

Introduction

Many software applications involve some form of editing: a user views a data structure and provides edit gestures in order to modify this data structure. Different kinds of documents require different ways of editing, and hence a multitude of editors exists, each having its own specific edit model and user-interface conventions. Moreover, since application designers have different ideas on what constitutes a pleasant edit model, even editors for the same document type may show significantly different edit behavior. Nevertheless, the core edit behavior, whether performed in a word-processor or a spreadsheet, is largely similar: document fragments may be copied and pasted, and new parts of the document may be constructed by selecting from menus or entering text.

An obvious research question is to abstract from the specific aspects of each editor and construct a generic system that can be instantiated to a specific editor application. Building an editor with such a system would require only a fraction of the amount of engineering required to build an editor from scratch. A generic editor enhances consistency between editors, because all instantiated editors share the same edit model, and, furthermore, it facilitates the integration of editors for different document types.

Especially in the nineteen-eighties, many research projects on structure editing were started. However, the editors developed were generally perceived as being overly restrictive, and attempts at developing less restrictive systems resulted mainly in text-only editors. Further, regardless of the restrictiveness of the edit model, the applicability of the generic editors was generally limited to source editors for programming languages and simple word-processing applications. In the years following, research interest in structure editing steadily declined, and many of the generic editors that were developed are now used only for educational purposes at the institute of origin.

In our opinion, the problem with most of these structure editors is that they either focus on editing the document structure, or the presentation (often just text). The document-

oriented editors may have a powerful presentation mechanism, but poor editing support in the presentation, which results in a restrictive edit model. On the other hand, purely presentation-oriented editors lack edit operations on the document, and have relatively weak presentation mechanisms.

With the increasing popularity of the XML format for representing structured documents, the advantages of a powerful generic editor are becoming even more apparent. Many XML document types are being developed, but support for editing documents of these types is still poor. There is a choice between using an expensive custom-made editor, or a generic XML editor, but the functionality of the latter does not come close to what a presentation-oriented (WYSIWYG) editor could potentially offer. It is, for instance, not possible to use any of the current XML editors as a convenient editor for a programming language or for mathematical equations.

In this thesis we investigate whether and how the advantages of structure editing and a powerful presentation formalism can be combined with a non-restrictive presentation-oriented edit model. The result of this research is the presentation-oriented structure editor Proxima. Before we introduce Proxima, we discuss the basic concepts that play a role in editing structured documents. In Section 1.2 we introduce the Proxima editor, followed by a summary of the introduced terminology in Section 1.3 and an overview of the thesis in Section 1.4.

1.1 Preliminaries

1.1.1 Structured documents

A *structured document* is a collection of logical entities between which a structural relation exists. Examples of structured documents are HTML pages, program sources, word processor documents, etc.

In this thesis, we restrict ourselves to structured documents that have a tree structure that can be described by an EBNF grammar. Although graphs can be viewed as structured documents as well, algorithms for performing computations over graphs are far more complex than tree algorithms. Furthermore, parsers for graphs are less well understood than parsers for trees, as well as computationally more expensive.

In cases where we explicitly want to describe the structure of a document fragment, we use monomorphic (i.e. parameter free) Haskell [70] data types together with the list type. Example document fragments are represented by Haskell terms. For example, a document representing the let expression “let $x = 1$; $y = 2$ in $x + y$ ” can be denoted in Haskell by:

Let [*Decl* (*Ident* “x”) (*Int* 1), *Decl* (*Ident* “y”) (*Int* 2)] (*Sum* (*Ident* “x”) (*Ident* “y”))

1.1.2 XML

The eXtensible Markup Language XML [16] is an increasingly popular standard for representing structured documents. The standard is a simplified descendant of SGML [38] (Standard Generalized Markup Language). An XML document is a sequence of characters that encodes a tree structure. The nodes of the tree are referred to as *elements*. The leaves of the tree are text or attributes (name-value pairs describing properties of an element). The structure of the tree is represented with opening and closing *tags*, and if these tags are nested correctly, the XML document is *well-formed*.

