl` or `@vs` – see Appendix B) that will be executed before any text is generated. This ensures that upon restart, the unparsing process will know in which column to place the output.

Before the unparsing process can be restarted at a restart point, the unparse state must be restored. This includes

1. determining the finite automaton to be executed,
2. determining the current state of the automaton,
3. determining the current unparsing method, and
4. restoring the current method's environment.

The finite automaton to be executed is simply the one that describes the class of the current node's parent. The current state, method, and environment information is stored with each node in the document tree and is updated whenever the document is changed.

5.2.2 Structure Display

The structure display consists of at most two labels that are used to mark the start and end of each composite object in the text window parallel to the structure window. Nested levels of composition are indicated by successive indentation[7].

The structure display is generated by the system and the only control that the class designer has over it is specifying whether or not a label will be displayed for a particular class and the label value if a label is displayed.

[7]A system installation parameter specifies the width of the structure window. The structure display is clipped to fit in this window. Since documents generally do not have very deep structures, this is quite acceptable.

The following rules are used to construct the structure display. For the purpose of stating the rules it is assumed the display is constructed sequentially from the start of the document. The concept of "text generated by an object" is used to refer to the text produced as a result of the unparsing of an object. This includes text generated by the system and text (supplied by the user) that is associated with the object or any of its constituent objects.

1. Only composite objects are labeled.

2. As far as possible, the start label appears on the first line of text generated by the object or any of its constituents.

3. As far as possible, the end label appears on the last line of text generated by the object or any of its constituents.

4. Only one label is displayed per line.

5. If a start label has to appear on a line that already contains a start label, the first label is replaced by the second, and the first is displayed on an empty line immediately above its old position. An empty line is a line that does not contain a label or text.

6. If an end label has to appear on a line that already contains a start label, the end label is displayed on an empty line immediately below the start label.

7. When labels are moved to empty lines, any existing empty lines are first used before extra empty lines are inserted.

5.3 Implementation History

Two prototypes have been created, while a full production system is currently under development.

In the first prototype a single class description was hard coded into the system. It was mainly used to investigate various aspects of the user interface, and to lay the ground work for the document manipulation and structure display routines.

The second prototype is very close to the description in this chapter. The differences are accounted for by the fact that some features are only partially implemented, merely to show their feasibility. For example only two unparsing schemes are supported (one for the display and one for a target formatter) — the addition of further unparsing schemes is straightforward. Another limitation is on the size of documents. Currently only fairly small documents can be handled, but this restriction is being removed in the production version (the first prototype had no such restriction).

The system is written in the portable Waterloo Systems Language [Boswe82] and at present runs under IBM's PC-DOS on IBM personal computers. The early prototype was ported to an HP Portable Plus as well as A DEC Rainbow. The system requires about 60Kb of memory, of which around 8Kb are typically used for a class description.

The response time for cursor movement, text entering and structure editing is well within the bounds of acceptability even though no special attention was given to optimisation.

The prototype systems have been used by a number of people within the Computer Systems Groups, as well as few outsiders, amongst which were non-technical staff such as secretaries. Much wider use will be necessary before any pronouncement can be made as to the effectiveness of the system, however, those that have used it have been favourably impressed.

Chapter 6

Conclusions

This disseration investigated the issues related to the accessibility of high quality text processing by non-computer specialists. An important objective was to make high-quality batch-processing document formatters available to the casual computer user. It is proposed that this be achieved by providing a document editing environment which is controlled by the syntax and semantics of a given document class. This way it is possible to reduce and possibly eliminate the requirement for a user to have a knowledge of document composition and formatting languages and systems.

6.1 Contributions

The design and implementation of a prototype document processing system suitable for use by casual users is described. The prototype has the following properties:

1. Entering of text is controlled by an editor which "knows" the structure or syntax of the type of document being entered. When the user enters a new document the type of document is specified so that the editor can be reconfigured.

2. The appearance (semantics) of the text displayed on the screen while the document is being entered or edited, strongly resembles the appearance of the final document,

3. The appearance (semantics) of a document which is produced as an output from the system is a file containing the text of the document and imbedded formatter commands related to its structure. This document can be passed to the appropriate batch formatter to obtain the final high-quality document.

4. During the editing phase the constraints on document structure are considerably relaxed so that restructuring the document is feasible. However, the system does verify that a "complete" document exists before producing final output, prompting the user to complete the document if that is not the case.

5. The system is retargetable in several ways:

    - the documents produced are portable over different batch formatters,
    - the screen display can be modified to take advantage of different display technologies, and
    - the system itself is written using software portability principles.

In order to provide these properties, it was necessary to explore the field of document processing and to advance knowledge in several areas. In particular, a number of contributions have been made, including:

1. We have provided syntactic methods of describing documents, including methods:

    - to describe any document whose structure can be characterized by a context-free grammar,
    - to describe documents by context-free grammars where each production is a regular expression, thus allowing the use of finite automata models for semantic production,
    - to amplify the descriptive power of the such grammars by providing a mechanism for handling exceptions,

- to modify the grammar dynamically so that violations of the grammar are allowed during editing functions, and
- to dynamically generate menus which depend both on the document grammar and on the current state of the document.

2. We have provided semantic methods for describing documents, including methods:

   - to allow the document processing system to be easily retargetted to different types of displays, and
   - to allow the output of the document processing system to be retargetted easily to different existing document formatting systems, by allowing several document descriptions to be available simultaneously and by allowing the document and its structure to be stored separately from its semantics.

6.2 Further Investigation

At present no special provision is made for handling mathematical formulas, tables and graphic objects. Formulas can be incorporated with relative ease (at least as far as the user interface is concerned) by using the excellent approach taken in Grif [Quint86] and Edimath [Quint83, Quint84]. The existing unparsing language will have to be adapted/expanded to cater for the display of the formulas on screen.

Allowing for tables and graphic objects would be somewhat more complicated, primarily because editing of these objects in general requires a completely different user interface than that of text. It is also questionable if these kinds of objects can be described with a context-free language as required by our system.

It is possible that complete table and graphics subsystems will have to be developed, or alternatively, ways have to be found to interface with existing systems. The challenge would be to integrate the graphic, table and text user interfaces in a satisfactory way. Another difficulty lies in the fact that batch formatters usually have very limited capabilities as far as graphic images are concerned, so that it would be difficult to find a batch formatter for the final formatting of documents containing arbitrary graphic objects.

More work is needed in providing a suitable class description editor. While the system itself can be used to create class descriptions, a more specialised system is needed. Such a class designer system would allow the user to define document syntax using both extended regular expressions and finite automata. The class designer can then use either method to view the class syntax and assign unparsing programs to transitions. In this way full use can be made of the information encoded in each state of the finite automata. It would also be helpful if the user could specify the concrete representations using a program-by-example methodology using such principles as established in Query By Example [Zloof75].

Appendix A

Document Class Description Grammar

In the following BNF notation, "is defined as" (::=) and "or" (|) are the only meta characters, and only if they are delimited by spaces. All other symbols are part of the language being described. Where | has to be part of the language, '|' is used. We have followed the convention that all terminal symbols appear in upper case (e.g. ID).

class_def	*::= def_list*
def_list	*::= def_item ;*
	| *def_list ; def_item*
def_item	*::= symb_def*
	| *struct_def attr_defs semantic_def label_def*
symb_def	*::= symb_name = ext_reg_exp*
symb_name	*::= % ID*
struct_def	*::= ID*
	| *ID composition*
composition	*::= = ext_reg_exp exceptions*

```
                | =
ext_reg_exp     ::= e_term
                | ext_reg_exp # e_term
e_term          ::= e_term '|' e_term2
                | e_term2
e_term2         ::= e_term2 e_factor
                | e_factor
e_factor        ::= closure opt_int
                | group ?
                | group
closure         ::= group *
                | group +
group           ::= ( ext_reg_exp )
                | ID
                | symb_name
opt_int         ::=
                | INT
exceptions      ::=
                | : except_list
except_list     ::= except
                | except_list , except
except          ::= + ID
                | - ID


attr_defs       ::=
                | { attr_list }
attr_list       ::= attr_def
```

```
                | attr_list , attr_def
attr_def       ::= attr_name : var_type = value
                | attr_name ^ ID = ID
attr_name      ::= ID
var_type       ::= type_name
                | ( value_list )
type_name      ::= N_INTEGER
                | N_REAL
                | N_STRING
                | N_CHAR
value_list     ::= value
                | value_list , value
value          ::= ID | INT | REAL


semantic_def   ::= [ n_scheme_list ]
n_scheme_list  ::=
                | n_scheme_list scheme
scheme         ::= / ID / n_method_list
n_method_list  ::=
                | n_method_list method
method         ::= METHOD ID parameters
                     local_vars
                     action_list
                   ENDMETHOD
parameters     ::= ( n_var_def_list )
n_var_def_list ::=
                | var_def_list
```

```
var_def_list   ::= var_def
                 | var_def_list , var_def
var_def        ::= ID : var_type
local_vars     ::=
                 | { var_def_list }
action_list    ::= action
                 | action_list , action
action         ::= trans_list : program
trans_list     ::= trans
                 | trans_list , trans
trans          ::= ID
                 | ID . INT


program        ::=
                 | program instr
instr          ::= if_statement
                 | assignment
                 | call
                 | command
                 | expression


if_statement   ::= IF bool_exp THEN program
                   n_elseif_list
                   n_else
                   ENDIF
n_elseif_list  ::=
                 | n_elseif_list else_if
```

```
else_if       ::= ELSEIF bool_exp THEN program

n_else        ::=
                | ELSE program

bool_exp      ::= expression
                | expression rel_op expression
                | @EMPTY
                | @FIRST ( ID , rel_op , value )
                | @LAST ( ID , rel_op , value )

rel_op        ::= < | > | <> | = | > = | < =


assignment    ::= ID = expression

call          ::= @ ID call_parms

call_parms    ::= ( n_exp_list )

n_exp_list    ::=
                | exp_list

exp_list      ::= expression
                | exp_list , expression


command       ::= @NL
                | @VS ( INT )
                | @TODAY
                | @UNPARSE ( ID , call )
                | @CENTRE
                | @ECENTRE
                | @LEFT
                | @RIGHT
                | @NORMAL
```

```
                 | @BOLD
                 | @EBOLD
                 | @UNDERLINE
                 | @EUNDERLINE
                 | @REVERSE
                 | @EREVERSE
                 | @FILL_WORDS
                 | @FILL_CHARS
                 | @LM op ( INT )
                 | @RM op ( INT )
                 | @LL op ( INT )

op              ::= + | - | =


expression      ::= expression + term
                 | expression - term
                 | term
term            ::= term * factor
                 | term / factor
                 | term ! factor
                 | factor
factor          ::= ( expression )
                 | & ID
                 | ID
                 | INT
                 | REAL
                 | STRING
```

```
                | reference
reference       ::= ID ^ . ID

label_def       ::=
                | < STRING , STRING , STRING , STRING >
```

Appendix B

Unparsing Instructions

The following is a list of all the instructions and operators that may be used in the unparsing programs of an object class' semantic description.

