CHAPTER 1

Editing Structured Documents

*** Introduction is added when other chapters are finished.
*** Pictures are still drafts, because they may depend on experiences with the prototype.

1.1 Editing

While the term *editor* is usually only associated with text editors such as Emacs [**?**] or vi [**?**], we will use it in a much broader sense. We regard as an editor any application that keeps track of an internal data structure that is shown to the user in the form of a bitmap, and that can be changed by that user. The internal data structure is referred to as the *document*, and the bitmap is the *presentation*.

Figure **??** schematically shows the edit process. The editor shows a presentation of the internal document together with the current focus of attention to the user. The focus of attention, or *focus*, is a shared name for the selection as well the cursor (which is an empty selection). The user provides the editor with *edit gestures*, such as key presses or mouse movements, which are mapped onto updates on the document. The document is then re-presented and shown to the user. This process is repeated until the user quits the editor. Chapter **??** contains a more formal definition of the edit process.

Naturally, word processors, image editors, and XML editors are regarded as editors in this view, but there are also some less obvious examples. Take for example the preferences pane that is part of most window based applications. The set of check boxes, selection lists, and text fields can be seen as a presentation of the preferences of the application.

With such a broad view of editors, it is possible to regard every application, and even an entire operating system as an editor. In essence, all a computer user does is give edit

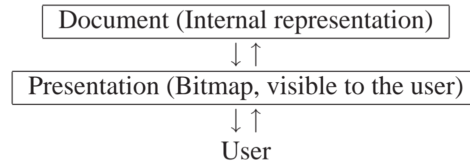


Figure 1.1: Editing (Draft)

gestures with the mouse and the keyboard in response to the presentation on the computer monitor. In reaction to the edit gestures, the internal state of the computer changes, giving rise to a new presentation.

There is no fundamental problem with this view, but we do not adopt it because a definition that is too broad does not help in finding appropriate abstractions for a generic structure editor. Therefore, we do not explicitly consider all applications to be editors, but adopt the view that many applications contain editors.

A *structure editor* is an editor that has knowledge of the structure of the document that is edited. More precisely, it offers edit operations that are not targeted at the presentation of the document, but rather at the document itself. Some structure editors exclusively offer edit operations targeted at the document structure (*document editing* or *structure editing*), whereas others also allow edit operations targeted at the presentation (*presentation editing*). Another possibility is that an editor offers structural navigation together with information on the structure but without any structure edit operations that modify the document.

We do not make a sharp distinction between text editing and structure editing. Rather, 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.

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. Instead of one single editor that works on arbitrary document types, genericity can also be achieved with an editor generator. The generator is an environment that generates an editor application based on a description of a specific document type. Although a generator is not as versatile as a single generic editor, we still do consider editor generators to be generic.

The term structure editor is often associated with genericity, but non-generic structure editors are quite common. A few familiar examples are: equation editors, bookmark editors in web browsers, and outline editors.

In the context of generic editing, the term *user* can refer to either an editor designer, who tailors the generic editor for a specific domain, or a user that is editing a document. Unless explicitly stated otherwise, we use the term user for the document editing user.

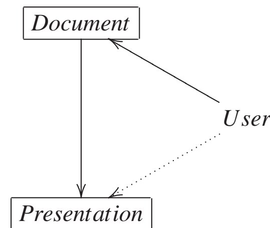


Figure 1.2: A syntax-directed editor

Because it is difficult to give a precise definition of a generic structure editor, as well as the possibly restrictive consequences of such a definition, we will rather use a number of typical use cases to clarify the concept. Section **??** presents the use cases.

1.1.1 Classes of Structure Editors

Three classes of structure editors are distinguished in 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 in literature.

Syntax-Directed Editors The first structure editors that were developed are the *syntax-directed editors* [**?**, **?**, **?**], also known as pure structure editors. Early syntax-directed editors show a plain text presentation of the document, usually a program source, but exclusively offer edit operations targeted at the internal document structure, and not at the presentation. The idea behind this was that if structural edit operations were available, a user would not need the textual edit operations anymore. Worse still, presentation-oriented edit operations would interfere with the user's structural model of the document and introduce errors, so they were prohibited altogether. Most XML editors and editors for preferences panes can be regarded as syntax directed editors.

Figure **??** shows a schematic representation of a syntax-directed editor. The editor works by computing some 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.

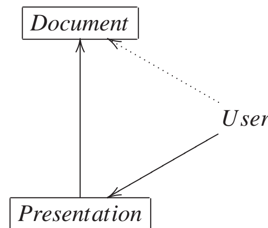


Figure 1.3: A syntax-recognizing editor

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, an edit operation on the presentation first needs to be mapped onto an edit operation on the document. Some syntax-directed editors offer a freely editable textual presentation of (part of) the document, and apply a parser to the edited text. Because this forces the user to work in a different mode of editing, this approach to editing is called *mode switching*.

The major problem with syntax-directed editors is the restrictiveness of the edit model ***. If a user wishes to change a while statement to an if statement, simply typing over the keyword is often not supported. The mode switching approach is not an ideal solution either. Often, a separate window showing a text-only presentation is opened and before the mode can be switched back again, 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 [**?**].

∗ ∗ ∗ add refs

Syntax-Recognizing Editors

At the other end of the spectrum, there are the *syntax-recognizing* structure editors [**?**,**?**,**?**]. 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, navigation, and, for some editors, edit operations are available.

Figure **??** 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 the presentation, hence the reversed direction of the arrow, compared to Figure **??**.

In order to show structural information in the presentation, as well as perform structural navigation, the editor needs to keep track of the origin in the presentation for each element in the document structure. When a document structure is recognized, extra information can be shown in the presentation, for example font and color changes, context sensitive menus, etc.

Similar to the syntax-directed editor, the picture of the syntax-recognizing editor also has a dotted arrow. In this case, because the document is derived from the presentation, edit operations on the document are difficult to support. A document edit operation has to be mapped onto an update on the presentation, such 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, so that it can 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 editors.

Hybrid Editors

The class of hybrid editors is used for editors that support structural edit operations as well as presentation (often just textual) edit operations.

In some publications (e.g. [**?**, **?**]), the term hybrid is used to refer to syntax-directed structure editors that support some form of presentation editing. As a consequence, most syntax-directed editors would qualify as hybrid editors, because most editors support some form of text parsing.

Others (e.g. [**?**, ?, **?**]), however, advocate that hybrid should be reserved for editors that support full textual editing of the document, as well as structural edit functionality, even if structural modifications on the document are not supported. In this view, almost all syntax-recognizing editors would classify as hybrid editors.

Because of the confusion, and because of the strictness of the syntax-directed versus syntax-recognizing classification, we rather regard the levels on which edit operations are supported together with the integration between edit operations on different levels. Syntax-directed editors mainly support document level edit operations, whereas syntax-recognizing editors mainly support presentation level edit operations.

1.1.2 Advantages of Structure Editors

An editor that knows about the structure of the edited document can offer a lot of interesting functionality. Advantages include:

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 with 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 rect-

angles. Graphical views may also be used for WYSIWYG document editing.

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

Structural Edit Operations. Some operations, such as changing a section with subsections to a subsection with subsubsections in a scientific article, are straightforward to specify at structural level, but awkward on presentation level.

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

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 much of the structure of a document cannot be recognized at a purely lexical level.

Despite the advantages, generic structure editors are not widespread at all. Several structure editors are used in small communities, but most development projects have been terminated, and in the last decade, very few publications on the subject have appeared. The rise of the XML standard has spawned a large number of generic editors, but when regarded as structure editors, XML editors do not show much variation and do not offer many of the possibilies that a structure editor could offer. Hence their applicability is limited, and using an XML editor for example to edit a java program source for example is not possible with the current generation of editors.

∗ ∗ ∗ add refs!!

As mentioned***, users regard structure editors as being either overly restrictive, or not powerful enough. In the remainder of this chapter, we first give a set of possible applications of a structure editing, and then explore a number of functional requirements that in our view are important for creating a flexible non-restrictive structure editor.

1.2 Use Cases

In this section we present five example applications of a generic structure editor. Some of these cases are well-known applications of structure editors, but a few more exotic applications have been included as well.