The let expression example of the previous section can be represented in XML by:

```
<Let><Decl><Ident>x</Ident><Int>1</Int></Decl>
  <Decl><Ident>y</Ident><Int>2</Int></Decl>
  <Sum><Ident>x</Ident><Ident>y</Ident></Sum></Let>
```

The type of an XML document can be specified in several formalisms. The *Document Type Definition* (DTD) is part of the XML specification, and basically describes an EBNF grammar over XML elements. A much more expressive formalism is XML Schema [87], which itself is a sublanguage of XML. Compared to the DTD formalism, the advantages of using a Schema definition include more control over textual content, as well as a form of inheritance. If an XML document conforms to a certain DTD or Schema, it is called *valid*. A third standard, which is not as common as DTDs or Schemas, is the Relax NG standard [23]. Relax NG is a combination of Relax [55] and TREX [22], and is based on regular expressions.

The number of standards for sublanguages of XML, also referred to as dialects, is rapidly growing. Besides the already mentioned XML Schema, we provide a few more examples.

The Mathematical Markup Language MathML [20] is a standard for describing mathematical equations and expressions. Technical documentation can be represented with the DocBook [92] standard, which exists for XML as well as for SGML. The standard can also be used for papers and books. Finally, we mention the XHTML [2] standard, which is an XML encoding of HTML. Although similar, an HTML document is not necessarily an XML document, since HTML is a dialect of SGML rather than XML.

1.1.3 Editing

While the term *editor* is usually only associated with plain-text editors such as Emacs [81] or the ubiquitous Microsoft Notepad, we will use the term in a much broader sense. We regard as an editor any application that presents a visual representation of an internal data structure to a user and allows the user to modify this structure. The internal data structure is referred to as the *document* and the visual representation is the *presentation*.

Obviously, word processors, image editors, and text editors are editors in this view, but there are also some less obvious examples. Take, for example, the preferences pane that is

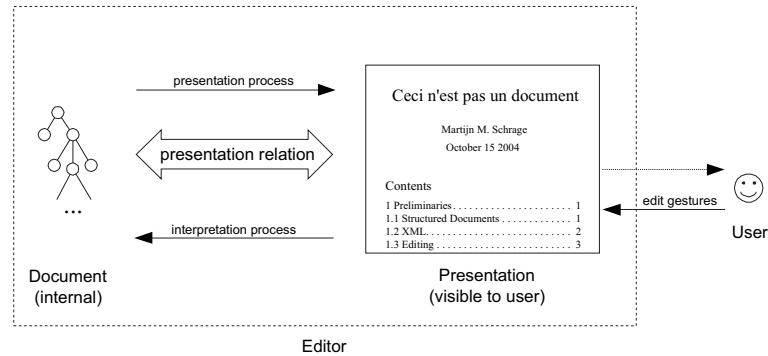


Figure 1.1: Schematic representation of an editor.

part of most window-based applications. The check boxes, selection lists, and text fields can be seen as a presentation of the preferences of the application. Another example of an application that is not usually regarded as an editor is a file browser. (For a description of a file browser as a text editor, see for example Fraser [28].)

Figure 1.1 contains a schematic representation of an editor. The main data structures in the editor (also referred to as *levels*) are the internal *document* on the left that is not visible to the user and the user-visible *presentation* on the right. The document should not be confused with a file, which is a representation of the document that is stored on a file system. Furthermore, we also do not consider an XML source to be a document, but rather a textual presentation of the internal document.

A *presentation*, or *view*, is the only thing a user sees of the document. A presentation may be textual, graphical, or a combination of both. We focus on static presentations only. Hence, we do not explicitly consider presentations containing sounds or animations, unless presented statically (e.g. as a textual link to a sound or video file). In the presentation, the editor shows the focus of attention, or, for brevity, just *focus*, which is a shared name for the selection as well the cursor (which is an empty selection). Several presentations of a single document may be shown simultaneously by the editor, each having its own focus. Finally, if a presentation closely mirrors the final physical appearance of the document when it is printed, it is referred to as a WYSIWYG presentation (What You See Is What You Get).

The relation between a document and its presentation is denoted by the term *presentation relation*, or *presentation mapping*. If, according to the presentation relation, the presentation shown to the user is a presentation of the document, we say that the *presentation invariant* holds. Computing the presentation of a document is called the *presentation process*, whereas computing a document from a presentation is called the *interpretation process*. Together, the two processes implement the presentation relation and maintain the presentation invariant if either side of the relation changes.

A presentation is *valid* if there exists a document, which, when presented, yields that presentation. A presentation for which there is no corresponding document is invalid. An invalid presentation may result from an editing the presentation level. Note the difference with the term valid document, which denotes a document that is well typed.