"*<string>*"

Generate (emit) the string *<string>*.

<name>

The value of the attribute, parameter or variable with the given name.

&*<name>*

The value of the attribute, parameter or variable with the given name, converted to a string.

=

Assignment

!

String concatenation. E.g. `"abc"!"XYZ"` is equivalent to `"abcXYZ"`.

if *<bool_exp>* **then** *<program>* **else** *<program>* **endif**

The obvious.

<name>^.*<attr_name>*

The value of the attribute *<attr_name>* of the object to which the variable *<name>* points. *<name>* must be a link variable.

@nl

Start a new line.

@vs(*<number>*)

Conditional vertical space: generate a maximum of *<number>* newline characters, i.e. if necessary, add newline characters to the output so that *<number>* of them will follow each other.

@today

Supply today's date as a string.

@empty

The value 1 if the object being unparsed has no content, otherwise the value 0. In other words, if the current object is a composite one and it has no children, or if it is a basic object and has no text (i.e. the null string) associated with it, the value of this operator is 1.

@first(*<attribute>*, *<relation>*, *<value>*)

The value 1 if the current object is the first object of its class (starting from the beginning of the document) that has the indicated attribute whose value satisfies the given relation. *<relation>* can be any of the relational operators = (equal), < (less than), > (greater than), <= (less or equal to), >= (greater or equal to), or <> (not equal to). If the current object does not have the indicated attribute, the attribute value does not satisfy the relation, or the object is not the first of its kind satisfying the condition, the value of the operator is 0.

@last(*<attribute>*, *<relation>*, *<value>*)

The value 1 if the current object is the last object of its class (before the end of the document) that has the indicated attribute whose value satisfies the given relation. <*relation*> can be any of the relational operators = (equal), < (less than), > (greater than), <= (less or equal to), >= (greater or equal to), or <> (not equal to). If the current object does not have the indicated attribute, the attribute value does not satisfy the relation, or the object is not the last of its kind satisfying the condition, the value of the operator is 0.

@<*method_name*>(<*parm*$_1$>, ... <*parm*$_p$>)

Unparse the constituent object, using its <*method_name*> unparsing method (in the current scheme) and pass the given parameters (if any) to that method.

@unparse(<*obj_name*>, **@**<*method_name*>(<*parm*$_1$>, ... <*parm*$_p$>))

Unparse the object <*obj_name*> using its <*method_name*> unparsing method (in the current scheme) and pass the given parameters (if any) to that method. <*obj_name*> must be a reference to an existing object.

@centre ... @ecentre

Centre lines.

@left

Left justify lines.

@right

Right justify lines.

@bold ... @ebold

Bold text.

@underline ... @eunderline

Underline text.

@reverse ... @ereverse

Display text in reverse video.

@normal

The combined effect of @ebold, @eunderline, and @ereverse.

@fill_words

Perform word-wrap during text output, i.e. fill lines with words.

@fill_chars

Do not perform word-wrap during text output, i.e. fill lines with characters rather than words.

@lm+(<*n*>)

Adjust the left margin by +<*n*> columns and position the output pointer there (<*n*> is an unsigned number). If there is already text in that column, position the output pointer at the end of that text.

@lm–(<*n*>)

Adjust the left margin by –<*n*> columns and position the output pointer there (<*n*> is an unsigned number). If there is already text in that column, position the output pointer at the end of that text.

@lm=(<*n*>)

Set the left margin to be <*n*> columns wide and position the output pointer on the left margin (<*n*> is an unsigned number). If there is already text in that column, position the output pointer at the end of that text.

@rm+(<*n*>)

@rm–(<*n*>)

@rm=(<*n*>)

Same as @lm, but pertaining to the right margin.

@ll+(<*n*>)

@ll-(<n>)

@ll=(<n>)

Adjustment of line length.

Appendix C

Document Class Descriptions

This appendix contains a number of class descriptions serving as examples of the use of the class description language.

C.1 A Simple Technical Document

This is a class description for a class of simple technical documents. The semantic definition contains unparsing schemes to generate concrete representations suitable for

1. a screen display,
2. input to a GML formatter, and
3. input to TeX using the LaTeX macros.

In order to limit the size of the description, not all objects that would normally be part of a document class are described here.