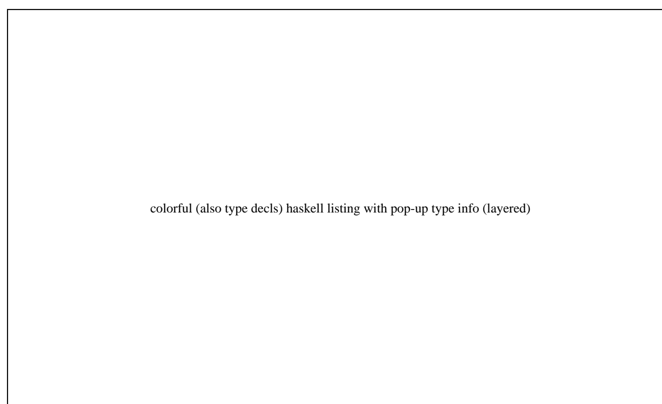
The use cases serve a two-fold purpose. Firstly, we will use them as standard examples to explain and define edit behaviour in the next chapters. Secondly, the use cases illustrate

the rather vague definition of the term editor from the previous section, and serve as a basis for the formulation of a set of functional requirements.

Unlike any current structure editor, the Proxima editor will be able to handle all five use cases. It is important to note that although the use cases are discussed as separate applications, aspects of them be combined in a single editor instance.

1.2.1 An Editor for Haskell

As an example of a program editor, we take an editor for the functional language Haskell. The editor supports an extended form of syntax highlighting, in-place display of syntactic and semantic errors, and a range of language based edit operations. Unlike most text editors, the editor supports highlighting at a semantic rather than syntactic level, making it possible to use different display styles for language constructs that are hard to recognize purely syntactically. The type declarations in the next screenshot are an example of such a construct. Although syntactically identical, identifiers in type expressions are colored differently from identifiers in ordinary expressions.



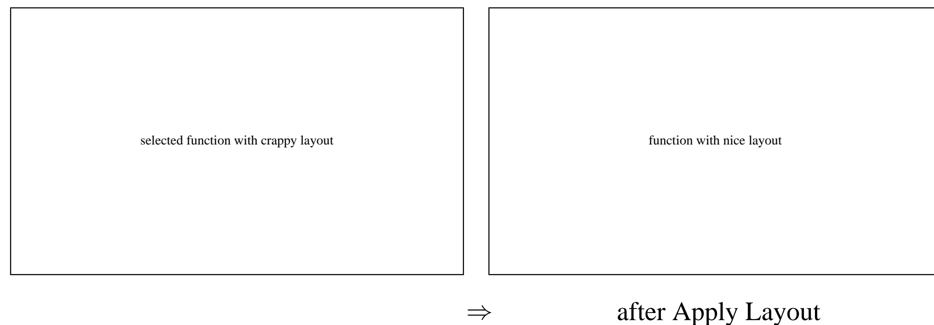
The Haskell source editor

Haskell is a particularly interesting language for a program editor because it has a rich type system and information about types is very useful during programming. Haskell programmers often experience that once type errors have been removed, a function is correct. Therefore, an environment that supports in-place display of type errors, as well as easy access to type information of variables in scope, will help rapid program development.

The screenshot shows the type of the expression in focus, together with a list of variables in scope. Selection in the menu causes a jump to the definition of the identifier. Because many identifiers may be in scope, the menu is layered according. If many identifiers are in scope, a separate window may be used for inspecting type information.

Automatic Layout

Some structure editors use an automatic layout scheme while editing program sources. The user then does not need to worry about layout issues, such as the alignment of parameters in functions with multiple clauses. However, for a Haskell editor, this situation is not optimal because Haskell programs mainly consist of expressions, which are hard to layout automatically. Therefore, rather than enforcing automatic layout, the editor should offer support to layout selected parts of the program on demand. The specification of the layout of the program is part of the presentation sheet. Of course, if desired, it is also possible to turn on continuous automatic layout.

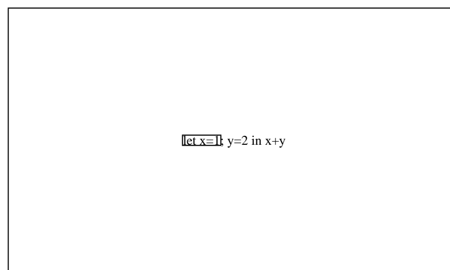


⇒ after Apply Layout

The screen-shot shows the automatic layout operation applied to the selected function. The new function on the right side is still freely editable, including its layout.

Structural Edit Operations

Because a program construct is represented by a contiguous area in the presentation, moving a program construct can usually be done in a straightforward way by moving its presentation. However, this is not always the case. Consider for example the following let expression:

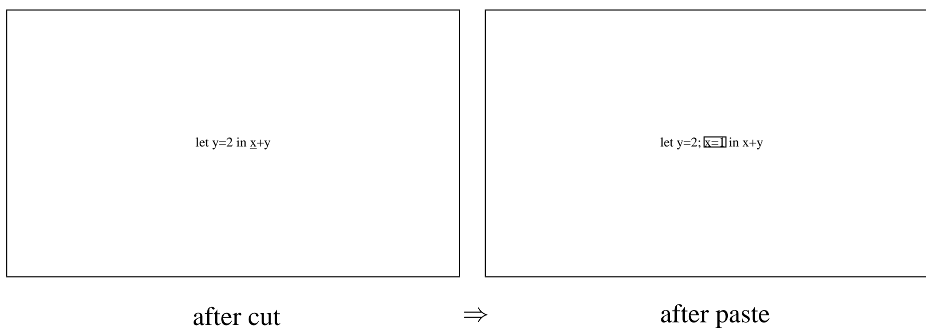


after select

The let expression consists of a list of declarations that are separated by semicolons and whitespace. The semicolon here is a separator and belongs to the presentation of the entire let expression rather than to the presentation of a single declaration. As a result, the semicolons may cause problems when declarations are moved.

As an example, consider moving the first declaration (`x = 1`) to the end of the let expression. When the declaration is cut, the semicolon behind it must be deleted, and when the declaration is pasted at the end, a semicolon together with appropriate whitespace must be added. Similar problems occur with all list structures that are presented using separators, such as Haskell lists `[1, 2, 3]`, tuples `(1,2,3)`, or monadic `do` expressions `do {a <- getChar ; putStr [a]}`.

If the structure of the edited list is taken into account, the problem can be solved elegantly. When the first declaration is selected, the editor recognizes it as an element of the let expression's declaration list, and when it is cut, the semicolon next to it disappears:

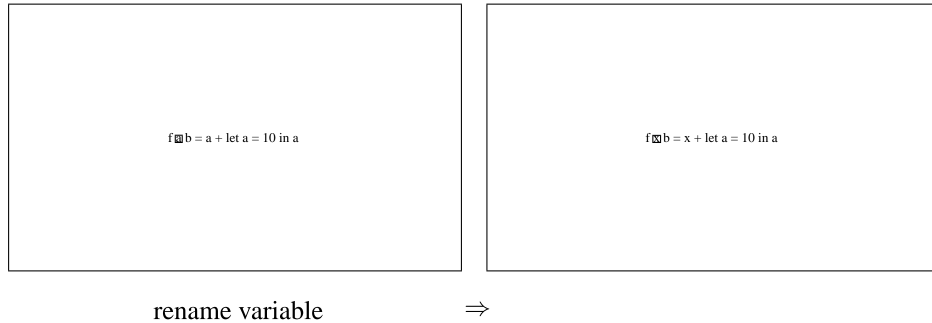


after cut ⇒ after paste

When the declaration is pasted, a semicolon is automatically placed in front of it. The whitespace from the semicolon is copied from the whitespace of the other semicolons in the presentation. When the presentation of the list contains explicitly added whitespace, causing an irregular presentation (eg. `[1,  2,     3, 4]`), the layout of the result after paste may not be what is expected. However, since list structures are usually layed out consistently, this need not be a problem.

Rename within Scope

A second example of an edit operation that takes the document structure into account, is a rename operation on an identifier. In a regular text editor, occurrences of the identifier name need to be changed using search and replace. However, because the identifier name may appear in a string, or an identifier of the same name may be declared in inner structures, automatic search and replace does not always work.

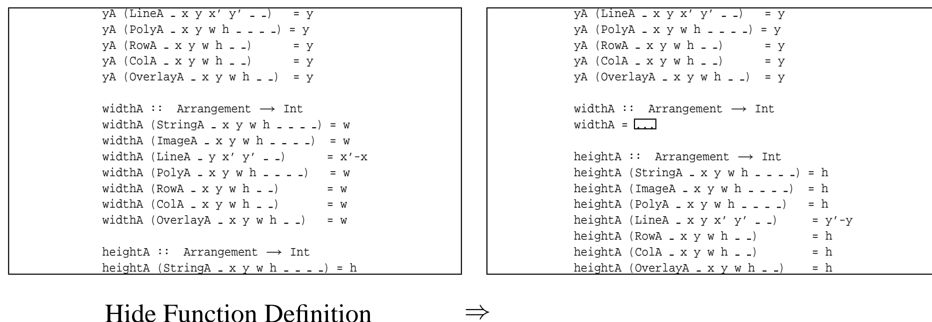


rename variable ⇒

The rename operation takes into account the scoping rules of Haskell, and only changes the appearances that lie in scope of the updated identifier. If the new name is captured by an inner declaration, or if it shadows an identifier that is already declared, a warning is issued.