The presentation relation for an editor may be (partially) specified in a style sheet, or *presentation sheet*. A presentation sheet describes how elements of the document type are to be presented, and is a parameter of the presentation process. By modifying the sheet, a user may influence the appearance of the document without having to modify the editor itself. A presentation sheet can be regarded as a parameter to the interpretation process as well, since the interpretation depends on the presentation specified in the sheet. Examples of style-sheet formalisms are the Cascading Style Sheets (CSS) [11] for HTML as well as XML, and the Extensible Stylesheet Language (XSL) [1] for XML.

Generally speaking, editing consists of repeated interactive cycles of presenting and interpreting. The editor shows a presentation of the document together with the current focus to the user. The user then provides the editor with an *edit gesture*, such as a key press or a mouse movement, which is interpreted as an update on the document. The document is then re-presented and shown to the user. The process is repeated until the user quits the editor. Chapters 4 and 5 provide a more formal description of the editing process.

Document-oriented versus presentation-oriented editing

Because edit gestures may be targeted either at the document or the presentation, we distinguish two kinds of editing: *document-oriented* versus *presentation-oriented* editing.

On the one hand, we have *document-oriented editing*, which consists of edit operations (including navigation and selection) that are targeted at the structure of the document rather than at its presentation. Examples are swapping two chapters in a word processor, selecting an entire chapter, or navigating to a next section.

On the other hand, *presentation-oriented* editing consists of edit operations on the presentation, which do not necessarily make sense at the document level. If a presentation is textual, presentation-oriented editing amounts to freely editing the text. As an example, take the mathematical expression $(1 + 2) \times (3 + 4)$. Deleting the middle part $(1 + \overline{2}) \times \overline{(3 + 4)}$ yields $(1 + 3 + 4)$ and is a presentation-oriented edit operation that does not directly correspond to a logical operation on the document level. Another example is navigating downwards in a formatted paragraph of a word processor, since the concept of lines in a paragraph only exists at the presentation level.

Section 3.5 provides a more thorough discussion of both document- and presentation-oriented editing. Furthermore, the section discusses editing at several other levels, which are introduced at the beginning of Chapter 3.

Different kinds of editors

The term *structure editor* is used to make explicit that an editor has document-oriented editing functionality (also including navigation). We do not make a sharp distinction

between plain-text editing and structure editing. Instead, we regard all editing as structure editing, but with a varying level of structure. A text editor can be seen as a structure editor with a very simple structure model: a string or a list of strings. Document-oriented and presentation-oriented editing coincide for a text editor.

An editor is a *generic editor* if it is not specifically built for a single document type but can be used to edit a whole class of document types. A generic editor may be *instantiated* to yield an editor for a specific document type. Genericity can be achieved with a single generic editor that edits documents of arbitrary types, but also with an editor generator. An *editor generator* is an environment that generates an editor application based on descriptions of the document type and its presentation. Although a generator is not as versatile as a single generic editor, we view both as generic editors.

For brevity, we will often adopt the common practice that the term structure editor implies genericity as well. Still, structure editors that are not generic are quite common. A few examples are: equation editors, bookmark editors in web browsers, and file browsers. On the other hand, a generic editor is always a structure editor since it knows about the type of the document.

In the context of generic editing, the term *user* is ambiguous. A user can either be an editor designer, who instantiates the generic editor for a specific domain, or a user who is editing a document. Unless explicitly stated otherwise, we use the term for the document-editing user.

Because it is difficult to give a precise definition of a generic structure editor and because such a definition might be restrictive, we will discuss a number of typical use cases to clarify what we mean by a generic structure editor. Section 2.1 presents these use cases.

1.1.4 Advantages of generic structure editors

An editor that knows about the structure of the edited document can offer interesting functionality. We list several potential advantages of generic structure editors. The first two advantages stem from the genericity of the editor, whereas the rest are mainly about the structural (document-oriented) abilities.

Uniform user interface/edit model. Rather than a separate editor application for each type of document, a single generic editor can be used for a range of document types. Thus, instead of having to cope with several slightly different interfaces, a user only needs to deal with a single uniform interface and edit model.

Integration of documents. Besides offering editors for different types of documents, a structure editor also facilitates the integration of different types of documents into a single editor instantiation. Thus, it is relatively easy to build an editor for a specific document type, with advanced functionality for the different kinds of edit. Examples are a word-processing editor with spreadsheet functionality, or an editor for slide shows that has syntax coloring and type checking for program code appearing in the slides.