Examples of the display output for this class description can be found throughout Chapter 5. Appendix D.1 contains sample output for the GML and LaTeX unparsing schemes.

```
%basicObj = paragraph | footnote | ordList | unordList | example;
%anyText = (TEXT | quote) (TEXT | quote | fnRef)*;
```

```
TechDoc = titlePage body appendices?
   [ /display/
       Method Standard()
         initial:     ,
         titlePage:  @Standard(),
         body:       @Standard(),
         appendices: @Standard(),
         final:
       endMethod

     /gml/
       Method Standard()
         initial:    @vs(1) ":GDOC",
         titlePage:  @Standard(),
         body:       @Standard(),
         appendices: @Standard(),
         final:      @nl ":eGDOC"
       endMethod

     /latex/
       Method Standard()
         initial:    @vs(1) "\documentstyle{report}"
                     @nl "\begin{document}",
         titlePage:  @Standard(),
         body:       @Standard(),
         appendices: @Standard(),
         final:      @vs(1) "\end{document}"
       endMethod
   ]
   < "TechDoc", "", "", "" >;


titlePage = title author? date? abstract?
   [ /display/
       Method Standard()
         initial:  @centre,
         title:    @Standard(),
         author:   @Standard(),
         date:     @Standard(),
         abstract: @centre @Standard(),
         final:
       endMethod
```

```
      /gml/
        Method Standard()
          initial:  @nl ":TITLEP",
          title:    @Standard(),
          author:   @Standard(),
          date:     @Standard(),
          abstract: @Standard(),
          final:    @nl ":eTITLEP"
        endMethod

      /latex/
        Method Standard()
          initial:  ,
          title:    @Standard(),
          author:   @Standard(),
          date:     @Standard(),
          abstract: @Standard(),
          final:    @vs(1) "\maketitle"
        endMethod
    ]
    < "titlePg", "", "T", "Title page" >;



title = PLAIN_TEXT
    { footRef ^ footnote = NONE }
    [ /display/
        Method Standard()
          initial:    @vs(2),
          PLAIN_TEXT: @Standard(),
          final:      @vs(2)
        endMethod

      /gml/
        Method Standard()
          initial:    @nl ":TITLE."
                      if footRef <> NONE then
                         " fnid='" footRef^.id "'"
                      endif
                      ".",
          PLAIN_TEXT: @Standard(),
        final:
      endMethod

    /latex/
      Method Standard()
        initial:    @vs(1) "\title{",
        PLAIN_TEXT: @Standard(),
        final:      if footRef <> NONE then
                       @unparse( footRef, @Standard() )
                    endif
                    "}"
      endMethod
   ]
   < "title", "", "T", "Title" >;


author = PLAIN_TEXT
   [ /display/
      Method Standard()
        initial:    @vs(1),
        PLAIN_TEXT: @Standard(),
        final:
      endMethod

    /gml/
      Method Standard()
        initial:    @nl ":AUTHOR.",
        PLAIN_TEXT: @Standard(),
        final:
      endMethod

    /latex/
      Method Standard()
        initial:    @vs(1) "\author{",
        PLAIN_TEXT: @Standard(),
        final:      "}"
      endMethod
   ]
   < "author", "", "U", "aUthor" >;


date = PLAIN_TEXT
```

```
   [ /display/
       Method Standard()
         initial:    @vs(2),
         PLAIN_TEXT: @aDate(),
         final:
       endMethod

     /gml/
       Method Standard()
         initial:    @nl ":DATE.",
         PLAIN_TEXT: @Standard(),
         final:
       endMethod

     /latex/
       Method Standard()
         initial:    @vs(1) "\date{",
         PLAIN_TEXT: @Standard(),
         final:      "}"
       endMethod
   ]
   < "date", "", "D", "Date" >;


abstract = %basicObj+ : -footnote, -fnRef
   [ /display/
       Method Standard()
         initial:  @vs(2) "Abstract" @vs(1),
         basicObj: @Standard(),
         final:
       endMethod

     /gml/
       Method Standard()
         initial:  @nl ":ABSTRACT.",
         basicObj: @Standard(),
         final:
       endMethod

     /latex/
       Method Standard()
         initial:  @vs(1) "\begin{abstract}",
        basicObj: @Standard(),
        final:    @vs(1) "\end{abstract}"
      endMethod
  ]
  < "abstract", "", "A", "Abstract" >;



body = part+ | section+
  [ /display/
      Method Standard()
        { rank : integer }
        initial:  rank=0,
        part:     rank=rank+1 @Standard(rank),
        section:  rank=rank+1 @Standard(&rank),
        final:
      endMethod

    /gml/
      Method Standard()
        initial:  @nl ":BODY.",
        part:     @Standard(),
        section:  @Standard(1),
        final:
      endMethod

    /latex/
      Method Standard()
        initial:  ,
        part:     @Standard(),
        section:  @Standard(1),
        final:
      endMethod
  ]
  < "body", "", "B", "Body" >;


part = heading  %basicObj*  section*
  { id         : string = NONE,
    shortTitle : string = NONE
  }
  [ /display/
```

```
    Method Standard( rank:integer )
      { sectCount : integer }
      initial:  sectCount=0
                @vs(2) @centre "PART " rank @ecentre,
      heading:  @PartHead(),
      basicObj: @Standard(),
      section:  sectCount=sectCount+1
                @Standard(&sectCount),
      final:
    endMethod

  /gml/
    Method Standard()
      initial:  @nl ":HO"
                if id <> NONE then
                  " ID=" id
                endif
                if shortTitle <> NONE then
                   " STITLE=""" shortTitle """"
                endif
                "." ,
      heading:  @Standard(),
      basicObj: @Standard(),
      section:  @Standard(1),
      final:
    endMethod

  /latex/
    Method Standard()
      initial:  @vs(1) "\part"
                if shortTitle <> NONE then
                   "[" shortTitle "]"
                endif,
      heading:  @Standard(id),
      basicObj: @Standard(),
      section:  @Standard(1),
      final:
    endMethod
]
< "part", "", "P", "Part" >;
```

```
section = heading  %basicObj*  section*
   { id         : string = NONE,
     shortTitle : string = NONE
   }
   [ /display/
       Method Standard( hNum:string )
         { ssCount:integer }
         initial:  ssCount=0 @vs(2) hNum " ",
         heading:  @Standard(),
         basicObj: @Standard(),
         section:  ssCount=ssCount+1
                   @Standard(hNum!"."!&ssCount),
         final:
       endMethod

     /gml/
       Method Standard( level:integer )
         initial:  @nl ":H" level
                   if id <> NONE then
                      " ID=" id
                   endif
                   if shortTitle <> NONE then
                      " STITLE=""" shortTitle """"
                   endif
                   "." ,
         heading:  @Standard(),
         basicObj: @Standard(),
         section:  if level < 6 then
                      @Standard(level+1)
                   else
                      @Standard(level)
                   endif,
         final:
       endMethod

     /latex/
       Method Standard( level:integer )
         initial:  @vs(1)
                   if level=1 then
                      "\chapter"
                   elseif level=2 then
                      "\section"
```

```
                 elseif level=3 then
                    "\subsection"
                 elseif level=4 then
                    "\subsubsection"
                 elseif level=5 then
                    "\paragraph"
                 else
                    "\subparagraph"
                 endif
                 if shortTitle <> NONE then
                    "[" shortTitle "]"
                 endif,
       heading:  @Standard(id),
       basicObj: @Standard(),
       section:  if level < 6 then
                    @Standard(level+1)
                 else
                    @Standard(level)
                 endif,
       final:
     endMethod
  ]
  < "section", "", "S", "Section" >;


heading = PLAIN_TEXT
  [ /display/
     Method Standard()
       initial:    ,
       PLAIN_TEXT: @Standard(),
       final:      @vs(2)
     endMethod

     Method PartHead()
       initial:    @vs(1) @centre,
       PLAIN_TEXT: @Standard(),
       final:
     endMethod

    /gml/
     Method Standard()
```

```
        initial:    ,
        PLAIN_TEXT: @Standard(),
        final:
      endMethod

    /latex/
      Method Standard(id:string)
        initial:    "{"
                    if id <> NONE then
                       "\label{" id "}"
                    endif,
        PLAIN_TEXT: @Standard(),
        final:      "}"
      endMethod
  ]
  < "heading", "", "H", "Heading" >;


paragraph  = %anyText
  [ /display/
      Method Standard()
        initial:  @vs(2) "    " ,
        anyText:  @Standard(),
        final:
      endMethod

      Method Blocked()
        initial:  @vs(2),
        anyText:  @Standard(),
        final:
      endMethod

      Method Plain()
        initial:  ,
        anyText:  @Standard(),
        final:
      endMethod

    /gml/
      Method Standard()
        initial:  @nl ":P." @nl,
        anyText:  @Standard(),
        final:
      endMethod

      Method Plain()
        initial:  @nl,
        anyText:  @Standard(),
        final:
      endMethod

    /latex/
      Method Standard()
        initial:  @vs(2),
        anyText:  @Standard(),
        final:
      endMethod

      Method Plain()
        initial:  ,
        anyText:  @Standard(),
        final:
      endMethod
  ]
  < "para", "", "P", "Paragraph" >;


footnote = paragraph %basicObj* : -footnote, -fnRef
  { id : string = required }
  [ /display/
      Method Standard()
        initial:  @vs(2) "[" id ": " @lm+(3) @rm-(3),
        paragraph.1: @Plain(),
        basicObj: @Standard() ,
        final:    @vs(1) @lm-(3) "]" @vs(2)
      endMethod

    /gml/
      Method Standard()
        initial:  ,
        basicObj: "",
        final:
      endMethod
      Method AsFootNote()
        initial:  @nl ":FN id=" id,
        paragraph.1: @Plain(),
        basicObj: @Standard(),
        final:    @nl ":eFN."
      endMethod

    /latex/
      Method Standard()
        initial:  ,
        basicObj: "",
        final:
      endMethod

      Method AsFootNote()
        initial:  "\footnote{",
        paragraph.1: @Plain(),
        basicObj: @Standard(),
        final:    "}"
      endMethod
   ]
   < "footnote", "endFoot","F", "Footnote" >;



ordList  =  (listItem | listPart)+
   { compact : (0, 1, 2) = NONE }
   [ /display/
      Method Standard()
        { itemcount : integer }
        initial:  itemcount=0  @vs(2),
        listItem: itemcount=itemcount+1
                  @Ordered(itemcount),
        listPart: @Standard(),
        final:    @vs(2)
      endMethod

    /gml/
      Method Standard()
        initial:  @nl ":OL"
                  if compact <> NONE then
                     " COMPACT=" compact
```

```
                 endif
                 ".",
        listItem: @Standard(),
        listPart: @Standard(),
        final:    @nl ":eOL."
      endMethod

   /latex/
      Method Standard()
        initial:  @vs(1) "\begin{enumerate}",
        listItem: @Standard(),
        listPart: @Standard(),
        final:    @vs(1) "\end{enumerate}"
      endMethod
   ]
   < "OList", "eOList", "O", "Ordered list" >;


unordList  =  (listItem | listPart)+
   { compact : (0, 1, 2) = NONE }
   [ /display/
      Method Standard()
        initial:  ,
        listItem: @Unordered(),
        listPart: @Standard(),
        final:    @vs(2)
      endMethod

    /gml/
      Method Standard()
        initial:  @nl ":UL"
                  if compact <> NONE then
                     " COMPACT=" compact
                  endif
                  ".",
        listItem: @Standard(),
        listPart: @Standard(),
        final:    @nl ":eUL."
      endMethod

   /latex/
      Method Standard()
```

```
        initial:  @vs(1) "\begin{itemize}",
        listItem: @Standard(),
        listPart: @Standard(),
        final:    @vs(1) "\end{itemize}"
      endMethod
   ]
   < "UList", "eUList", "U", "Unordered list" >;


listItem  =  paragraph %basicObj*
   [ /display/
      Method Ordered( index : integer )
        initial:      @vs(2) index "." @lm+(5),
        paragraph.1: @Plain(),
        basicObj:     @Standard(),
        final:        @vs(2)
      endMethod

      Method Unordered()
        initial:      @vs(2) "*" @lm+(5),
        paragraph.1: @Plain(),
        basicObj:     @Standard(),
        final:        @vs(2)
      endMethod

     /gml/
      Method Standard()
        initial:      @nl ":LI." ,
        paragraph.1: @Plain(),
        basicObj:     @Standard(),
        final:
      endMethod

     /latex/
      Method Standard()
        initial:      @vs(1) "\item ",
        paragraph.1: @Plain(),
        basicObj:     @Standard(),
        final:
      endMethod
   ]
   < "Item", "eItem", "I", "Item" >;
```

```
listPart = paragraph  (paragraph | example)*
   [ /display/
       Method Standard()
         initial:     @vs(2) ,
         paragraph.1: @Blocked(),
         paragraph.2: @Standard(),
         example:     @Standard(),
         final:       @vs(2)
       endMethod

     /gml/
       Method Standard()
         initial:     @nl ":LP." ,
         paragraph.1: @Plain(),
         paragraph.2: @Standard(),
         example:     @Standard(),
         final:       @vs(2)
       endMethod

     /latex/
       Method Standard()
         initial:     @vs(1) "\vskip{\bigskipamount}",
         paragraph:   @Standard(),
         example:     @Standard(),
         final:       @vs(1) "\vskip{\bigskipamount}"
                      @vs(1)
       endMethod
   ]
   < "LP", "", "P", "list Part" >;



example = (aLine | %basicObj)+ : -example, -footnote, -fnRef
   [ /display/
       Method Standard()
         initial:   @lm+(3),
         aLine:     @Standard(),
         basicObj:  @Standard(),
         final:
       endMethod
```

```
    /gml/
      Method Standard()
        initial:   @nl ":XMP",
        aLine:     @Standard(),
        basicObj:  @Standard(),
        final:     @nl ":eXMP."
      endMethod

    /latex/
      Method Standard()
        initial:   @vs(2) "{\tt ",
        aLine:     @Standard(),
        basicObj:  @Standard(),
        final:     "}"
      endMethod
  ]
  < "example", "", "X", "eXample" >;


aLine = %anyText
  [ /display/
      Method Standard()
        initial:  @vs(1) @fill_chars,
        anyText:  @Standard(),
        final:    @vs(1)
      endMethod

    /gml/
      Method Standard()
        initial:  @vs(1) @fill_chars,
        anyText:  @Standard(),
        final:    @vs(1)
      endMethod

    /latex/
      Method Standard()
        initial:  @vs(1) "\\ ",
        anyText:  @Standard(),
        final:    @vs(1)
      endMethod
  ]
```

```
    < "line", "", "L", "line" >;


appendices = section+
    [ /display/
        Method Standard()
          { apdxCnt : integer }
          initial:  apdxCnt = 0,
          section:  apdxCnt=apdxCnt+1
                    @Standard("A"!&apdxCnt),
          final:
        endMethod

      /gml/
        Method Standard()
          initial:  @nl ":APPENDIX.",
          section:  @Standard(1),
          final:
        endMethod

      /latex/
        Method Standard()
          initial:  @vs(1) "\appendix" @nl,
          section:  @Standard(1),
          final:
        endMethod
    ]
    < "appdx", "", "A", "Appendix" >;


 PLAIN_TEXT
    [ /display/
        Method Standard()
          initial:  ,
          final:
        endMethod

        Method aDate()
          initial:  @vs(2)
                    if @empty then
                       @today
                    endif,
        final:
      endMethod

    /gml/
      Method Standard()
        initial:  ,
        final:
      endMethod

    /latex/
      Method Standard()
        initial:  ,
        final:
      endMethod
  ]
  < "", "", "P", "Plain text" >;


TEXT
  { emphasis : (some, more, most) = NONE }
  [ /display/
      Method Standard()
        initial:  if emphasis = NONE then
                     @normal
                  elseif emphasis = some then
                     @underline
                  elseif emphasis = more then
                     @bold
                  elseif emphasis = most then
                     @bold @underline
                  endif ,
        final:    if emphasis = some then
                     @eunderline
                  elseif emphasis = more then
                     @ebold
                  elseif emphasis = most then
                     @eunderline @ebold
                  endif
      endMethod

    /gml/
      Method Standard()
```

```
          initial:  if emphasis = some then
                       ":HP1."
                    elseif emphasis = more then
                       ":HP2."
                    elseif emphasis = most then
                       ":HP3."
                    endif ,
          final:    if emphasis = some then
                       ":eHP1."
                    elseif emphasis = more then
                       ":eHP2."
                    elseif emphasis = most then
                       ":eHP3."
                    endif
        endMethod

      /latex/
        Method Standard()
          initial:  if emphasis = some then
                       "{\em "
                    elseif emphasis = more then
                       "{\bf "
                    elseif emphasis = most then
                       "{\sc "
                    endif ,
          final:    if emphasis <> NONE then
                       "}"
                    endif
        endMethod
    ]
    < "", "", "T", "Text" >;


quote = %anyText
    [ /display/
        Method Standard()
          initial:  """",
          anyText:  @Standard(),
          final:    """"
        endMethod

      /gml/
```

```
        Method Standard()
          initial:  ":Q.",
          anyText:  @Standard(),
          final:    ":eQ."
        endMethod

      /latex/
        Method Standard()
          initial:  "``",
          anyText:  @Standard(),
          final:    "''"
        endMethod
    ]
    < "", "", "Q", "Quote">;


fnRef =
    { footRef ^ footnote = REQUIRED }
    [ /display/
        Method Standard()
          initial:  "{see footnote " footRef^.id "}",
          final:
        endMethod

      /gml/
        Method Standard()
          initial:  if @first( footRef, =, footRef ) then
                       ":FNREF refid=" footRef^.id "."
                    else
                       @unparse( footRef, @AsFootNote() )
                    endif,
          final:
        endMethod

      /latex/
        Method Standard()
          initial:  @unparse( footRef, @AsFootNote() ),
          final:
        endMethod
    ]
    < "", "", "R", "footRef" >;
```

C.2 A Business Letter Class

This is a partial class description for a business letter. In order to limit the size of the description, some objects (e.g. paragraph) are not defined. Their definitions can be obtained from the class description of Appendix C.1. Furthermore, unparsing schemes are only given for GML and LaTeX and none for the display.