Hide Function Definitions

Function definitions in the source presentation may be collapsed, leaving only one left-hand side and the (possibly inferred) type declaration:



Hide Function Definition ⇒

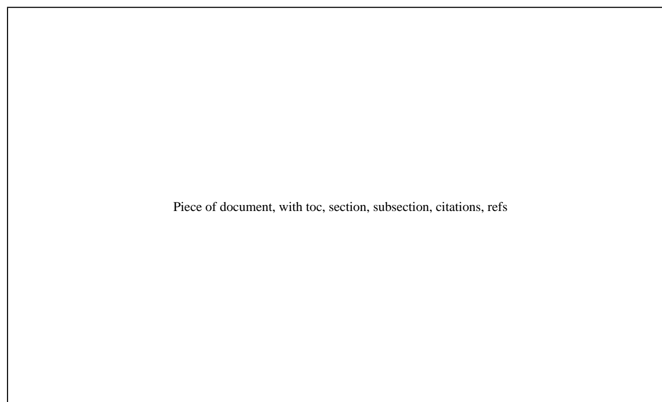
The two functions `widthA` and `heightA` have a large number of uninteresting clauses. After applying the *hide function body* edit operation to the function `widthA`, only one clause remains with a collapsed righthand side (...). The function is expanded again by clicking on the dots.

Requirements

The program editor generates a number of requirements, an important one being possibility of freely editing the textual program source, including the layout. At the same time, it must be possible to let the layout be set automatically as well. Furthermore, a formalism for specifying computations over the document is required for performing static analysis and type checking.

1.2.2 A Word Processor

This section describes a WYSIWYG document editor with a user interface similar to word processing applications such as Microsoft Word, but with a document model similar to the XML DocBook standard [?] and with an output quality similar to LaTeX [?].



Word processor

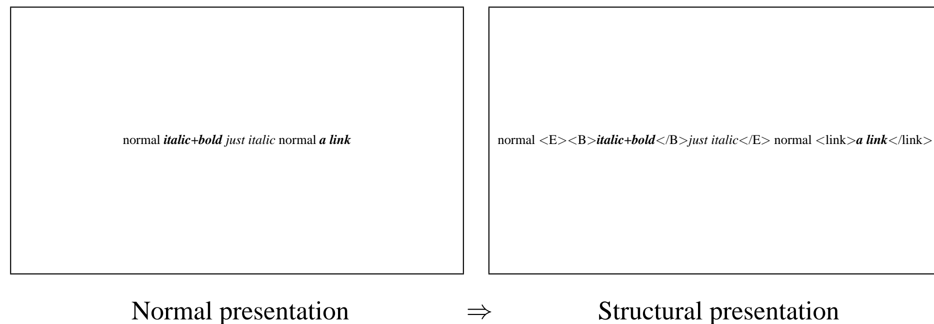
The document model consists of chapters, sections and subsections. The editor supports free editing in the WYSIWYG document presentation with support for optimal*** line breaking, a derived table of contents and an automatic bibliography. Cross-references such as references to tables and figures or citations, can be clicked to bring the referred part of the document into focus.

*** not yet

Structural View on the Document

Although Microsoft Word is one of the most popular word processing tools in the world, an often heard complaint concerns its confusing document model. Sometimes edit operations are not allowed because of underlying document structure, but it is not obvious why this is the case. Furthermore, the reason why a document fragment looks the way it does is not always clear. The user may have set specific style attributes for a particular fragment, or the style may originate from the document's presentation rules. Microsoft Word lacks a structural view such as Wordperfect's "underwater" screen, which shows the exact structure of the document tree.

A structure editor can support presentations with various degrees of structure.



Normal presentation ⇒ Structural presentation

The two screenshots show two presentations of the same document fragment. The left presentation is the regular WYSIWYG presentation, whereas the right one is a more structural presentation that shows the markup tags. The example document also contains a fictitious `<link>` element that is presented in a bold and emphasized style***. In the left presentation, it cannot be distinguished from ordinary bold and emphasized text, but in the right presentation it can.

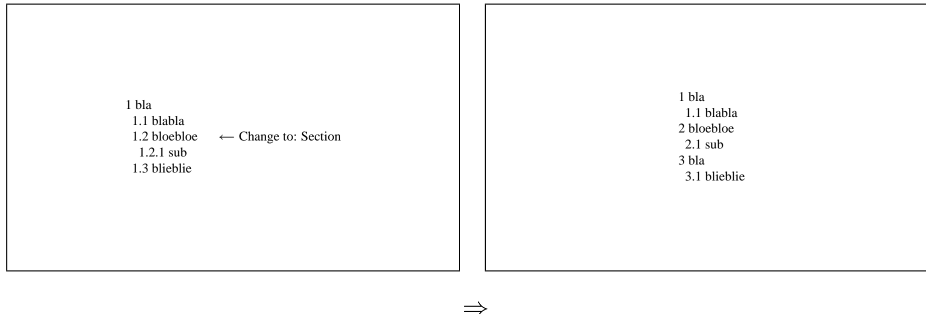
*** maybe find a non-fictional element, or explain more?

The structural view can also be helpful for positioning the focus. For example, in the left presentation, the start of the first emphasized text just before the bold part overlaps with the start of the bold part, therefore adding text that is emphasized but not bold is rather tricky in the first presentation. In the second presentation, on the other hand, the positions do not overlap and emphasized as well as emphasized and bold text can be added without a problem. In order for the structural views to be helpful, the editor supports easy switching between views while preserving the current focus.

Structural Edit Operations

Edit operations that rearrange the document structure, such as lifting a subsection to a section are awkward to perform on a textually represented document, such as a LaTeX source. The tags or LaTeX commands specifiying the subsection and its descendants need to be changed. This is a rather specific search/replace operation on only part of the document source, which is a hassle to automate.

A structure editor may be of some help here, because the structural similarities between sections and subsections are known to the editor and can be used to define edit operations for restructuring the document.



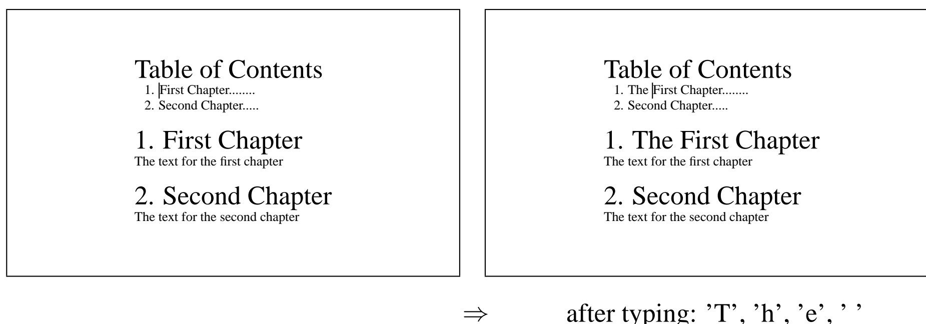
The screenshot shows the effect of the document edit operation *Change to Section*. The containing section is split in two sections with the same title, with the lifted subsection in between.

An operation that changes the level of a section or subsection is rather complex, because it involves splitting and changing elements. Moreover, there are special cases to consider. For example, when a section that contains a subsubsection is changed to a subsection, a warning needs to be issued since no level below subsubsection exists. Therefore, such an operation needs to be specified explicitly by the editor designer or user. Other document operations, however, such as splitting and joining elements of a list, may be derived automatically.

Editing a Title in the Table of Contents

The word processor has support for the specification of a generated table of contents***. From the entries in the table, the user can jump to the corresponding position in the document presentation. The presentation of the table of contents itself can be customized to match the style of the rest of the presentation. When the document is edited, the table of contents is updated accordingly. Moreover, a title in the table of contents can be edited, causing an update to the title in the document.