Different Views on the Document. A structure editor may provide a user with several editable views on the document. The views can show the document in a different order, or with a varying amount of detail.

Graphical Views. A view may contain color and fonts in order to clarify document structure, but also use layout alignment, and graphical elements such as lines and boxes.

Derived Information in the Presentation. The editor can analyze the document during editing and display information computed from the document structure. Examples are the results of static analysis and type checking in source editors, but also chapter numbers or an automatically generated table of contents.

Structural Edit Operations. Certain edit operations, such as demoting a section with subsections to a subsection with subsections in a scientific article, are straightforward to specify at the structural level, but awkward at the presentation level.

Structural Navigation. Navigating over the document structure instead of its presentation can be very useful. In a source editor, when the focus is on an identifier, a user may easily navigate to its definition in the source. Furthermore, an outline view of the document can be shown in which a user can click to navigate to the corresponding position in the document.

Integration with Other Tools. A structure editor allows fine control over integration with other tools, such as spell checkers, program-transformation systems, and theorem provers. Furthermore, the editor may show the results coming from these tools at the appropriate position in the presentation, rather than as a list of messages with line numbers.

For document types with a textual presentation, such as program sources or XML documents, some of the advantages can be simulated with a text editor. Lexical analysis can be used on the edited text, and basic support for syntax coloring, auto-completion, and navigation can be provided. However, although simple and efficient, these solutions are very basic and prone to errors, because, in general, much of the structure of a document cannot be recognized at a purely lexical level.

1.1.5 Classes of structure editors

Three classes of structure editors are distinguished in the literature: *syntax-directed*, *syntax-recognizing*, and *hybrid* editors. Syntax-directed editors mainly support edit operations targeted at the document structure, whereas syntax-recognizing editors support edit operations on the presentation of the document. A hybrid editor combines syntax-directed with syntax-recognizing features, but the term is not used consistently.

Syntax-directed editors

The first structure editors that were developed are the *syntax-directed*, or *pure*, structure editors [6, 54, 77].

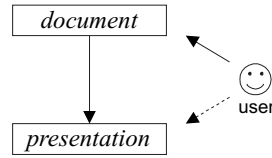


Figure 1.2: A syntax-directed editor.

Early syntax-directed editors show a textual presentation of the document (usually a program source) but exclusively offer edit operations targeted at the internal document structure, and not at the textual presentation. The original idea behind this was that if structural edit operations are available, a user would not need the textual edit operations anymore. Further, presentation-oriented edit operations would interfere with the user's structural model of the document and introduce errors. Hence they were prohibited altogether. Most editors for XML (see also Section 2.3.4), as well as editors for preferences panes, can be regarded as syntax-directed editors.

Figure 1.2 shows a schematic representation of a syntax-directed editor. The editor works by computing a presentation of the internal document structure, which is shown to the user together with a current focus of attention. The user provides an intended edit operation (edit gesture) on the document structure, from which a document update is computed. After the document is updated, a new presentation is computed, which is shown to the user.

If the editor supports clicking in the presentation to set the focus, the editor also needs to keep track of the origin in the document for each position in the presentation.

In the figure, the line between the user and the presentation is dotted because syntax-directed editors do not support edit operations on the presentation very well. Because the presentation is derived from the document, the editor needs to interpret the intended edit operation on the presentation as an edit operation on the document, which is difficult if the edit operation is not a logical operation on the document level.

A major problem with syntax-directed editors is the restrictiveness of the edit model (e.g. [62, 88]). New structures are easy to create, but not as easy to modify. For example, if a user wishes to change a while statement to an if statement, simply typing over the keyword is typically not supported.

Many later syntax-directed editors offer a form of presentation-oriented editing by providing a freely editable textual presentation of (part of) the document, and applying a parser to the edited text. Some publications [58, 86] refer to such editors as hybrid, but, as we will explain below, we still regard these editors as syntax-directed editors.

Unless the two forms of editing are completely integrated, the textual presentation forces a user to work in a different mode of editing, which is referred to as *mode switching*. Mode switching does not solve the problem of restrictiveness adequately. Often, a separate

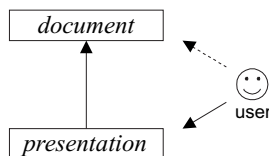


Figure 1.3: A syntax-recognizing editor.

window showing a text-only presentation is opened and before the mode can be switched back, the edited text has to be valid. Furthermore, separate modes require a user to be constantly aware of the current mode of the editor. The resulting increased cognitive burden has been shown to be a source of errors [79].