The purpose of this example is to illustrate how different target formatter input models can be combined into one class description. Note for example how the closing greeting and signature are handled in the close object.

Appendix D.2 illustrates the output obtained with the given unparsing schemes.

```
%Objects  =  paragraph | simpList | ordList | unordList;

 BusLetter  =  header  body  close
   [ /gml/
        Method Standard()
          initial:              @left @fill_words @ll=(80) @lm=(0) @rm=(6)
                                ":GDOC",
          header,body,close:    @Standard(),
          final:                @nl ":eGDOC"
        endMethod
     /LaTeX/
        Method Standard()
          initial:              @left @fill_words @ll=(80) @lm=(0) @rm=(6)
                                "\documentstyle{letter}"
                                @nl "\begin{document}",
          header,body,close:    @Standard(),
          final:                @nl "\end{document}"
        endMethod
   ] < "BusLetter", "eBusLetter", "B", "Business Letter" >;

 header  =  fromAddress?  date  toAddress  salutation
   [ /gml/
        Method Standard()
          fromAddress, date, toAddress, salutation: @Standard()
        endMethod
     /LaTeX/
        Method Standard()
        fromAddress, date, toAddress, salutation: @Standard()
      endMethod
  ] < "header", "", "H", "Header" >;

fromAddress  =  aLine aLine*
  { depth : integer = NONE }
  [ /gml/
      Method Standard()
        initial: @nl ":FROM"
                 if depth <> NONE then " DEPTH=" depth endif ".",
        aLine:   @Standard()
      endMethod
    /LaTeX/
      Method Standard()
        initial: @nl "\address{",
        aLine.1: @Standard(),
        aLine.2: "\\@nl" @Standard(),
        final:   "}"
      endMethod
  ] < "from", "", "F", "From" >;

date
  { depth : integer = NONE,
    align : (left, right) = NONE
  }
  [ /gml/
      Method Standard()
        initial: @nl ":DATE"
                 if align <> NONE then " ALIGN=" align endif
                 if depth <> NONE then " DEPTH=" depth endif
                 "."
      endMethod
    /LaTeX/
      Method Standard()
        initial: @nl if @empty then "\date{"
                 else "\renewcommand{\today}{" endif
        final:   "}"
      endMethod
  ] < "date", "", "D", "Date" >;


toAddress  =  aLine aLine*
```

```
  { depth : integer = NONE }
  [ /gml/
        Method Standard()
          initial: @nl ":TO"
                   if depth <> NONE then " DEPTH=" depth endif ".",
          aLine.1: @NoNewline(),
          aLine.2: @Standard()
        endMethod
    /LaTeX/
        Method Standard()
          initial: @nl "\begin{letter}{",
          aLine.1: @Standard(),
          aLine.2: "\\@nl" @Standard(),
          final:   "}"
        endMethod
  ] < "to", "", "T", "To" >;

salutation
  [ /gml/
        Method Standard()
          initial: @nl ":OPEN."
        endMethod
    /LaTeX/
        Method Standard()
          initial: @nl "\opening{",
          final:   "}"
        endMethod
  ] < "salute", "", "S", "Salutation" >;

body  =  subject?  %Objects*
  [ /gml/
        Method Standard()
          subject, Objects: @Standard()
        endMethod
    /LaTeX/
        Method Standard()
          subject, Objects: @Standard()
        endMethod
  ] < "body", "", "B", "Body" >;

subject
  { depth : integer = NONE }
  [ /gml/
        Method Standard()
          initial: @nl ":SUBJECT"
                   if depth <> NONE then " DEPTH=" depth endif "."
        endMethod
    /LaTeX/
        Method Standard()
          initial: @nl "{\bf ",
          final: "}"
        endMethod
  ] < "subject", "", "B", "suBject" >;

close = aLine aLine*
  { depth    : integer = NONE,
    greeting : string = "Yours sincerely",
    initials : string = NONE
  }
  [ /gml/
        Method Standard()
          initial: @nl ":CLOSE"
                   if depth <> NONE then " DEPTH=" depth endif "."
                   if greeting <> NONE then greeting endif,
          aLine:   @Standard(),
          final:   @nl ":eCLOSE."
                   if initials <> NONE then initials endif
        endMethod
    /LaTeX/
        Method Standard()
          initial: @nl "\signature{",
          aLine.1: @Standard(),
          aLine.2: "\\@nl" @Standard(),
          final:   "}" @nl "\closing{"
                   if greeting <> NONE then greeting endif "}"
                   if initials <> NONE then @nl "\ps " initials endif
       endMethod
  ] < "close", "", "C", "Close" >;

aLine
  [ /gml/
        Method Standard()
          initial: @nl
        endMethod
        Method NoNewline()
          initial:
        endMethod
     /LaTeX/
        Method Standard()
          initial:
        endMethod
   ] < "line", "", "L", "Line" >;
```

C.3 A Class Description For Classes

This class description describes the class of document class descriptions and can be used to turn the prototype system into a class description editor. This description serves both as an example of a class description and as an example of the output produced by that class description.