*** page numbers?

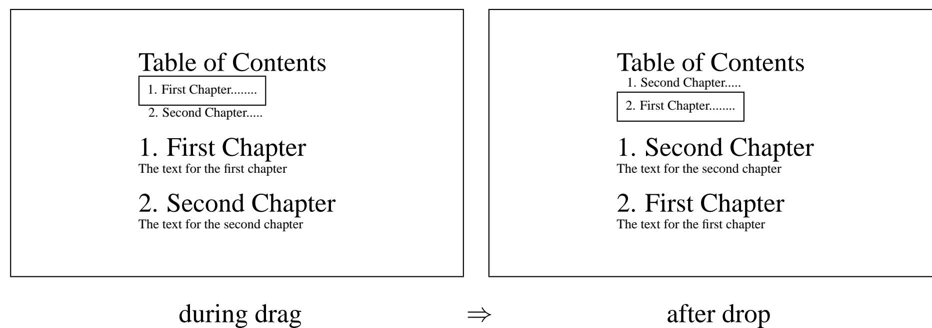


The screenshot shows a two chapter document with a table of contents. The first entry

in the table of contents is edited by adding the text "The " to the start of the title, which causes an update on the title of the corresponding chapter.

Moving a Section in the Table of Contents

Besides textual edit operations, it is also possible to perform structural edit operations on derived structures. The screenshot shows a move operation on a section title in the table of contents, which has as its result that the corresponding chapter is moved in the document.



during drag ⇒ after drop

The chapter entry for chapter 1 is selected and dragged to its new location, just below the entry for chapter 2. The result is an edit operation on the document structure that puts the first chapter after the second chapter. The chapter numbers switch because they are generated automatically. Whenever an edit operation on a derived structure is performed, the user may be signalled that the operation affects more than just the visible selection.

Although structure changing operations on derived structures may not always make sense, it is important that they can be specified for the cases in which they do.

Requirements

Compared to the program editor, the word processor requires a more powerful presentation formalism. Besides text in different fonts, colors, and sizes, the presentation also contains images and basic graphical elements. Furthermore, optimal line and page breaking support is needed for formatting paragraphs and pages.

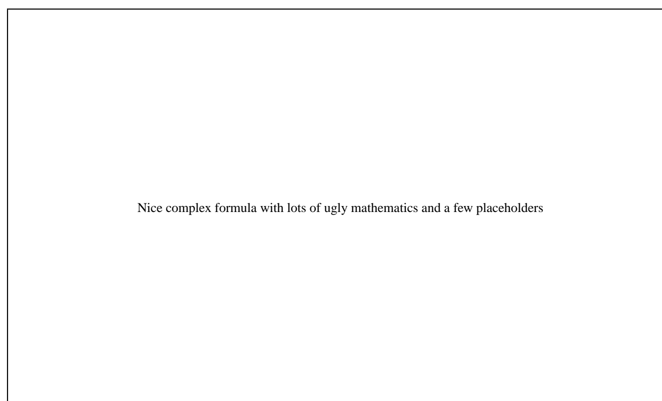
Finally, in order to handle edit operations on the table of contents, the editor must support editing not only on presentation and document level, but also on the level of derived structures. In order to do so, a mapping must be maintained between a derived structure and its origin in the document.

1.2.3 Equation Editor/MathML

Because mathematical formulae have a high degree of structure, a mathematical equation editor is a good candidate for structure editing. In an actual instance of the generic editor,

the equation editor can be integrated with the word processor to support equation editing in the document that contains the equations. However, we separate the two examples because of the different requirements they provide.

The screenshot shows a WYSIWYG equation editor with support for mathematical constructs such as fractions, roots, and integrals. The document model for the equation editor can be the mathematical markup language MathML [?]



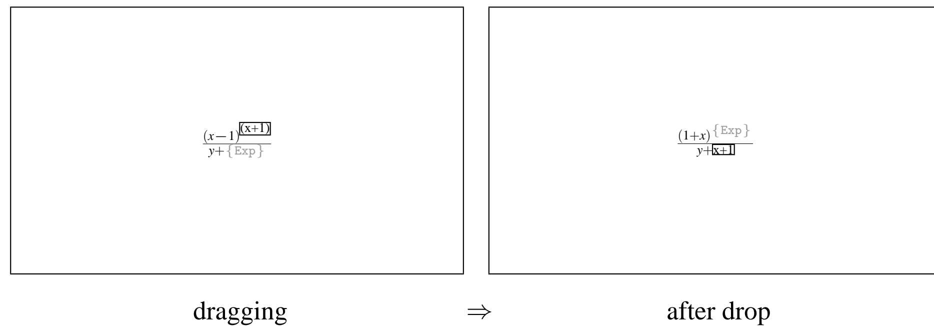
The equation editor

Due to their structured nature, mathematical formulae are suitable for document edit operations, using menus and buttons for structure entry. Free presentation editing, on the other hand, is not as clearly defined on a formula as it is on a program source. For example, shrinking the 2 in the number 42 and moving it upwards a bit, could theoretically lead to recognition of the square 4^2. However, this requires a complicated visual parsing scheme, the exact behavior of which is not clear. Therefore, the editor only allows free editing in the textual parts of a formula that can be parsed unambiguously.

Although the restrictions that are put on the edit model are common even in the current generation of non-generic equation editors, we believe that a more sophisticated and flexible edit model is possible. The Proxima architecture does not prohibit such an edit model, but more research on parsing two-dimensional structures is required before it can be supported.

Drag and Drop

Direct manipulation of parts of the formula is supported on a structural level. A proper subtree of the formula can be dragged to a different location.



dragging ⇒ after drop

The subformula is dragged to its new location below the fraction bar, leaving a placeholder ({Exp}) at its origin. Note that the parentheses disappear because the + and − operators are of equal precedence.

Only proper subtrees in the document may be selected in the equation editor. This means that in the formula 2^{3^4}, the 2^3 part may not be selected because it is not a proper subtree (the power operator is right associative). ***

∗ ∗ ∗ mention associative operators?

In practice, we do not expect this restriction to be a major problem. A fragment of the presentation that does not correspond to a proper subtree does not actually represent a meaningful computation in the formula. Hence, the chance that the fragment is reused elsewhere or needs to be moved is small. One situation in which this will occur is when a user needs to build an expression that by chance has exactly the same presentation as some already present non-subtree selection, which is not very likely.

Textual Structure Entry

For quick and easy structure entry, the editor supports textual entry of mathematical structures without having to switch to a different mode.

keyboard: `19 + 1 * 5 + 2 * 8` ⇒

The entered text causes the insertion of the expression $19+1*5+2*8$, as shown on the right. It should be noted, however, that textual entry does not always lead to the desired

result in a two dimensional presentation. For example when ”2/4” is entered, an intuitive result is the addition of a fraction with the focus ending up below the fraction line to the right of the 4: $\frac{2}{4}$. But now, the expected result of entering ”+6” is $\frac{2}{4+6}$, whereas the correct meaning of ”2/4+6” is $\frac{2}{4}+6$. ***

∗ ∗ ∗ wat doen we hieraan?

Domain-Specific Transformations

Because the editor has knowledge about the exact structure of a document, rather than just about the structure of the presentation, it is possible to specify domain specific mathematical transformations.

$x = a * (b + c)$

$x = a * b + a * c$

⇒ after distribution transformation

The example shows the application of a distribution transformation to the selected subformula. Similar transformations, such as factorize or reverse, may be specified by the editor designer or the editor user.

Requirements

Presenting mathematics puts a heavy demand on the presentation formalism. Fine control over automatic alignment and resizing of presentation objects is needed for complex presentations such as integrals, square roots and fractions.

Editing mathematics requires basic document edit support (copy and paste), as well as drag and drop editing. For supporting domain-specific transformations, a formalism for specifying document edit operations is needed. Presentation editing on mathematical formulae is desirable as well, but as of yet not a strict requirement, due to its still unclear nature.

1.2.4 Non-primitive Outline View/Tree Browser

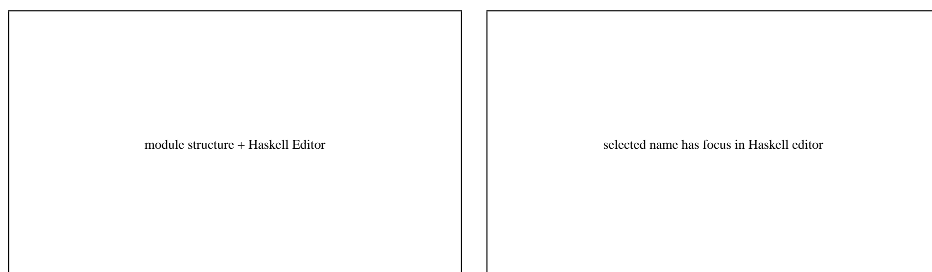
An outline view, or tree browser, is a hierarchical view on tree structures. It is found in the Java Swing GUI library and also forms the main navigation tool in Microsoft’s Windows Explorer application.

Tree Browser View

Some editors, especially XML editors, provide tree browser views on the document, but in all editors, the view is built-in. However, if the editor is sufficiently powerful to express a tree browser view without resorting to a primitive tree browser widget, this offers many possibilities for integrating the tree view with other views on the document, creating for example WYSIWYG views of parts a program source, in which branches can be hidden or shown. At the same time, it opens the door for using the generic editor for source module or document management.