Syntax-recognizing editors

At the other end of the spectrum are the *syntax-recognizing* structure editors [7, 18]. A syntax-recognizing editor keeps track of the textual presentation of the document. The user can freely edit the text, and the editor tries to recognize the document structure by means of a parser. Once the text has been (partially) recognized, structural information (e.g. syntax-coloring or type information), navigation, and, in some editors, edit operations are available.

Figure 1.3 schematically shows a syntax-recognizing editor. The user's edit operations are targeted at the presentation, which can be edited freely. The document is derived by parsing (interpreting) the presentation; hence the reversed direction of the arrow, compared to Figure 1.2.

For each element in the document structure, the editor needs to keep track of what parts of the presentation it corresponds to, in order to show structural information in the presentation, as well as support structural navigation. When a document structure has been recognized, the presentation may show additional information using font and color changes, context-sensitive menus, tooltips, etc.

Similar to the syntax-directed editor, the picture of the syntax-recognizing editor in Figure 1.3 also contains a dotted arrow. In this case, because the document is derived from the presentation, structural edit operations on the document are difficult to support. A document-oriented edit operation has to be mapped onto an update on the presentation, in such a way that parsing the updated presentation returns the intended updated document. Presentation information that is not stored in the document tree, such as whitespace and comments, has to be related to the document tree in some way, in order to be put in the right place after a structural edit operation.

The main problem with syntax-recognizing editors lies in their limited applicability. Because the presentation needs to contain enough information to derive the document, interesting presentations that only show part of the document are hard to support. Furthermore,

graphical presentations, as well as presentations containing computed values and structures, do not fit the model, as these are difficult to parse. As a result, syntax-recognizing editors are mainly limited to text-oriented applications, such as program-source editors.

Hybrid editors

A *hybrid* editor supports structural as well as presentation-oriented edit operations. Figure 1.4 shows a hybrid editor. Because both levels can be edited, there are no dotted arrows in the figure. However, in order to offer this edit functionality, the editor must realize both the presentation and interpretation mappings. Hence the double arrow between the document and the presentation.

In some publications (e.g. [58, 86]), the term hybrid is used to refer to syntax-directed structure editors that have a limited form of syntax-recognizing functionality. As a consequence, most syntax-directed editors would qualify as hybrid editors, because most editors support some form of text parsing.

In contrast, other publications (e.g. [7, 49]) advocate that the term hybrid should be reserved for editors that support full textual editing of the document, as well as limited syntax-directed functionality, even if structural modifications on the document are not supported. According to this view, almost all syntax-recognizing editors would classify as hybrid editors, since most of these editors support a form of structural navigation.

Because of the confusion, and because most editors tend to be either primarily syntax-directed or syntax-recognizing, we will often use those terms, instead of the term hybrid.

1.2 Proxima

The subject of this thesis is the design of the presentation-oriented structure editor Proxima. Proxima is suitable for a wide range of applications, including word-processors and source editors, but also mathematical-equation editors and spreadsheets. An important aspect of Proxima is that the editor fully supports presentation-oriented as well as document-oriented editing. Thus, the editor classifies as a hybrid structure editor.

The implementation of the bidirectional mappings between the document and the presentation is facilitated by a layered architecture. The computation of the presentation is broken up in several stages, and each intermediate value in this computation corresponds to a *data level* (or just *level*). The interpretation process has the same intermediate values. Between two levels, there is a *layer*, which is a component that takes care of mapping values at one level onto values at another level, and thus implements a single stage of the presentation and interpretation processes.

The first step of the presentation process is that the *document* is enriched with computed values and structures, by the evaluator. The resulting *enriched document* is mapped onto a logical *presentation* in which positions and sizes are specified relatively. The layout layer adds whitespace that is not stored in the document to the presentation, yielding the *layout*

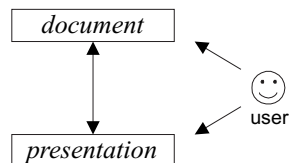


Figure 1.4: A hybrid editor.