Only one unparsing scheme is given, namely one that produces a file that may serve as input to a class description compiler (that compiles the description into a set of tables suitable for use by the prototype system). The same scheme can be used for display purposes. What follows is the content of the file just mentioned, when the class description itself is used as input. In other words, the description given here describes itself.

```
%type = an_int | a_real | a_string | pointer | enumerate;
%command = nl | vs | today | centre | ecentre | left | right | normal |
    bold | ebold | underline | eunderline | reverse | ereverse | fillWord
    | fillChar | lm | rm | ll;

 docClass = macro* objClass+
    [ /classgen/
        Method Standard()
          initial: @ll=(80) @lm=(1) @rm=(6) @left @fill_words @normal,
          macro,objClass: @Standard(),
          final:
        endMethod
    ] < "Class", "endClass", "D", "DocClass" >;

 macro = name expression
    [ /classgen/
```

```
        Method Standard()
          initial: @vs(1) @lm-(1),
          name: @Standard(),
          expression: " = " @lm+(4) @Standard(),
          final: ";"
        endMethod
    ] < "macro", "", "M", "Macro" >;

objClass = name composition? attributes? unpScheme* labels
    [ /classgen/
        Method Standard()
          { hadUnp:integer = 0 }
          initial: @vs(2),
          name,composition,attributes: @Standard(),
          unpScheme: if hadUnp then @vs(2) else @vs(1) "    [" endif
            @Standard() hadUnp=0,
          labels: @vs(1) if hadUnp then "    ] " else "    []" endif
            @Standard() hadUnp=1,
          final: if hadUnp=0 then @vs(1) "[]" endif ";"
        endMethod
    ] < "object", "", "O", "Object" >;

name
    [ /classgen/
        Method Standard()
          initial:,
          final:
        endMethod
    ] < "", "", "N", "Name" >;

composition = expression (inclusion | exclusion)*
    [ /classgen/
        Method Standard()
          { first:integer = 1 }
          initial: " = ",
          inclusion,exclusion: if first then " : " else ", " endif
            @Standard() first=0,
          final:
        endMethod
    ] < "", "", "C", "Composition" >;

inclusion
```

```
   [ /classgen/
       Method Standard()
         initial: "+",
         final:
       endMethod
   ] < "", "", "I", "Inclusion" >;

exclusion
   [ /classgen/
       Method Standard()
         initial: "-",
         final:
       endMethod
   ] < "", "", "E", "Exclusion" >;

expression
   [ /classgen/
       Method Standard()
         initial:,
         final:
       endMethod
   ] < "", "", "X", "Xpression" >;


attributes = (name %type default?)+
   [ /classgen/
       Method Standard()
         { moreThanOne:integer = 0 }
         initial: @lm+(3) @vs(1) "{" @lm+(2),
         name: if moreThanOne then "," @vs(1) endif @Standard()
           moreThanOne=1,
         type,default: @Standard(),
         final: if moreThanOne then @lm-(2) @vs(1) "}" else " }" endif
       endMethod
   ] < "attr", "", "A", "Attributes" >;

an_int =
   [ /classgen/
       Method Standard()
         initial: ":integer",
         final:
       endMethod
   ] < "", "", "I", "Integer" >;

a_real =
   [ /classgen/
       Method Standard()
         initial: ":real",
         final:
       endMethod
   ] < "", "", "R", "Real" >;

a_string =
   [ /classgen/
       Method Standard()
         initial: ":string",
         final:
       endMethod
   ] < "", "", "S", "String" >;

pointer
   [ /classgen/
       Method Standard()
         initial: "^",
         final:
       endMethod
   ] < "", "", "P", "Pointer" >;

enumerate = name name*
   [ /classgen/
       Method Standard()
         initial: "(",
         name.1: @Standard(),
         name.2: ", " @Standard(),
         final:  ")"
       endMethod
   ] < "", "", "E", "Enumerate" >;

default
   [ /classgen/
       Method Standard()
         initial: " = ",
         final:
       endMethod
   ] < "", "", "D", "Default" >;

unpScheme = name aMethod+
   [ /classgen/
       Method Standard()
         initial: @lm+(5),
         name: "/" @Standard() "/",
         aMethod: @Standard(),
         final:
       endMethod
   ] < "scheme", "", "S", "Scheme" >;


aMethod = name parms variables? (transition+ program)*
   [ /classgen/
       Method Standard()
         { firstTrans:integer = 1,
           comma:integer = 0,
           undoIndent:integer = 0
         }
         initial: @lm+(2) @vs(1) "Method ",
         name,parms,variables: @Standard(),
         transition: if comma then "," endif if undoIndent then @lm-(4)
           endif if firstTrans then @lm+(2) @vs(1) endif @Standard() if
           firstTrans then @lm+(2) endif firstTrans=0 comma=1,
         program: ":" @Standard() undoIndent=1 firstTrans=1,
         final: if comma then @lm-(4) endif @vs(1) "endMethod"
       endMethod
   ] < "method", "", "M", "Method" >;

parms = (name %type)*
   [ /classgen/
       Method Standard()
         { comma:integer = 0 }
         initial: "(",
         name: if comma then ", " endif @Standard() comma=1,
         type: @Standard(),
         final: ")"
       endMethod
   ] < "", "", "P", "Parms" >;

variables = (name %type default?)+
   [ /classgen/
       Method Standard()
         { moreThanOne:integer = 0 }
         initial: @lm+(2) @vs(1) "{" @lm+(2),
         type,default: @Standard(),
         name: if moreThanOne then "," @vs(1) endif @Standard()
           moreThanOne=1,
         final: if moreThanOne then @lm-(2) @vs(1) "}" else " }" endif
       endMethod
   ] < "variables", "", "V", "Variables" >;

transition
   { occurrence:integer = NONE }
   [ /classgen/
       Method Standard()
         initial:,
         final: if occurrence<>NONE then "." occurrence endif
       endMethod
   ] < "", "", "T", "Transitions" >;

program = (an_if | assign | call | %command | expression)*
   [ /classgen/
       Method Standard()
         initial:,
         an_if,assign,call,command: @Standard(),
         expression: " " @Standard(),
         final:
       endMethod
   ] < "", "", "R", "pRrogram" >;

an_if = boolExpr program (boolExpr program)* program?
   [ /classgen/
       Method Standard()
         initial: " if",
         boolExpr.1: @Standard() " then",
         boolExpr.2: " elseif" @Standard() " then ",
         program: @Standard(),
         final: " endif"
       endMethod
   ] < "", "", "I", "If" >;

boolExpr = expression | empty | first | last
```

```
   [ /classgen/
       Method Standard()
         initial: " ",
         expression,empty,first,last: @Standard(),
         final:
       endMethod
   ] < "", "", "B", "BoolExpr" >;

empty =
   [ /classgen/
       Method Standard()
         initial: "@empty",
         final:
       endMethod
   ] < "", "", "E", "isEmpty" >;


first = name expression
   { relation:(less, le, eq, ne, ge, gt) = REQUIRED }
   [ /classgen/
       Method Standard()
         initial: "@first(",
         name: @Standard() if relation=less then ",<," elseif
           relation=le then ",<=," elseif relation=eq then ",=," elseif
           relation=ne then ",<>," elseif relation=ge then ",>=," elseif
           relation=gt then ",>," endif,
         expression: @Standard(),
         final: ")"
       endMethod
   ] < "", "", "F", "First" >;

last = name expression
   { relation:(less, le, eq, ne, ge, gt) = REQUIRED }
   [ /classgen/
       Method Standard()
         initial: "@last(",
         name: @Standard() if relation=less then ",<," elseif
           relation=le then ",<=," elseif relation=eq then ",=," elseif
           relation=ne then ",<>," elseif relation=ge then ",>=," elseif
           relation=gt then ",>," endif,
         expression: @Standard(),
         final: ")"
```

```
        endMethod
    ] < "", "", "L", "Last" >;

assign = name expression
    [ /classgen/
        Method Standard()
          initial: " ",
          name: @Standard() "=",
          expression: @Standard(),
          final:
        endMethod
    ] < "", "", "=", "Assign" >;

call = name callParm*
    [ /classgen/
        Method Standard()
          { comma:integer = 0 }
          initial: " @",
          name: @Standard() "(",
          callParm: if comma then "," endif @Standard() comma=1,
          final: ")"
        endMethod
    ] < "", "", "C", "Call" >;

callParm
    [ /classgen/
        Method Standard()
          initial:,
          final:
        endMethod
    ] < "", "", "P", "callParm" >;

labels = begTag endTag menuTag menuLabel
    [ /classgen/
        Method Standard()
          initial: @lm+(5) "<",
          begTag,endTag,menuTag: @Standard() ",",
          menuLabel: @Standard(),
          final: " >"
        endMethod
    ] < "labels", "", "L", "Labels" >;
```

```
begTag
   [ /classgen/
       Method Standard()
         initial: """",
         final: """"
       endMethod
   ] < "", "", "B", "BegTag" >;

endTag
   [ /classgen/
       Method Standard()
         initial: """",
         final: """"
       endMethod
   ] < "", "", "E", "EndTag" >;

menuTag
   [ /classgen/
       Method Standard()
         initial: """",
         final: """"
       endMethod
   ] < "", "", "T", "menuTag" >;

menuLabel
   [ /classgen/
       Method Standard()
         initial: """",
         final: """"
       endMethod
   ] < "", "", "L", "menuLabel" >;


nl =
   [ /classgen/
       Method Standard()
         initial: " @nl",
         final:
       endMethod
   ] < "", "", "N", "Newline" >;

vs =
    { count:integer = REQUIRED }
    [ /classgen/
        Method Standard()
          initial: " @vs(" count ")",
          final:
        endMethod
    ] < "", "", "V", "VertSpace" >;

today =
    [ /classgen/
        Method Standard()
          initial: " @today",
          final:
        endMethod
    ] < "", "", "T", "Today" >;

centre =
    [ /classgen/
        Method Standard()
          initial: " @centre",
          final:
        endMethod
    ] < "", "", "C", "Centre" >;

ecentre =
    [ /classgen/
        Method Standard()
          initial: " @ecentre",
          final:
        endMethod
    ] < "", "", "E", "eCentre" >;

leftJust =
    [ /classgen/
        Method Standard()
          initial: " @left",
          final:
        endMethod
    ] < "", "", "J", "justifyLeft" >;

rightJust =
    [ /classgen/
```

```
        Method Standard()
          initial: " @right",
          final:
        endMethod
    ] < "", "", "J", "justifyRight" >;

normal =
    [ /classgen/
        Method Standard()
          initial: " @normal",
          final:
        endMethod
    ] < "", "", "O", "nOrmal" >;

bold =
    [ /classgen/
        Method Standard()
          initial: " @bold",
          final:
        endMethod
    ] < "", "", "B", "Bold" >;

ebold =
    [ /classgen/
        Method Standard()
          initial: " @ebold",
          final:
        endMethod
    ] < "", "", "E", "eBold" >;

underline =
    [ /classgen/
        Method Standard()
          initial: " @underline",
          final:
        endMethod
    ] < "", "", "U", "Underline" >;

eunderline =
    [ /classgen/
        Method Standard()
          initial: " @eunderline",
        final:
      endMethod
  ] < "", "", "E", "eUnderline" >;

reverse =
  [ /classgen/
      Method Standard()
        initial: " @reverse",
        final:
      endMethod
  ] < "", "", "R", "Reverse" >;

ereverse =
  [ /classgen/
      Method Standard()
        initial: " @ereverse",
        final:
      endMethod
  ] < "", "", "E", "eReverse" >;

fillWord =
  [ /classgen/
      Method Standard()
        initial: " @fill_words",
        final:
      endMethod
  ] < "", "", "F", "fillWords" >;

fillChar =
  [ /classgen/
      Method Standard()
        initial: " @fill_chars",
        final:
      endMethod
  ] < "", "", "F", "fillChars" >;

lm =
  { operator:(increment, decrement, assign) = REQUIRED,
    width:integer = REQUIRED
  }
  [ /classgen/
      Method Standard()
```

```
        initial: " @lm" if operator=increment then "+" elseif
          operator=decrement then "-" else "=" endif "(" width ")",
        final:
      endMethod
   ] < "", "", "L", "LeftMarg" >;

rm =
   { operator:(increment, decrement, assign) = REQUIRED,
     width:integer = REQUIRED
   }
   [ /classgen/
      Method Standard()
        initial: " @rm" if operator=increment then "+" elseif
          operator=decrement then "-" else "=" endif "(" width ")",
        final:
      endMethod
   ] < "", "", "R", "RightMarg" >;

ll =
   { operator:(increment, decrement, assign) = REQUIRED,
     length:integer = REQUIRED
   }
   [ /classgen/
      Method Standard()
        initial: " @rm" if operator=increment then "+" elseif
          operator=decrement then "-" else "=" endif "(" length ")",
        final:
      endMethod
   ] < "", "", "K", "LineLength" >;
```

Appendix D

Sample Output

In this appendix we provide some examples of the output produced by the class descriptions in Appendix C.

D.1 A Simple Technical Document

Examples of the output generated by the `display` unparsing scheme of the class description of Appendix C.1 can be found throughout Chapter 5. Figure D.1 illustrates a fragment of output produced by the `latex` scheme of the same class description[1]. The same document fragment is shown in Figure 5.10 on page 81, while the output produced by the LaTeX formatter (using Figure D.1 as input), can be seen on page 81.

Figure D.2 shows the same document fragment in GML, while Figure D.3 shows the result if formatted with Waterloo SCRIPT GML [Water86].

[1]The `latex` scheme of Appendix C.1 is actually a subset of the scheme that produced the output of Figure D.1 — not all objects present in Figure D.1 are defined in the class description of Appendix C.1. The same comment applies to the GML example.

```
of each of the strings that we now describe:
\begin{enumerate}
\item {\em Start label.} This string is used to mark the start of the
object in the structure window and is put as close as possible to the
start of the object (see Section~\ref{strdisp}). In Figure~\ref{strings}
{\tt OList}, {\tt Item}, and {\tt para} are all start labels. The null
string ({\tt ""}) will cause no label to be displayed.
\item {\em End label.} This string is used to mark the end of the object
in the structure window, and is put as close as possible to the end of the
object (see Section~\ref{strdisp}). In Figure~\ref{strings} {\tt eOList}
and {\tt eItem}, are end labels. The null string will cause no label to
be displayed (as for example with the {\tt paragraph} object).
\item {\em Menu key.} This single character indicates which key will
select this object class name when the name is displayed in the create
menu.
\end{enumerate}
```

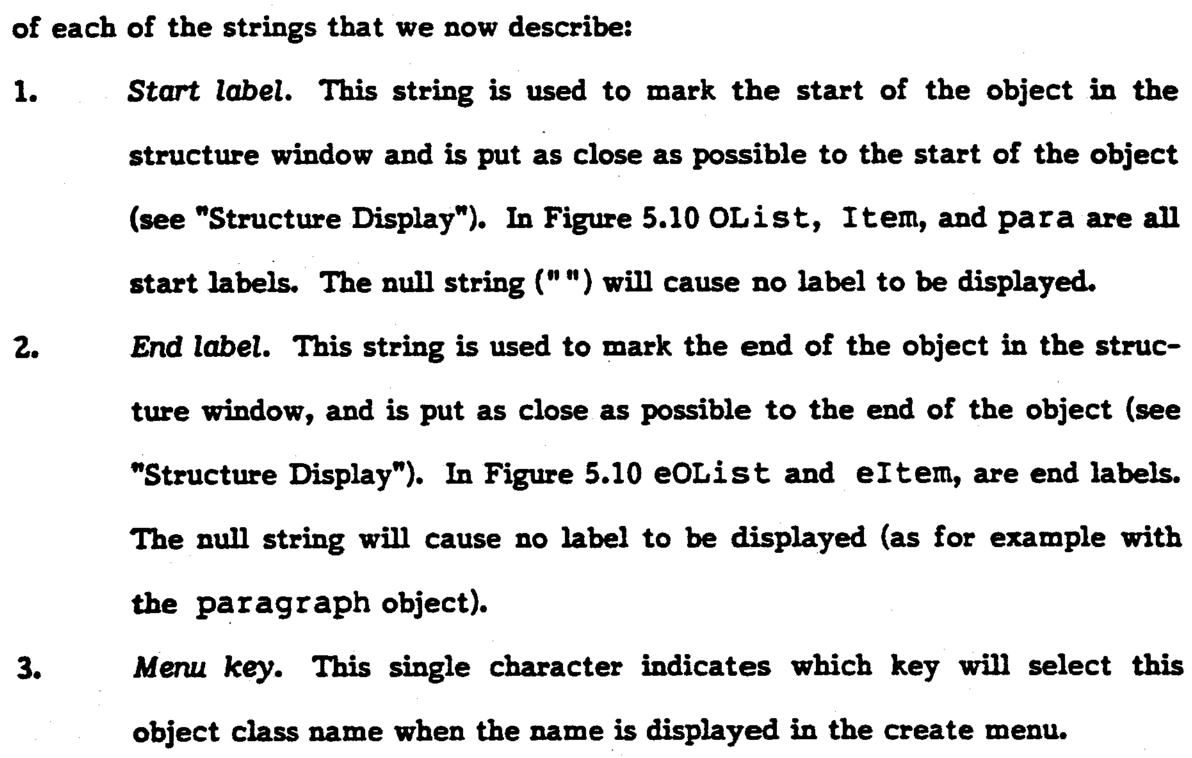
Figure D.1: LaTeX *code generated by the* `latex` *scheme for a technical document*