Navigation

Tree views are useful for showing an overview of large structures, such as a program source that consists of a number of different modules.



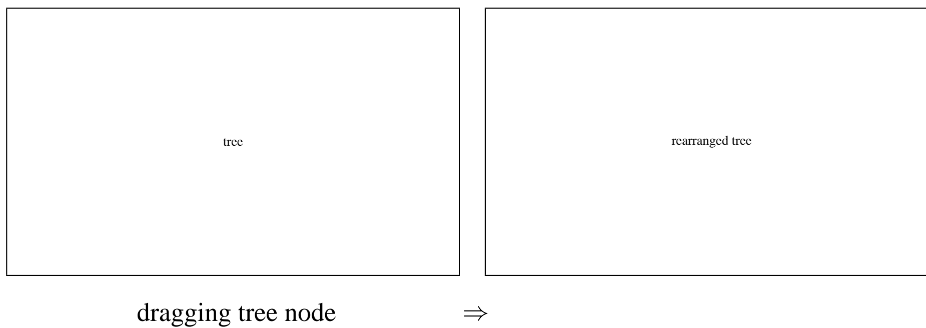
click on function name

The editor window consists of two panes, the right pane contains a Haskell source editor and the left pane contains a tree view of the module structure of the edited program. Below the module in the tree are the functions and types it defines. When the user clicks

on a name in the left pane, the corresponding module is shown in the right pane and the function definition or type declaration is brought into focus.

Drag and Drop

The tree browser supports drag and drop edit behaviour that allows nodes in the tree to be dragged to new locations.



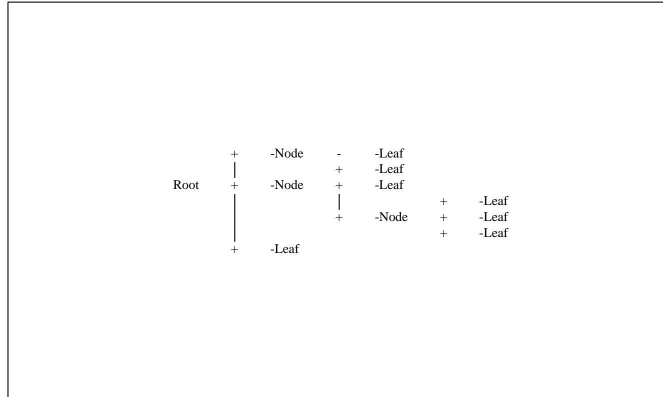
dragging tree node ⇒

The screenshot shows the effect of dragging the leaf with label “leaf 1” to its new position below “leaf 3”. The operation causes a document edit operation that moves the element whose presentation is “leaf 1” to become a child of the element that has presentation “node 1”, immediately after the element with presentation “leaf 3”.

In this example, the elements of the tree are all of equal type, and therefore can be moved anywhere in the structure. Using the tree view for outline editing in the word processor example is slightly more complex, because a move operation may first require a transformation on the moved element. For example, when a subsection is moved immediately under a chapter element, it must be changed to a section first.

Customized Tree Views

Because the tree presentation is not primitive, the editor designer or user can customize it, or even define entirely different tree presentations.



The tree view in the screenshot is a more spatious presentation, in which the child nodes are presented to the right of the parent rather than below.

Requirements

Similar to the equation editor, the tree browser has a two dimensional graphical presentation that requires fine control over the alignment of the presentation elements. Customizability of the tree view requires that the presentation specifications are transparent and reusable.

Edit operations on the tree structure are similar to edit operations on the table of contents in the word processing example, because the tree is typically a derived structure that follows the structure of the document (or part of it). Updates on the tree need to be mapped on updates on the document itself. Navigation operations can be considered as an update on the focus, and hence, the document edit specification formalism must support for focus updates.

An aspect specific to the tree browser is that it has a notion of state. Each node in the tree view is either collapsed or expanded and this information must be stored somewhere. Such presentation state, or *local state*, as we call it, does not belong in the document, because if it stored there, the document type will need to be changed whenever a tree view is added to the presentation. The fact that this state is not part of the document, but rather of the presentation of the document, makes it hard to model in a structure editor. A mapping between the document and the presentation needs to be maintained to associate a document node with its expansion state, even when the document is edited.

1.2.5 Tax Form

The last example is a tax form application, which is basically a spread-sheet with a rather specialized presentation. It contains questions and explanatory text, mixed with input fields and fields that contain derived information.

Income

Nr. of jobs	2		
Job nr.	Description	Total Salary	Tax Decucted
1	PhD student	10	2.50
2	Scientific programmer	20	5
Total income	30		
Total tax paid	7.50		
Interest:	2		
Total tax:	35% of income - paid =	**3.7**	

A tax form

A difference between the tax form and the previous use cases is the fact that the results of computations that are specified by the user form part of the presentation. Furthermore, the tax form requires a table-oriented layout with support for user interface widgets such as text fields, radio buttons, and selection lists.

The tax form knows two different kinds of users: the user that designs the tax form, and the user that fills out the form. Both users use the same document type, albeit with different presentations. The distinction between two kinds of users is a different one from the distinction in Section **??** between editor designer users and document editing users, because both tax form users edit the document and are therefore document editing users.

Presentation depending on document values

In most presentations, the structure of the presentation depends on the document structure. However, the presentation structure may also depend on a document value, rather than the structure. An example is the following section of the tax form:

Income			
Nr. of jobs	2		
Job nr.	Description	Total Salary	Tax Decucted
1	PhD student	10	2.50
2	Scientific programmer	20	5
Total income	30		
Total tax paid	7.50		
Interest:	2		
Total tax:	35% of income - paid =	**3.7**	

Income			
Nr. of jobs	3		
Job nr.	Description	Total Salary	Tax Decucted
1	PhD student	10	2.50
2	Scientific programmer	20	5
3	.	0	0
Total income	30		
Total tax paid	7.50		
Interest:	2		
Total tax:	35% of income - paid =	**3.7**	

⇒ Nr of jobs is changed is changed to 3

The number of input fields for job information depends on the number of jobs. When the number is increased, the structure of the input form changes accordingly, showing one input field extra. presentation structure depend on values in the document handling input in them.

The tax man view

A different presentation of the tax form allows the user to edit the structure of the tax form itself, rather than its input fields.

Income			
Nr. of jobs	C1: 2		
Job nr.	Description	Total Salary	Tax Deducted
1	PhD student	C2: 10	C5: 2.5
2	Scientific programmer	C4: 20	C6: 5
Total income:	C7: derived C2 + C4 *30*		
Total tax paid:	C8: derived C3 + C5 *7.50*		
Interest:	C9: 2		
Total tax:	35% of income - paid =	C10: derived 0.35*(C7+C9)-C4 ***3.7***	

Tax form for the tax man (draft)

The screenshot shows the same tax form document as the previous screenshot, but now the tax form structure and layout are editable. Text blocks, as well as input fields and derived value fields can be inserted or deleted, and the computations for the derived values can be specified. The labels (C1 ... C10) are supplied automatically, but can also be specified. The input values of the input fields are editable to allow for easy testing of the specified computations.

∗ ∗ ∗ labels of generated fields (salary and tax paid) need to be treated specially. How exactly is not clear yet.

Requirements

In contrast to the other use cases, the tax form presentation is rather similar to a user interface. Instead of just words and graphicals, it contains widgets, such as check boxes, selection lists and input fields, with the corresponding edit behavior. Although it may be possible to describe such widgets in a powerful enough presentation language, built-in support for widgets will be acceptable as well.

The tax form also features computations with results that appear explicitly in the document presentation itself. Unlike the type computations in the Haskell editor, the computations in the tax form can be specified by the editor user (the tax man), rather than the editor designer. Therefore, the editor needs to be able to support spread-sheet behaviour.

1.3 Functional Requirements

With the use cases of the previous section in mind, we now discuss a number of functional requirements for a generic structure editor.

1.3.1 Genericity

In order to support the five use cases, the editor must be generic in the sense that it is not built for a specific document type, or class of document types. However, we do restrict ourselves to trees rather than graphs. Most documents can be represented by trees, including the five use cases. A formalism for specifying links between tree nodes is desirable, but full graph editing is not a requirement.

Although a distinction can be made between editor generators and generic editors, we regard both kinds of system as generic.

1.3.2 Computations over the Document

An interesting aspect of an editor that has knowledge of the structure of the document, is that it can show computed or derived values over that structure to the user. Examples of computations are automatic chapter numbering and a derived table of contents, but also derived type information for identifiers and function definitions in a program source. Two aspects have an influence on the usefulness of the computations: the strength of the formalism in which the computations are specified, and the integration of computed values and structures with the document presentation.