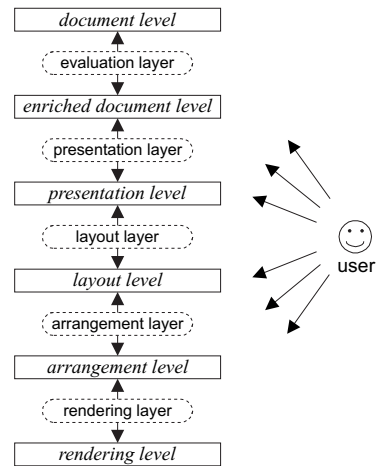


Figure 1.5: Proxima.

level. The layout level is mapped onto an *arrangement* level, which contains absolute positions and sizes. And finally, the renderer maps the arrangement onto a *rendering*, which is made visible to the user. A more detailed discussion of the presentation and interpretation processes is provided in Chapter 3.

Figure 1.5 shows the levels and layers of Proxima. Because the name *presentation* is reserved for one of the intermediate levels, the lowest level is referred to, more appropriately, as the *rendering* level. The figure shows multiple edit arrows coming from the user, because edit operations may be targeted at intermediate levels as well.

The layered architecture makes it possible to combine presentation-oriented editing with a powerful document-presentation mechanism that includes support for derived values and structures. A platform-independent Haskell prototype of Proxima has been implemented, and experiments with instantiated editors have yielded promising results.

1.3 Terminology

We give a brief summary of the terms that were introduced in the previous sections.

Editor: Application for creating and modifying documents. In this thesis, the term also used to refer to structure editors and generic editors (or generic structure editors).

Document: Internal data structure that represents the information that is edited.

Presentation: Term for a visible representation of the document (also called a *View*), as well as for one of the intermediate levels of the presentation process.

Presentation mapping/relation: The relation between the document and its presentation.

Presentation-oriented editing: Edit operation targeted at the presentation:
e.g. deleting “ + ” from “1 + 2”, yielding “12”.

Document-oriented editing/Structure editing: Edit operation targeted at the document:
e.g. swapping two sections in an article.

Presentation sheet: Parameter to the presentation/interpretation process.

Presentation process: Process of computing the presentation of a document.

Interpretation process: Process of computing a document from a presentation.

Level: Intermediate value of the presentation/interpretation process, including the document and the presentation.

Layer: Component that realizes the presentation and interpretation mappings between two levels.

Structure editor: An editor that has knowledge of the structure of the edited document.
Usually assumed to be a *generic editor* as well.

Generic editor: A structure editor suitable for editing documents of different types.

Syntax-directed editor: An editor that primarily supports document-oriented editing.

Syntax-recognizing editor: An editor that primarily supports presentation-oriented editing.

Valid document: A well-typed document, mainly used in the context of XML.

Valid presentation: A presentation that is the result of presenting some document.

Focus: Shared name for cursor and selection.

1.4 Outline of the thesis

The remainder of this thesis has the following structure:

Chapter 2 explores applications of generic structure editing by providing five use cases of real-world editors. With these use cases in mind, we formulate a number of functional requirements that in our view are important for a flexible non-restrictive structure editor. We evaluate a number of existing editors according to the requirements, and conclude with an overview of how the Proxima editor is designed to meet the requirements and be able handle all use cases.

The layered architecture of Proxima is introduced in Chapter 3. The chapter discusses the various data levels, as well as the layers that maintain the mappings between the

levels. The discussion is illustrated with examples of the presentation and interpretation processes.

In chapters 4 and 5, we develop a specification of the Proxima editor. Chapter 4 serves as an introduction to the specification and introduces our model of the edit process, as well as the concepts of extra state and duplicates in the presentation. In Chapter 5, we start by specifying a simple editor, to which extra state and multiple layers are added in subsequent sections. The chapter ends with an informal discussion on how to handle presentations that contain duplicates.

In Chapter 6, we discuss the XPREZ presentation formalism of the Proxima. XPREZ is a declarative presentation language, suited for specifying a wide range of presentations. We state a number of requirements for a presentation language for structured documents, and provide an informal overview of XPREZ, using a series of examples.

A prototype that offers much of the functionality discussed in this thesis has been implemented in the functional language Haskell. Chapter 7 discusses the prototype as well as a number of editors that have been instantiated. The chapter also explains which components need to be provided to instantiate an editor.

Finally, Chapter 8 presents the conclusions and gives an overview of future research.