D.2 A Business Letter

This section illustrates the output generated by the unparsing schemes of the business letter class of Appendix C.2. Figures D.4 and D.5 show the output generated from a sample letter using the `gml` and `LaTeX` schemes respectively, while Figures D.6 and D.7 show the output from the two formatters.

```
of each of the strings that we now describe:
:OL.
:LI.
:HP1.Start label.:eHP1.  This string is used to mark the start of the
object in the structure window and is put as close as possible to the
start of the object (see
:HDREF refid=strdisp.).  In
:FIGREF refid=strings. :HPO.OList:eHPO., :HPO.Item:eHPO., and
:HPO.para:eHPO. are all start labels.  The null string
(:HPO.:Q.:eQ.:eHPO.) will cause no label to be displayed.
:LI.
:HP1.End label.:eHP1.  This string is used to mark the end of the object
in the structure window, and is put as close as possible to the end of the
object (see
:HDREF refid=strdisp.).  In
:FIGREF refid=strings. :HPO.eOList:eHPO. and :HPO.eItem:eHPO., are end
labels.  The null string will cause no label to be displayed (as for
example with the :HPO.paragraph:eHPO. object).
:LI.
:HP1.Menu key.:eHP1.  This single character indicates which key will
select this object class name when the name is displayed in the create
menu.
:eOL.
```

Figure D.2: *GML code generated by* `gml` *scheme for technical document*


of each of the strings that we now describe:

1. *Start label.* This string is used to mark the start of the object in the structure window and is put as close as possible to the start of the object (see "Structure Display"). In Figure 5.10 `OList`, `Item`, and `para` are all start labels. The null string (" ") will cause no label to be displayed.
2. *End label.* This string is used to mark the end of the object in the structure window, and is put as close as possible to the end of the object (see "Structure Display"). In Figure 5.10 `eOList` and `eItem`, are end labels. The null string will cause no label to be displayed (as for example with the `paragraph` object).
3. *Menu key.* This single character indicates which key will select this object class name when the name is displayed in the create menu.

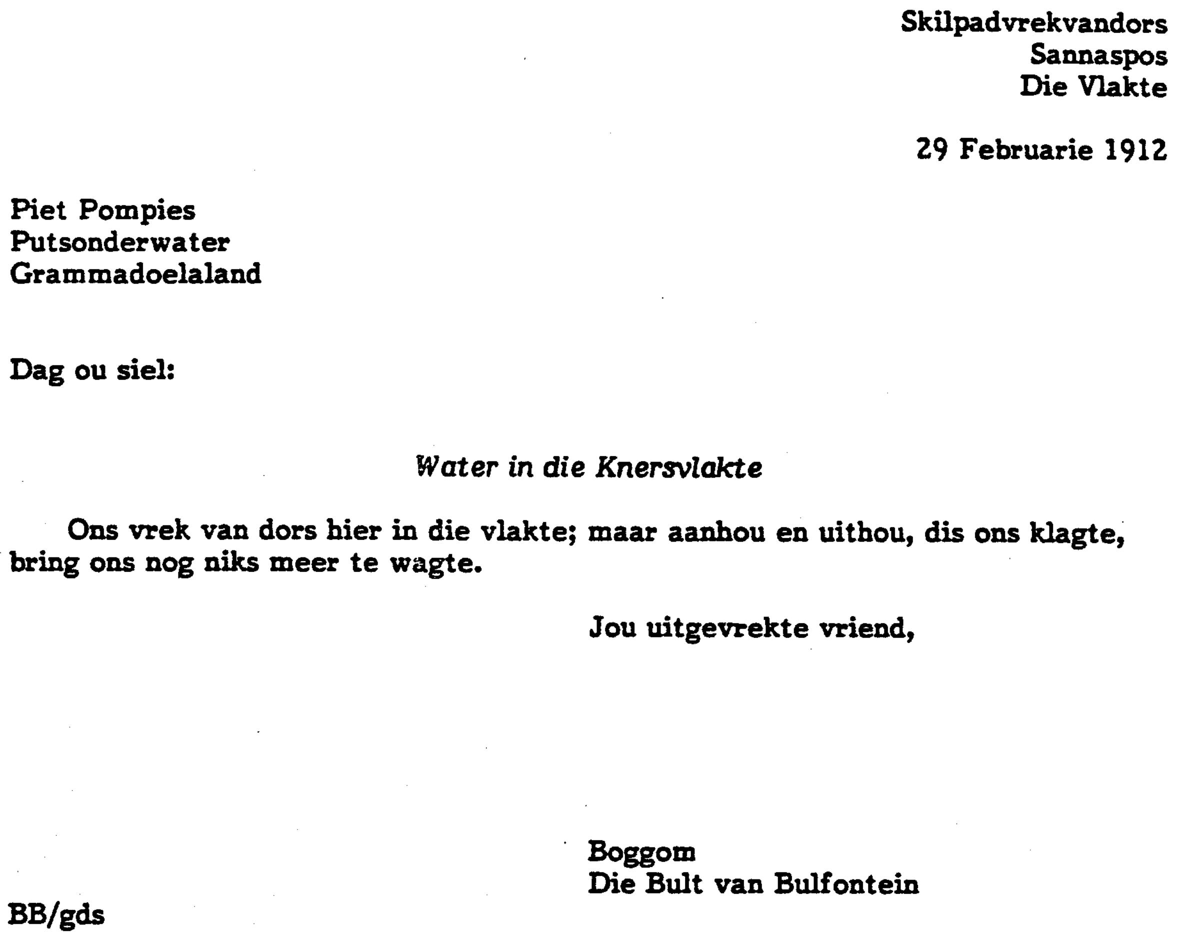
Figure D.3: *Output produced by the GML formatter*

```
:GDOC
:FROM.
Skilpadvrekvandors
Sannaspos
Die Vlakte
:DATE.29 Februarie 1912
:TO.Piet Pompies
Putsonderwater
Grammadoelaland
:OPEN.Dag ou siel
:SUBJECT.Water in die Knersvlakte
:P.
Ons vrek van dors hier in die vlakte; maar aanhou en uithou, dis ons
klagte, bring ons nog niks meer te wagte.
:CLOSE.Jou uitgevrekte vriend
Boggom
Die Bult van Bulfontein
:eCLOSE.BB/gds
:eGDOC
```

Figure D.4: *Output produced using the* `gml` *scheme for business letter*

```
\documentstyle{letter}
\begin{document}
\address{Skilpadvrekvandors\\
Sannaspos\\
Die Vlakte}
\renewcommand{\today}{29 Februarie 1912}
\begin{letter}{Piet Pompies\\
Putsonderwater\\
Grammadoelaland}
\opening{Dag ou siel}
{\bf Water in die Knersvlakte}

Ons vrek van dors hier in die vlakte; maar aanhou en uithou, dis ons
klagte, bring ons nog niks meer te wagte.
\signature{Boggom\\
Die Bult van Bulfontein}
\closing{Jou uitgevrekte vriend}
\ps BB/gds
\end{letter}
\end{document}
```

Figure D.5: *Output produced using the* LaTeX *scheme for business letter*



Skilpadvrekvandors
Sannaspos
Die Vlakte

29 Februarie 1912

Piet Pompies
Putsonderwater
Grammadoelaland

Dag ou siel:

Water in die Knersvlakte

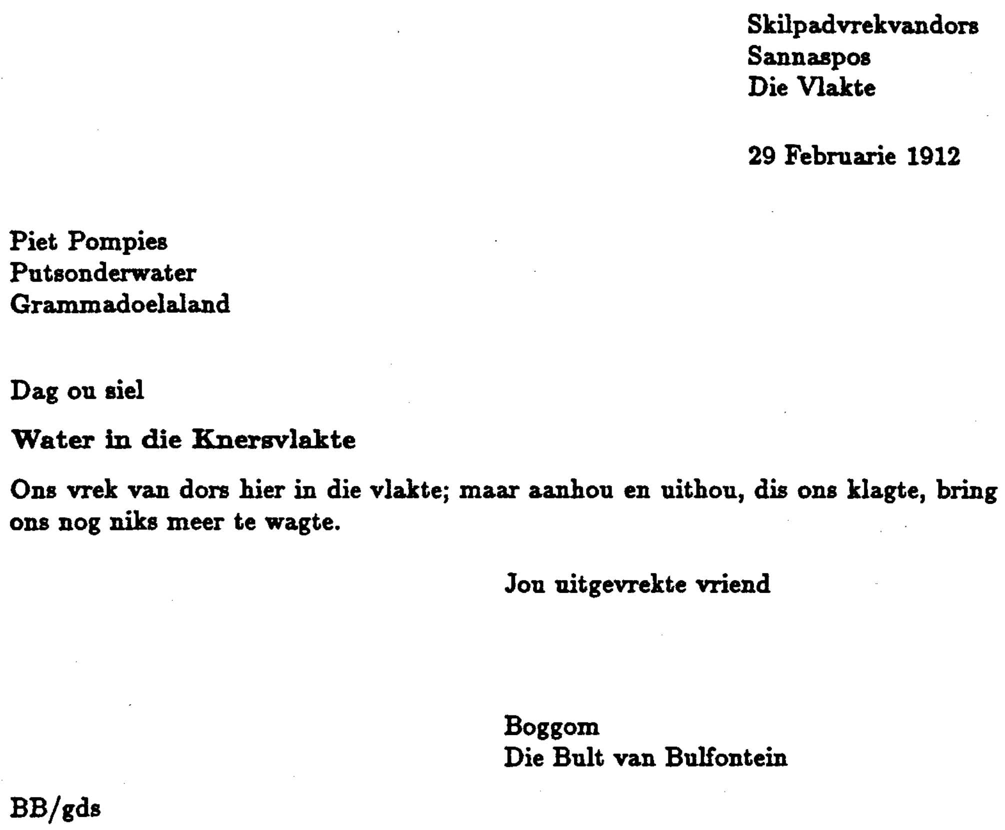
Ons vrek van dors hier in die vlakte; maar aanhou en uithou, dis ons klagte, bring ons nog niks meer te wagte.