For program editing as well as the tax form, the strength of the computation formalism is important. Computations can provide static analysis, e.g. detecting name clashes and scoping problems, as well as a type derivation. In order to be able to specify these computations for arbitrary languages, a turing complete formalism, such as an attribute grammar [**?**], is desirable. Another option is to allow the specification of constraints on

the document tree [?] and have the editor check automatically whether the constraints hold, but this is not as powerful. Moreover, in such a constraint based system it is hard to specify numberings and derived structures in an elegant way.

For the word processing example, as well as the tax form, the integration of computed values with the document presentation is important. Whereas type errors may be shown in separate windows or by underlining the location and showing the message in a tooltip, chapter numbers and a table of contents form an actual part of the presentation.

1.3.3 Presentation Formalism

The presentation formalism has two different aspects that we consider together here. One is the formalism in which the building blocks of the presentation are expressed (the presentation target language), whereas the other is the formalism in which the mapping from the document onto the presentation target language is expressed (the presentation transformation language). For XML, a well known presentation language is the Extensible Stylesheet Language (XSL) [?], it is split into the mapping language XSLT and the target language XSL Formatting Objects. Chapter **??** discusses the two aspects of the presentation formalism in more detail.

In many editors the target language is just plain text, sometimes with some color and font attributes. However, in order to support the graphical renderings of the equation editor and outline view use cases, a more advanced target formalism is required. It must be possible to specify graphical elements such as lines and rectangles, as well as to show images. Furthermore, the presentation of a mathematical formula requires an advanced alignment model that offers full control over the positioning of presentation elements.

Another requirement for the presentation target language comes from the tax form example. The tax form typically contains user interface widgets, such as radio buttons, selection lists, menus, and normal buttons. Therefore, the target language must support user interface widgets.

Finally, the word processor use case requires that the presentation target language supports line and page breaking, preferably optimal [?].

The presentation transformation language has to support the specification of complex graphical presentations with compact readable style sheets. It must be possible to specify simple presentations in an easy way, while still allowing the specification of more complex presentations.

Although the transformation language is related to the computation formalism, since a presentation can be seen as a computed value, we do separate the two. One reason is that separation of computation and presentation makes it possible to specify multiple presentations of a document together with its computed values. Another reason is that the separation makes it easier to support edit operations on derived structures.

1.3.4 Editing Power

The editing power of an editor is determined by the levels at which edit operations can be targeted, together with the complexity of the edit operations and the extent in which they are user specifyable. Levels at which editing is possible are the presentation level and document level. Syntax-directed editors mainly offer edit operations targeted at the document level, whereas syntax-recognizing editors edit the presentation level. Both levels, however, are important. Furthermore, as we will show in Chapter 2, a number of other levels may be distinguished, among which a level that contains derived structures and values.

The program editor use case shows the necessity for free textual editing in the presentation. Purists argue that text editing may introduce syntactic errors, and that it is not necessary for programming (e.g. [?, **?**]). However, no clear consensus has been reached on the subject (e.g. [**?**] and reactions [**?**, **?**], and [**?**]) and nowadays even most syntax-directed editors support some form of free text editing. Furthermore, because up to now, no pure structure editor has ever become popular with programmers, we believe that free textual editing is an essential requirement for program editing.

Besides presentation oriented-edit operations, document-oriented operations are important as well. The equation editor as well as the outline editor rely heavily on document editing. For equation editing, structure entry is typically a document edit operation, because many expression structures, such as a quotient, a power expression, or a square root, have no presentation that can easily be entered with conventional editing methods. The outline editor requires document editing because it has to support navigation over the structure, as well as support moving nodes in the tree to other locations, which is a document oriented operation. Document edit operations typically include basic copy, paste, and delete operations, as well as selection and navigation operations.

Especially for document edit operations, a transformation specification formalism is desirable. It allows the editor designer to define edit operations specific to the edited type of document. An example of such a user specified edit operation is the rename operation in the Haskell editor. Furthermore, the formalism can be used to specify standard generic document edit operations such as split and join.

As the word processing use case shows, it can be desirable to support edit operations on derived structures. This is not to say that all derived structures and values should be editable, but in those cases in which it makes sense to a user, it should be possible to specify the edit behaviour for derived structures.

1.3.5 Modeless Editing

Besides support for editing on different levels, an important requirement is the integration of the edit operations on the different levels. A seamless integration of document and presentation editing provides a more pleasant edit interface to the user, as the intended operation can be performed on the presentation that the user is working on, without first

explicitly having to switch modes.

The most extreme form of mode-switching is when different-level edit operations have to take place in separate windows and also have a separate undo-history. This is the approach taken by many pure editors that offer some support for free text editing, as well as by all existing XML editors. Even worse, the separate free-editing text mode often has a special text-only format, in which derived values are not shown and interesting graphical presentations are not possible. In order to get back to document editing, the user needs to leave the text in a valid state, or abandon the text update.

If the editable textual presentation is displayed in-place in the document presentation, the mode-switching becomes less intrusive. However, the most user-friendly approach is to avoid mode switching altogether, so that a user can freely edit in the presentation, even when it contains computations and graphical presentations. Moreover, if a presentation edit makes the presentation invalid, the invalid area should be kept as small as possible, and document editing must still be available on the valid parts.

1.3.6 Local State

A feature that is not explicitly present in any of the current structure editors, is the notion of local state. With local state, we mean information that needs to be kept track of when editing a document, but which does not conceptually belong to the document itself.

A clear example of local state is present in the outline view example. The expansion state of the nodes of the tree view needs to be kept track of. However, this is not information that should be stored in the document tree structure, since the design of the document type should not have to consider what views may be defined for that document type. Moreover, several views may be opened simultaneously, each with their own expansion state. Other examples of local state are focus information, local layout settings (e.g. whether or not auto-layout is turned on), and whitespace in the presentation. These examples all concern local state at presentation level, but local state also appears on other levels, as will be shown in Chapter **??**. In order to handle local state declaration, a formalism must be present to declare variables local to presentation elements. Furthermore, when the editor application is closed, and reopened, it must be possible to preserve the local state.

Support for local state complicates the presentation process as well as the translation of edit events onto document updates. A document element needs to keep track of its presentation elements, and when it is re-presented, it must be mapped onto those same presentation elements because local state may be associated with the presentation. The editor needs to have facilities for easily maintaining the required mappings between a document and its presentation to keep track of the local state, and when no local state is used, no extra effort should be required from the editor designer.

1.3.7 Summary

Summarizing, to support all five use cases, a generic structure editor must meet the following requirements.

- Genericity
- Support for Turing-complete computations over the document
- A graphical presentation language with a powerful mapping formalism.
- Support for edit operations on all levels, including edit operations on derived structures.
- Modeless editing
- Support for local state

1.4 Overview of Existing Editors

Because of the large number of existing systems, we cannot mention all editors in this overview. The editors mentioned are some of the early systems, together with a number of other editors that contain novel features.

1.4.1 Syntax-Directed Editors

Most of the editors in this section are specially designed for program editing and hence have a rather text-oriented presentation formalism. Moreover, the computation formalism in such editors is aimed mainly at analysing source code, instead of performing general purpose computations.

Synthesizer Generator

The Synthesizer Generator [?] is the successor of the Cornell Program Synthesizer [**?**], one of the early syntax-directed editors. Because the system is targeted at programming languages, the presentation is simple and text-only, although newer versions have some font and color control.

An interesting aspect of the Synthesizer Generator is its support for computations over the document structure. The presentation of the document can contain computed values, that are specified using an attribute grammar.

The edit model supports user specified transformations on the structure, but plain text editing is poorly supported. The editor uses mode-switching and after switching to textual mode, the presentation must be put in a parseable state before any structure editing is available again.

Over the years, the behaviour and design have not undergone many drastic changes, but the system is still being used and commercially maintained.

LRC

The LRC attribute grammar system [?] was a research project at Utrecht University. Higher-order attribute grammars are used to specify the derived values, as well as the presentation. The system is based on an efficient higher-order attribute grammar evaluator. Higher-order attribute grammars allow some computations to be specified more elegantly than normal attribute grammars.

For the presentation of the document, the Tcl/Tk language is used. This allows for complex presentations with multiple windows, GUI widgets, colors, and basic graphical elements. However, the integration between the generated presentation and the editor is very weak. No general focus model is present, and although edit events can be attached to the Tcl presentation, free editing is only possible a separate window that contains a purely textual presentation of the document. The textual presentation cannot be used to edit the layout of the main presentation, and it does not contain derived values.

LRC is still being used in a number of commercial settings, but is not maintained or developed anymore.

SbyS, Mjölner Orm

SbyS [?] is the structure editor from the Mjölner Orm environment. Mjölner Orm is a generic language and software development environment. An interesting aspect of the environment is that it is truly a generic environment, since language descriptions can be changed without the need to recompile or regenerate the editor. In contrast, most other systems are editor generators, that generate an editor for a specific language.

The structure editor is syntax-directed text and text-oriented with a parser only for entering expressions. In order to overcome the usability problems associated with pure syntax-directed editing, the editor employs the concept of direct manipulation. Program constructs are shown in a pallette, from which they can be dragged to the program source or a clipboard.

No formalism for specifying transformations is present, and the only computations that can be specified are aimed at semantic analysis and code generation. Derived values cannot be part of the presentation.

PSG

PSG (Programming System Generator) [**?**] is a generator for language based interactive environments, developed at the Technical University of Darmstadt. As the name already suggests, the system is designed for programming languages. The presentations are text-only and only LL(1) grammars are supported. The system generates an editor based on a number of formal descriptions for a language, including a syntax definition, a presentation sheet (called a *format syntax* in PSG), and a specification of the semantic analysis.

Special focus has been put on incremental analysis over incomplete program fragments. PSG uses a special form of the attribute grammar formalism that supports sets of possible attribute values in order to handle attribution of incomplete document fragments.

However, the presentation may not contain derived values or structures. And although textual editing takes place in the same view as document editing, this does involve a mode switch. Furthermore, layout information cannot be edited freely, but is determined by the presentation sheet.

Other syntax-directed editors

Other textual syntax-directed editors for program editing are the Aloe editor in Gandalf environment [], Mentor [], its successor Centaur [], Pregmatic [] Poe [] , Dose [] Gnome [], Pecan [], Muir [], Dice []. These systems have their own interesting aspects, but as far as the editors are concerned they do not deviate much from the systems already discussed, and hence are not discussed separately.

Some more exotic editors, that do not support editing on the presentation are Multiview [] and VL-Eli [].

1.4.2 Syntax-Recognizing Editors

Similar to the syntax-directed editors, most syntax-recognizing editors are designed for program editing. Regarding the computations, however, due to problems with free editing of presentations with derived values, none of the syntax-recognizing editors support arbitrary computations that may appear in the presentation.

Pan

Pan [**?**] is a text-only program editor environment The presentations are text with multiple fonts, styles, and colors. The system has good support for handling partially incorrect or incomplete documents.

The computation formalisms in Pan are oriented towards semantic analysis. Logical constraint grammars are used for specifying, checking, and maintenaining of contextual constraints. Computed information is shown in the presentation by changing the font and color attributes of the text, but it is not possible to specify arbitrary computations that form part of the document presentation. Furthermore, it is not possible to specify an editable presentation that shows only part of the document (eg. a presentation in which function bodies may be hidden), as the editor is syntax-recognizing, and therefore the presentation must contain all information necessary to derive the document structure.

Pan offers some document editing, but edit operations on document structures are performed by editing the corresponding parts in the presentation and reparsing the presentation. Edit operations that modify the document structure directly are not supported, as these are believed to confuse the user. As a consequence, only basic document edit

operations such as cut and paste are supported, and no document transformations can be specified. Free text editing, on the other hand, is fully supported, including layout editing.

GSE, ASF/SDF

The GSE [?] editor has been developed as part of the Esprit project "Generation of Interactive Programming Environments" (GIPE). It is still being used in the ASF/SDF meta environment. [**?**] The editor is primarily aimed at programming languages and the presentations are assumed to be lines of text. GSE supports free editing of the program text without an explicit mode switch, but structural edit operations on the program that keep user specified layout in tact are not supported. Also, computations on the document cannot be specified.

Ensemble

The Ensemble project is a successor to Pan, based on the recognition that structure editing cannot only be used for program editing, but also for editing documents of a more graphical nature, such as documentation. The system handles compound documents containing subdocuments of different types, and provides document management functionality, such as versioning.

Ensemble specifies formalisms for performing incremental semantic analysis, but arbitrary computations appearing in the presentation cannot be specified. However, some support for derived structures is present in the presentation formalism.

Ensemble has a powerful graphical presentation formalism, including a constraint based box layout. The presentation transformation language, however, does not elegantly allow presentations with a different structure from the document. The presentation formalism may be used to specify derived structures, but these are not editable.

The edit model supports modeless free text editing, including layout editing, as well as structural editing.

The Ensemble project has been terminated, but its successor, Harmonia [**?**], is still under development. Because the monolithic character and ambitious design requirements of Ensemble slowed down its development, Harmonia is a framework for incremental language analysis rather than a single editor generator. The services from Harmonia can be used to augment text editors, such as Emacs, with language-aware editing and navigation functionality.

Desert

Built using the experience of the FIELD [**?**] project, Desert [**?**] is a syntax-recognizing editor generator that uses the commercial editor system Framemaker for editing program sources. The system has many facilities for software development, including database facilities and an interface for easily defining (non-editable) software visualizations. The actual editor is a syntax-recognizing editor with attributed text and images in the presen-

tation. However, no structural edit operations, or derived structures in the presentation are supported.

Other syntax-recognizing editors

Other syntax-recognizing editors similar to the ones that were discussed include Babel [?], Saga [?], and Pregmatic [?].

1.4.3 Editor Toolkits

Besides generic editors and edit generators, an editor can also be built using an editor toolkit. The toolkit is a collection of libaries and tools that can be used when building an editor. The editor application itself, however, has to be written by hand. The separation between a toolkit from a generator is not always completely clear, since the specifications that an editor generator uses for specifying language, presentation, and semantics can be considered programs as well. The toolkits we consider here, require a substantial amount of programming in order to build an editor.

The advantage of a toolkit is that the final editor can be customized to a high degree, but this comes at the cost of the increased effort required for building an editor.

Amaya, Thot

Thot is probably also a generic editor, not just a toolkit

Amaya [?] is the W3C web browser that is built on top of the editor toolkit Thot [?], which is a successor of GRIF [?]. The Thot toolkit supports a number of specification languages for document structure, presentation, and transformation, but in order to build an actual editor C code is required to connect the various components***.

*** can it be done without?

The presentation formalism in Thot, called P, is a powerful graphical presentation formalism, somewhat similar to Proteus (Ensemble), but with more advanced alignment features. As a result, complex presentations are possible, such as the presentation for equation editor use case.

Thot editors are of a syntax-directed nature. Multiple views on the document may be edited simultaneously, and user specified transformations are supported. However, free text editing can only be done in a separate window in a different mode. Also, no computations are supported, other than some basic counters in the presentation.

Visual Studio model

The Microsoft Visual Studio environment includes an integrated program editor. Although the editor does not contain any novel features, and thousands of lines of code need to be written to tailor the editor for a specific language, we do include it in the dis-

cussion because it is a structure editor that is actually used by a rather large number of people.

The Visual Studio editor is of the syntax-recognizing kind with colored text presentations. No document edit functionality is supported, other than the displaying results of semantic analyses. The visual mechanism for displaying the results is by marking a location in the source presentation with a squiggly line, and showing a corresponding message in a separate window pane as well as in a tooltip. Pop-up list boxes can be used to show auto-completion alternatives. Despite its simple model, in which semantic analysis is only possible when the entire presentation is syntactically valid, the editor provides a surprisingly usable environment.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑

Other editor toolkits

First find out more about them.
Xemacs
Andrew system
Opendoc.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

1.4.4 XML Editors

A large number of XML editors has been developed, but the differences between them are not fundamental. Almost all XML editors classify as pure structure editors with mode switching.

Because of the little amount of variation, we will first discuss how XML editors in general according to our functional requirements. Afterwards, two *** editors are discussed separately.

∗ ∗ ∗ three?

Genericity. The XML editors have no problem with genericity. Most reviewed editors are actual generic editors, rather than editor generators, and support editing of documents with arbitrary DTDs. Although the DTDs have a few restrictions in order to make parsing easier [**?**], the type language is very similar to the EBNF grammar description formalism and powerful enough to describe the tree based document structures we wish to edit.

∗ ∗ ∗ What about schemas?

Computations over the document. Support for computations is very weak for all reviewed editors. A few editors support basic numbering of elements in the docu-

ment, but no arbitrary computations can be specified. Some editors support the transformation formalism XSLT, but the view on the transformed document is not editable in any of them.

Presentation formalism. Most XML editors provide only standard views on the document. Popular are the raw-text XML source view, a built-in tree view showing the document structure with PCDATA values in the leafs, and a slightly less raw view with tags, represented using a more graphical presentation.