Jou uitgevrekte vriend,

Boggom
Die Bult van Bulfontein

BB/gds

Figure D.6: *Business letter output from a GML formatter*



Skilpadvrekvandors
Sannaspos
Die Vlakte

29 Februarie 1912

Piet Pompies
Putsonderwater
Grammadoelaland

Dag ou siel

Water in die Knersvlakte

Ons vrek van dors hier in die vlakte; maar aanhou en uithou, dis ons klagte, bring ons nog niks meer te wagte.

Jou uitgevrekte vriend

Boggom
Die Bult van Bulfontein

BB/gds

Figure D.7: *Business letter output from* LaTeX *formatter*

Appendix E

Command Summary

This is a summary of all the commands available to the user of the document preparation system. Some commands are tied to function or other special keys, and some appear in command menus. For those that are tied to function keys, we have listed the key name (for an IBM-PC keyboard) in parenthesis after the command name.

E.1 Control

E.1.1 Edit Structure (TAB)

Allow structural editing of the document. The current object is highlighted and the create menu is displayed and activated. The menu contains all objects that may be inserted in the document after the current object.

E.1.2 Edit Content (TAB)

Allow textual (content) editing of the document. The create menu is removed and the user is allowed to edit the text associated with the basic document objects.

E.1.3 Command Menu (Esc)

Display and activate the command menu. The command menu contains standard word-processing commands such as filing, search and replace, directory maintenance and copying commands.

E.1.4 Edit Attributes (F9)

Enable the editing of the current object's attributes, if it has any. A menu listing all the attributes, the values they may have, and their current values are displayed and the user is allowed to change it through selection and/or entering new values.

E.1.5 Patch

Edit the patch area. A non-restrictive grammar is used to edit the content of the patch area.

E.1.6 Exit Patch

This command terminates editing of the patch area.

E.1.7 Insert/Overstrike (F5)

Flip between inserting/overstriking characters when editing text.

E.1.8 Extend (F6)

Anchor the cursor at its current position and allow it to be enlarged with the use of the cursor movement keys.

E.1.9 Help (F10)

Whenever this command is issued, on-line, context-sensitive help is displayed in a pop-up help window.

E.2 Cursor Movement (Structure)

E.2.1 Previous (↑)

Position the cursor on the previous sibling of the current object if one is defined according to the current unparsing scheme. If the current object is the leftmost sibling and the class grammar allows the current object to have an older sibling, position the cursor on the phantom first child of the current object's parent. If no older sibling is allowed, position the cursor on the previous object in the pre-order traversal of the document tree. If the current object is the root of the document tree, this command has no effect.

E.2.2 Next (↓)

Position the cursor on the next sibling of the current object if one is defined according to the current unparsing scheme. If no sibling is defined, position the cursor on the next object in the pre-order traversal of the document tree (i.e. the next sibling of the parent of the current object, recursively). If the current object is the rightmost in the document tree (as defined in pre-order), this command has no effect.

E.2.3 Parent (←)

Position the cursor on the parent of the current object.

E.2.4 Child (→)

Position the cursor on the first child of the current object. If a phantom first child is allowed, position the cursor on the phantom child.

E.2.5 Root (Ctrl-Home)

Position the cursor on the root of the document tree.

E.2.6 Oldest Sibling (Home)

Position the cursor on the current object's oldest sibling.

E.2.7 Youngest Sibling (End)

Position the cursor on the current object's youngest sibling.

E.3 Cursor Movement (Content)

In text mode, the cursor can only be positioned on text that the user has typed in, and only this text can therefore be edited. The one exception to this rule is that the cursor may be positioned in the single column beyond the end of the text of a basic object, or on the placeholder for an empty basic object.

E.3.1 Character (F1)

Set the cursor size to one character.

E.3.2 Word (F2)

Set the cursor size to a word.

E.3.3 Sentence (F3)

Set the cursor size to a sentence.

E.3.4 Block (F4)

Set the cursor size to a "text block". A text block is the unit of text that is associated with a basic object.

E.3.5 Up (↑)

Move the cursor up one line, or, if in sentence or block mode, to the previous sentence/block.

E.3.6 Down ($\downarrow$)

Move the cursor down one line, or, if in sentence or block mode, to the next sentence/block.

E.3.7 Left ($\leftarrow$)

Position the cursor on the previous character, word, sentence or block.

E.3.8 Right ($\rightarrow$)

Position the cursor on the next character, word, sentence or block.

E.3.9 Start of Line (Home)

Position the cursor at the start of the line.

E.3.10 End of Line (End)

Position the cursor at the end of the line.

E.3.11 Screen Up (PgUp)

Move the screen window a screen full of lines towards the start of the document and position the cursor at the same relative position on the screen.

E.3.12 Screen Down (PgDn)

Move the screen window a screen full of lines towards the end of the document and position the cursor at the same relative position on the screen.

E.4 Cursor Movement (Structure and Content)

The following two commands are convenient to use to reposition the cursor on the screen while still pointing to the same position in the document, as well as to examine the full extent of the cursor if it covers an area larger than the screen.

E.4.1 Scroll Up (Ctrl ←)

Move the screen window up one line in the document, while keeping the cursor at its current position in the document. The top of the cursor is not allowed to scroll off the bottom of the screen.

E.4.2 Scroll Down (Ctrl →)

Move the screen window down one line in the document, while keeping the cursor at its current position in the document. The bottom of the cursor is not allowed to scroll off the top of the screen.

E.5 Structure Editing

E.5.1 Copy

The part of the document currently "covered" by the cursor is copied to the patch area, overwriting the previous content of the patch area. The document remains unchanged.

E.5.2 Delete (Del)

The part of the document currently covered by the cursor is deleted and saved in the patch area, overwriting the previous content of the patch area.

E.5.3 Insert (Ins)

The current content of the patch area is inserted in front of the cursor. The command has no effect if its execution will result in a document that is not partially correct.

E.5.4 Nest (F8)

The currently marked objects (which have to be siblings), are made the children of a single composite object that replaces them in the document tree. The user is prompted with a create menu to indicate the class of the new parent. The command has no effect if it cannot be performed while still maintaining a partially correct document.

E.5.5 Extract (F7)

The opposite of Nest. The current composite object (the cursor must be positioned on one, thus marking a subtree in the document) is replaced by its children. The command has no effect if it cannot be performed while still maintaining a partially correct document.

E.5.6 Split

The parent of the current object is split at the current object, creating two new objects of the same class as the parent. The older siblings of the current object become the children of the older of the two new objects, while the current object and its younger siblings become the children of the other new object. This second new object becomes the new current object. The command has no effect if it cannot be performed while still maintaining a partially correct document.

E.5.7 Join

The current object and its older sibling are joined into one object that has both their children. The user is prompted with a create menu to indicate the class of the new parent. The command has no effect if there is no older sibling, or if it cannot be performed while still maintaining a partially correct document.

E.5.8 Change Class

Change the class of the current object. The user is presented with a menu that lists the classes into which the object may be changed. The command has no effect if there are no legal options.

E.6 Content Editing

E.6.1 Backspace ($\longleftarrow$)

Delete the character preceding the cursor. The command has no effect if that character belongs to a different text block.

E.6.2 Copy

The text currently covered by the cursor is copied to the patch area, overwriting the previous content of the patch area. The document remains unchanged. Text is copied to the patch area as basic objects. These basic objects become siblings of each other in the patch area.

E.6.3 Delete (Del)

Delete the text covered by the cursor and save it in the patch area, overwriting the previous content of the patch area. Text is copied to the patch area as basic objects. These basic objects become siblings of each other in the patch area. Note that only the text of basic objects can be deleted. No objects are deleted, with the exception of basic objects that are already empty.

E.6.4 Insert (Ins)

The text of all the basic objects in the patch area is inserted as one text string into the current basic object in front of the cursor.

E.6.5 Search

Allows the user to search for occurrences of a text string in the document.

E.6.6 Replace

Allows the user to search for occurrences of a text string and replace them with another string.

Appendix F

Some Structure Editing Algorithms

The algorithms presented here implement the structure editing commands of Appendix E.5. We use the following notational conventions: a_p is the current object, while $a_1 ... a_p ... a_m$ are the children of the current object's parent.

We make use of the functions *CONSTRUCT_MENU* and *TEST_CORRECT* described in Chapter 4, as well as the following functions that we do not describe any further as their names are self-explanatory:

PARENT_OF,

CHILDREN_OF,

CLASS_OF, and

CONCAT

The function *TEST_SUBTREE* is used in the insert algorithm and is defined in Figure F.1.

```
$TEST\_SUBTREE( s )$
INPUT:
    $s$ - any document object
OUTPUT:
    the value $TRUE$ if the object and its constituents recursively conforms to their
    respective class descriptions
begin
    if $s$ is a basic object then
        return $TRUE$
    else
        $b_1...b_n := CHILDREN\_OF( s )$
        if( $TEST\_CORRECT(CLASS\_OF(s), b_1...b_n)$ )then
            for $i := 1$ to $m$ do
                if( $TEST\_SUBTREE(b_i) = FALSE$ )then
                    return $FALSE$
                endif
            endfor
        endif
    endif
    return $TRUE$
end
```

Figure F.1: *TEST_SUBTREE*

F.1 Insert

```
begin
    $b_1...b_n$ := content of patch area
    $class := CLASS\_OF(PARENT\_OF(a_p))$
    if( $TEST\_CORRECT(class, a_1...a_{p-1}b_1...b_na_p...a_m)$ )then
        for $i := 1$ to $m$ do
            if( $TEST\_SUBTREE(b_i) = FALSE$ )then
                return $FALSE$
            endif
        endfor
    endif
    insert $b_1...b_n$ in front of $a_p$
end
```

F.2 Nest

This algorithm assumes the user has marked $a_p...a_q$, a substring of $a_1...a_m$, and returns a menu of object classes, any one of which can be used to nest the marked objects.

```
begin
    $class := CLASS\_OF(PARENT\_OF(a_p))$
    $Menu := CONSTRUCT\_MENU(class, a_1...a_{p-1}a_{q+1}...a_m, p-1)$
    for each class $class \in Menu$ do
        if( $TEST\_CORRECT(class, a_p...a_q) = FALSE$ )then
            remove $class$ from $Menu$
        endif
    endfor
    return $Menu$
end
```

F.3 Extract

```
begin
    $class := CLASS\_OF(PARENT\_OF(a_p))$
    if( $TEST\_CORRECT(class, CHILDREN\_OF(a_p))$ )then
        replace $a_p$ with its children
    endif
end
```

F.4 Split

```
begin
    $b_q := PARENT\_OF(a_p)$
    $b_1 ... b_q ... b_n := CHILDREN\_OF(PARENT\_OF(b_q))$
    if( $TEST\_CORRECT(CLASS\_OF(PARENT\_OF(b_p)), b_1 ... b_p b_p ... b_n)$ )then
        create a node $s$ of class $CLASS\_OF(b_p)$
        assign $a_p ... a_m$ to $s$ as children
        insert $s$ after $b_p$
    endif
end
```

F.5 Join

```
begin
    $class := CLASS\_OF(PARENT\_OF(a_p))$
    $Menu := CONSTRUCT\_MENU(class, a_1 ... a_{p-2} a_{p+1} ... a_m, p - 2)$
    $b_1 ... b_n := CONCAT(CHILDREN\_OF(b_{p-1}), CHILDREN\_OF(b_p))$
    for each class $class \in Menu$ do
        if( $TEST\_CORRECT(class, b_1 ... b_n) = FALSE$ )then
            remove $class$ from $Menu$
        endif
    endfor
    return $Menu$
end
```

F.6 Change Class

```
begin
    $class := CLASS\_OF(PARENT\_OF(a_p))$
    $Menu := CONSTRUCT\_MENU(class, a_1 ... a_{p-1} a_{p+1} ... a_m, p - 1)$
    $b_1 ... b_n := CHILDREN\_OF(a_p)$
    for each class $class \in Menu$ do
        if( $TEST\_CORRECT($ class, $b_1 ... b_n) = FALSE$ )then
            remove $class$ from $Menu$
        endif
    endfor
    return $Menu$
end
```

Bibliography

[Adere84] Aderet, A., and Hoffman, P. Mnemonic strategies in word processing systems, *Proc. Second ACM SIGOA Conf. Office Info. Systems*, *SIGOA Newsletter* **5**, 1–2 (June 1984), pp 188–198.

[Aho74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. 1977.

[Adobe85] Adobe Systems, Inc., *PostScript Language Reference Manual*, Addison-Wesley, Reading, Mass. 1985.

[Aho86] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass. 1986.

[Allen81] Allen, T., Nix, R., and Perlis, A. PEN: A hierarchical document editor, *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation*, *SIGPLAN Notices* **16**, 6 (June 1981), pp 74–81.

[Beach85] Beach, R. J. *Setting tables and illustrations with style*, PhD dissertation, Dept. of Computer Science, Univ. of Waterloo, Ontario, 1985.

[Boswe82] Boswell, F. D. *Waterloo Systems Language: Tutorial and Reference Guide*, WATFAC Publications, Waterloo, Ontario, 1982.

[Brade81] Brader, M. S. *An incremental text formatter*, Research Rep. CS-81-12, Dept. of Computer Science, Univ. of Waterloo, Ontario, 1981.

[Campb84] Campbell, R. H., and Kirslis, P. A. The SAGA project: a system for software development, *SIGPLAN Notices* **19**, 5 (May 1984), pp 73–80.

[Coray86] Coray, G., Ingold, R., and Vanoirbeek, C. Formatting structured documents: batch versus interactive?, In *Proc. Int. Conf. Text Processing and Document Manipulation*, van Vliet, J. C. (ed.), Cambridge University Press, April 1986, pp 154–170.

[Chamb82] Chamberlin, D. D., et al. JANUS: An interactive document formatter based on declarative tags, *IBM Syst. J.* **21**, 3 (1982), pp 250–271.

[Feile83] Feiler, P. H., and Kaiser, G. E. Display-oriented structure manipulation in a multi-purpose system, *Proc. IEEE Comp. Soc. Seventh International Computer Software and Applications Conference* (COMPSAC 83), November 1983, pp 40–48.

[Feile86] Feiler, P. H., Fahimeh, J., and Schlichter, J. H. An interactive prototyping environment for language design, *Proc. Nineteenth Annual Hawaii International Conference on System Sciences*, Vol II, 1986, pp 106–116.

[Furut82] Furuta, R., Scofield, J., and Shaw, A. Document formatting systems, *Computing Surveys* **14**, 3 (September 1982), pp 417–472.

[Furut86] Furuta, R. An integrated, but not exact-representation, editor/formatter, In *Proc. Int. Conf. Text Processing and Document Manipulation*, van Vliet, J. C. (ed.), Cambridge University Press, April 1986, pp 246–259. Also: Tech. Rep. 86-09-08 (Ph.D. Thesis), Dept. of Computer Science, Univ. of Washington, Seattle, September 1986.

[Gansn83] Gansner, E. R., et al. SYNED - A language-based editor for an interactive programming environment, *COMPCON Spring 83*, IEEE Computer Society, 1983, pp 406–410.

[Ginsb77] Ginsberg, S. *The Mathematical Theory of Context Free Languages*, McGraw-Hill, New York, 1977, p108.

[Hamme81] Hammer, M., et al. The implementation of Etude, an integrated and interactive document production system, *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation*, *SIGPLAN Notices* **16**, 6 (June 1981), pp 74–81.

[Hodge85] Hodgson, G. M., and Ruth, S. R. The use of menus in the design of on-line systems: a retrospective view, *ACM SIGCHI Bulletin* **17**, 1 (July 1985), pp 16–22.

[Hopcr79] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass. 1979.

[Horak84] Horak, W., and Kronert, B. An object-oriented office document architecture model for processing and interchange of documents, *Proc. Second ACM SIGOA Conf. Office Info. Systems*, *SIGOA Newsletter* **5**, 1–2 (June 1984), pp 152–160.

[IBM84a] *Document Composition Facility: General Markup Language Concepts and Design Guide* (Second Edition), Document No. SH20-9188-1, IBM, White Plains, N.Y., January 1984.

[IBM84b] *Document Composition Facility: General Markup Language Starter Set User's Guide* (Third Edition), Document No. SH20-9186-2, IBM, White Plains, N.Y., August 1984.

[IBM85] *Document Composition Facility: SCRIPT/VS Language Reference* (Release 2), Document No. SH35-0070-2, IBM, White Plains, N.Y., March 1985.

[ISO85] Information Processing Systems - Text Preparation and Interchange - Processing and Markup Languages - Part Six: Generic Document Representation Specification (SGML), Draft Proposal International Standard ISO/DP 8879/6, ISO TC97/SC18/WG8, January 1985.

[Johns85] Johnson, J. H. INR: A Program for computing finite automata, Unpublished report, University of Waterloo, Ontario, September 1985.

[Kazma86] Kazman, R. *Structuring the Text of the Oxford English Dictionary through Finite State Transduction*, MMath thesis, (also Data Structuring Group report CS-86-20), University of Waterloo, Ontario, June 1986.

[Kimur84a] Kimura, G. D., and Shaw, A. C. The structure of abstract document objects, *Proc. Second ACM SIGOA Conf. Office Info. Systems, SIGOA Newsletter* **5**, 1–2 (June 1984), pp 161–169.

[Kimur84b] Kimura, G. D. *A structure editor and model for abstract document objects*, Tech. Rep. 84-07-04, Dept. of Computer Science, Univ. of Washington, Seattle, July 1984.

[Kimur86] Kimura, G. D. A structure editor for abstract document objects, *IEEE Trans. on Software Engineering*, **SE-12**, 3 (March 1986), pp 417–435.

[King86] King, P. R. An overview of the W document preparation system, In *Proc. Int. Conf. Text Processing and Document Manipulation*, van Vliet, J. C. (ed.), Cambridge University Press, April 1986, pp 154–170.

[Knuth81] Knuth, D. E., and Plass, F. P. Breaking paragraphs into lines, *Softw. Prac. Exper.* **11**, 11, (Nov. 1981), pp 1119–1184.

[Knuth84] Knuth, D. E. *The TEXbook*, Addison-Wesley, Reading, Mass. 1984.

[Lampo86] Lamport, L. LATEX*: A Document Preparation System*, Addison-Wesley, Reading, Mass. 1986.

[Medin82] Medina-Mora, R. *Syntax-directed editing: Towards integrated programming environments*, PhD Thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania, 1982.

[Meyro82] Meyrowitz, N., and van Dam, A. Interactive editing systems: Part II, *Computing Surveys* **14**, 3 (September 1982), pp 353–415.

[Morri81] Morris, J. M., and Schwartz, M. D. The design of a language-based editor for block-structured languages, *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation*, *SIGPLAN Notices* **16**, 6 (June 1981), pp 28–33.

[Morri85] Morris, R. A. Is what you see enough to get? A description of the Interleaf Publishing System, In *Proc. Second Int. Conf. on Text Processing Systems* (PROTEXT II), Miller, J. J. H. (ed.), Boole Press, October 1985, pp 56–81.

[Ossan76] Ossana, J. F. *NROFF/TROFF user's manual*, Computer Science Tech. Rep. 54, Bell Laboratories, Murray Hill, N.J., October 1976.

[Quint83] Quint, V. An interactive system for mathematical text processing, *Technology and Science of Informatics* **2**, 3 (1983), pp 169–179.

[Quint84] Quint, V. Interactive editing of mathematics, In *Proc. First Int. Conf. on Text Processing Systems* (PROTEXT I), Miller, J. J. H. (ed.), Boole Press, October 1984, pp 55–68.

[Quint86] Quint, V., and Vatton, I. Grif: an interactive system for structured document manipulation, In *Proc. Int. Conf. Text Processing and Document Manipulation*, van Vliet, J. C. (ed.), Cambridge University Press, April 1986, pp 154–170.

[Reid80] Reid, B. K. *Scribe: a document specification language and its compiler*, Report CMU-CS-81-100, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, October 1980.

[Reps84] Reps, T., and Teitelbaum, T. The synthesizer generator, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, *SIGPLAN Notices* **19**, 5 (May 1984), pp 42–48.

[Saltz65] Saltzer, J. Manuscript typing and editing: TYPESET, RUNOFF. In *The Compatible Time-Sharing System: A programmer's guide*, 2nd ed., Crisman, P. A. (ed.), The M.I.T Press, Cambridge, Mass., 1965, Sec. AH.9.01.

[Shaw78] Shaw, A. C. Software descriptions with flow expressions, *IEEE Trans. on Software Engineering*, **SE-4**, 3 (May 1978), pp 242–254.

[Shaw80] Shaw, A. C. *A model for document preparation systems*, Tech. Rep. 80-04-02, Dept. of Computer Science, Univ. of Washington, Seattle, April 1980.

[Stand80] Standish, T. A. *Data Structure Techniques*, Addison-Wesley, Reading, Mass. 1980.

[Teite81] Teitelbaum, T., and Reps, T. The Cornell Program Synthesizer: a syntax-directed programming environment, *Comm. ACM* **24**, 9 (September 1981), pp 563–573.

[vanHu85] van Huu, L. SGML: A standard language for text description, In *Proc. Second Int. Conf. on Text Processing Systems* (PROTEXT II), Miller, J. J. H. (ed.), Boole Press, October 1985, pp 198–212.

[Water86] *Waterloo SCRIPT GML User's Guide*, Department of Computing Services, University of Waterloo, Ontario, April 1986.

[Water85] *Waterloo SCRIPT Reference Manual*, Department of Computing Services, University of Waterloo, Ontario, 1985.

[Wiswe85] Wiswell, P. Word processing: the last word, *PC Magazine* **4**, 17 (August 1985), pp 110–134.

[Woods70] Woods, W. A. Transition networks for natural language analysis, *Comm. ACM* **13**, 10 (October 1970), pp 591–606.

[Zloof75] Zloof, M. M. Query By Example, *Proc. NCC* **44** (May 1975).