Some editors support a user definable presentation, or at least allow the user to specify some attributes for the presentation. However, the presentation formalisms are generally weak, and the presentations that can be used for editing, have to follow the structure of the XML document. Moreover, there is hardly any support for textual presentations, making it impossible to present an XML tree representing the abstract syntax tree of a program as the actual program text.

It is remarkable that support for textual presentations of XML documents is this weak, since many languages for processing and describing XML documents are specified in XML itself (for example XML Schema or XSLT) and editing these languages would be greatly simplified by providing the user with a concise concrete syntax, rather than the verbose XML syntax.

Editing power. Most XML editors offer simple document edit operations for structure entry and manipulation. However, none of the reviewed editors support user specified transformations on the tree structure.

In each of the editors, free text editing is supported only in the raw XML source. Because most XML documents have text and whitespace in the leafs, it may appear that the document edit operations are free text editing, but this is not the case. Textual presentations other than the source presentation cannot be edited freely. On the other hand, as mentioned, most XML editors do not allow a textual presentation of the document.

Modeless editing. None of the editors support free editing on the presentation without a mode switch. Each type of view has a separate window, and though some editors have a shared undo history for some of the views, no editor has a shared undo history for the XML source presentation and its other presentations. Hence, after switching to source mode, previous edit operations on other views cannot be undone anymore, and vice versa.

Local state. The XML editors do not support user specified local state.

Two XML editors have a more sophisticated presentation engine and some basic support for computations, and are therefore discussed separately.

X-Metal

The commercial system X-Metal from SoftQuad is a highly customizable XML editor, with support for many XML standards and database connectivity. Besides regular source

Editor	Genericity	Computations	Presentation formalism	Editing power	Modeless editing	Local state
Synthesizer Generator	++	++	+/-	+	- -	- -
LRC	++	++	+	+	- -	- -
PSG	++	+	- -	?+/-	+	- -
SbyS/MjölnerOrm	++	-	- -	-	n/a	- -
Pan	++	+/-	-	+/-	++	- -
GSE	++	- -	- -	+/-	++	- -
Desert	++	- -	+/-	+/-	- -	- -
Ensemble	++	+/-	+	+	++	- -
Thot/Amaya	+	+/-	+	+	- -	- -
VisualStudio	+/-	+/-	-	-	n/a	- -
XMetal	++	-	+/-	+/-	- -	- -
XMLSpy	++	+/-	+	+/-	- -	- -
Other XML editors	++	max. -	max. +/-	+/-	- -	- -
Proxima	++	++	++	++	++	++

Figure 1.4: Editor evaluation

and outline views, the editor offers built-in table editing and an editable CSS presentation of the document. CSS provides a quick and easy way to specify a document presentation, but its power is limited. General computations cannot be specified, but CSS does allow the specification of basic counters in the presentation.

Document edit operations in X-Metal are rather weak, and transformations cannot be specified. Furthermore, the freely editable source presentation can only be edited in a separate mode.

XML Spy

XML-Spy is a large system that has similar functionality as X-Metal. An important difference is the presentation system. XML-Spy supports a larger number of built-in presentations and also has a user-defined presentation that supports the specification of simple derived document structures. Values from the document that appear in the derived structure may be edited in place.

1.5 Discussion

Figure **??** contains a score table for all discussed editors.

In the table of the previous section, none of the existing systems has a line that contains only plusses. No editor supports local state, but besides that, each of the editors has at least one column with a low score (+/- or less). One of the reasons for this is that the requirements for computations and a powerful presentation formalism are difficult to reconcile with the requirements for editing power and modelessness.

The first two requirements determine the presentation complexity of the editor, whereas the last two determine the useability of the editor. A problem is that the more complex a presentation is, the harder it will be to still offer modeless free editing on the presentation level.

Syntax-Directed Editors. The syntax-directed editors tend to do well on the computation requirement, but at the same time, presentation editing is weakly supported, leading to a lower score on editing power, and modelessness is not supported at all. However, if the presentation formalism is simple, and no computations appear in it, then modelessness can be supported (see PSG).

Syntax-Recognizing Editors. The syntax-recognizing editors, on the other hand do well on the presentation editing and modelessness requirements, but the fact that a document is derived from its presentation has a number of consequences. Firstly, the presentation must at all times contain enough information to derive the document, which puts restrictions on the presentation formalism. Secondly, having derived values and structures in the presentation makes parsing a lot harder and is therefore not supported, hence the low scores on the computation requirement. And finally, edit operations on the document are harder to implement. As a result, syntax-recognizing editors will not score maximally in the computation, presentation, and edit power columns.

XML Editors. XML editors are similar to syntax-directed editors, but somehow the computation and presentation formalisms are not very well developed. Semantic analysis, is of course not a big requirement for an XML editor, but computations and derived structures have many applications also for XML editing. Furthermore, specification of a textual presentation with a parser is not supported, which is odd because the raw XML source has an extremely verbose syntax that is not very suitable for viewing or editing directly.

Although some XML editors have support for graphical presentations, the presentation transformation formalisms are generally weak, disallowing the structure of the presentation to be different from the structure of the document. Hence, there is a strong connection between an XML document and its presentation. A tree structured document with text in the leafs lends itself well for editing with an XML editor, but other structures are hard or impossible to edit. Examples are an XML representation of an abstract syntax tree, or a paragraph that is represented by a list of word elements. Current XML editors cannot handle such documents.

The close link between the XML document and its presentation sustains the view that an XML document is a piece of text with markup tags added to it. In this view, the current XML editors provide sufficient edit functionality. However, if a more powerful editor is available, which releases the tight connection between a document and its presentation, the view might change, causing new applications for XML to arise.

Because the discussed structure editors are evaluated only with respect to requirements for the edit model, some of the systems look rather bad. Partly, this is due to the fact that these systems were designed with a large number of other requirements in mind, which are

not taken into account here because they are concerned more with the environment than with the editor. Structure editors often have many facilities for managing and versioning documents, as well as complex semantic analysis methods, whereas XML editors often support built-in XSLT viewers, DTD viewers and editors, and database connectivity, as well as support for the many standards existing in the XML world. However, we view these requirements not as essential for the design of a generic structure editor.

Summarizing, the current and previous generations of structure editors are not powerful enough to edit the five use cases of Section **??**. The editors either lack flexibility to express the required presentations, or have an edit model that is overly restrictive, or suffer from both of these problems. In the next Section, we introduce our solution to this situation.

1.6 The Proxima Editor

Proxima is a generic structure editor that can handle all five use cases from Section **??**. It meets the requirements from Section **??**.

The Proxima editor uses the attribute grammar formalism for performing semantic analysis, as well as the specification of derived document structures and values, which may appear in the presentation. The presentation formalism supports graphical presentations, and a box layout model with alignment, strong enough to specify presentations of mathematical equations. Furthermore, edit operations may be targeted at both presentation and document level, as well as at derived document structures, without mode switching.

In order to support the edit operations on multiple levels, the editor keeps track of bidirectional mappings between the document and its presentations. A layered architecture, which breaks up the presentation process, as well as the handling of edit operations, in a number of steps, facilitates the process of keeping the mappings consistent. The problem that a higher level does not contain enough information to compute the lower level (eg. when the document does not contain the whitespace of the presentation) is handled by storing the required information as local state on the lower level.

An editor in Proxima is specified by a number of sheets that specify the computations, the presentation, the parser (inverse of the presentation), and the reducer (for handling edit operations on derived values and structures). The languages of the editor sheets are declarative and have a strong abstraction formalism, which helps to keep the specification of simple behavior short, while still allowing the specification of complex behavior as well.

To accomplish the requirements, Proxima makes use of the following concepts:

- A layered architecture
- Bidirectional mappings between document and presentation
- Concept of local state on several levels of the presentation process

- Declarative specification languages with strong abstraction mechanisms for specifying mappings between levels

Of course, many more requirements exist, but our focus is on the editing model. The computation model of Proxima is general, and in order to easily use it for example to do semantic analysis or code generation, libraries are required. Other requirements that we consider orthogonal to ours concern document management and database connectivity. Also, we have not paid much attention to incrementality. With current computer speeds, parsers are so fast that incremental parsing has become less of an issue. We expect that a rather coarse model for incrementality in the presentation will be sufficient for creating fast editors.

Many of the features in Proxima are optional rather than enforced. Edit operations on derived structures may be specified or automatically derived in cases for which they make sense, but if this is not the case, the editor designer need not specify them. A similar thing holds for the local state. Supporting local state in a Proxima application puts some effort on the editor designer, but if no local state is present, then the editor designer does not need to take it in account.