Master’s thesis

Formalisation of edit operations for structure editors

by

Johan Holmquist

LITH-IDA-EX--05/015--SE

2005-09-06
Master’s thesis

Formalisation of edit operations for structure editors

by Johan Holmquist

LITH-IDA-EX--05/015--SE

Supervisor: MSC Robert Wensman
Wensman Software, Linköping

Examiner: Associate Professor Anders Haraldsson
Dept. of Computer and Information Science
at Linköpings universitet
Although several systems with structure editors have been built, no model exist to formally describe the edit operations used in such editors. This thesis introduces such a model — a formalism to describe general structure edit operations for text oriented documents. The model allows free bottom-up editing for any tree-based structural document with a textual content. It can also handle attribute and erroneous structures. Some classes of common structures have been identified and structure editor specifications constructed for them, which can be used and combined in the creation of other structure editors.
Abstract

Although several systems with structure editors have been built, no model exist to formally describe the edit operations used in such editors. This thesis introduces such a model — a formalism to describe general structure edit operations for text oriented documents. The model allows free bottom-up editing for any tree-based structural document with a textual content. It can also handle attribute and erroneous structures. Some classes of common structures have been identified and structure editor specifications constructed for them, which can be used and combined in the creation of other structure editors.

Keywords: Structure editor, bottom-up editing, top-down editing, syntax recognizing editor, syntax directed editor, hybrid editor
Acknowledgements

Many thanks to my supervisor, Robert Wensman, who has been a great source of inspiration during this work and has always shown faith in my methods and solutions. Also many thanks to my examiner, Anders Haraldsson, for his everlasting energy and enthusiasm.
Contents

1 Introduction 1
  1.1 Purpose ................................................. 2
  1.2 Thesis outline ......................................... 2

2 Background to structure editing 4
  2.1 Structured documents ................................... 4
  2.2 Structure editors ....................................... 5
    2.2.1 Advantages of structure editors .................. 7
    2.2.2 Problems with structure editors .................. 8
    2.2.3 Classes of structure editors ...................... 9

3 Formalising edit operations 10
  3.1 Requirements .......................................... 10
  3.2 Overview ............................................. 12
  3.3 The document tree ..................................... 12
    3.3.1 Up and down in the tree .......................... 13
    3.3.2 Focus ........................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3</td>
<td>Simulating a cursor with focused symbols</td>
<td>15</td>
</tr>
<tr>
<td>3.3.4</td>
<td>The invisible $ symbol</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Structures</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Editors</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>Modelling focus</td>
<td>19</td>
</tr>
<tr>
<td>3.7</td>
<td>Events</td>
<td>20</td>
</tr>
<tr>
<td>3.8</td>
<td>Edit operation rules</td>
<td>20</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Edit rule patterns</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Matching focused elements</td>
<td>23</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Patterns in rules</td>
<td>23</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Edit rule actions</td>
<td>24</td>
</tr>
<tr>
<td>3.9</td>
<td>Common edit operations</td>
<td>24</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Edit functions</td>
<td>25</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Edit rules</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>Complementary edit operations</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Editor examples</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>The Lex editor</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1</td>
<td>The edit functions</td>
<td>29</td>
</tr>
<tr>
<td>4.1.2</td>
<td>The edit rules</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>A Line editor</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The edit functions</td>
<td>31</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The edit rules</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>A Text editor</td>
<td>34</td>
</tr>
<tr>
<td>4.3.1</td>
<td>The edit functions</td>
<td>34</td>
</tr>
</tbody>
</table>
4.3.2 The edit rules ................. 34
4.4 A comma separated list .............. 35
4.4.1 Edit functions .................. 35
4.4.2 Edit rules ..................... 36

5 Extensions and difficulties .......... 38
5.1 Lexical editors and subtyping ........ 38
5.1.1 Example ...................... 39
      A Num editor .................. 39
      A Ident editor ................ 40
5.2 Attributes ........................ 41
      5.2.1 Using attributes in edit rules 43
      5.2.2 Considerations ............ 44
      5.2.3 An FText editor ........... 46
5.3 Comments ........................ 48
5.4 Selection ........................ 49
5.5 Alternation ....................... 51
      5.5.1 The simplest case of alternation 51
      Example of the use of getlex ........ 52
      5.5.2 The general case of alternation 53
      5.5.3 Dealing with errors ........ 54
      Parser requirements .......... 54
5.6 Classes of editors ................ 55

6 Conclusions ....................... 57
A  Editor specifications  59
   A.1  Standard editors ................................... 59
      A.1.1  The default editor $D$ .......................... 59
      A.1.2  The lexical editor Lex .......................... 61
      A.1.3  The sequence editor Seq ........................ 62
      A.1.4  The list editor List .............................. 63
   A.2  A Lisp editor ........................................ 64

Bibliography  66
Editors for many different kinds of documents exist today; word processors, source code editors and editors for graphical data such as images and vector graphics. This document will mainly focus on editors for documents with some kind of text oriented structure, that is documents that are built from characters. Examples of editors that belong to this category are source code editors and word processors.

In order to make effective use of an editor, it should be aware of the structure of the document, so it can provide editing abilities over the structure of the document and not just character based operations. Most editors do not have this awareness and therefore do not offer much more than simple character based editing facilities. But efforts have been made to create so called structure editors who knows about the structure of the edited document in order to provide the user with more powerful edit operations. Unfortunately structure editors have never really made it to the every day users. Instead the more familiar plain text editors (like Emacs [1]) have gotten more and more sophisticated and offer some limited amount of structure oriented editing functionality, like moving across words and definitions etc. However this behaviour is only simulated, often by some error prone searching, so in order to get true structure editing we would need structure
editors. For a more thorough discussion about structure editors and their possibilities see [2].

1.1 Purpose

This work is based on the observation that no formal model exists to describe the behaviour of general structure editors. The main purpose of this work is therefore to find a way to formally specify how edit operations will change the structure of the document in a general, text oriented structure editor.

A second purpose is to find a common set of edit operations, which can be used in the creation of any typical editor for text based documents — preferably source code documents.

The formal model and the operations should meet the requirements specified in section 3.1.

1.2 Thesis outline

This thesis will begin with an introduction and background to structure editing in the next chapter. The concept is explained together with it’s advantages and difficulties and some existing structure editors are presented in short.

In chapter 3 the formal model, which is the result of this work, is described along with some common edit operations. The following chapter then demonstrates the use of the formalism by specifying some simple editors.

Then, in chapter 5, follows a discussion about how to incorporate several tricky concepts into the model. Some extensions are proposed in order to make the formalism powerful enough to enable specification of arbitrary, text oriented structure editors.
Conclusions and results are presented in the last chapter. Finally, an appendix is included with specifications of some common editors. Enjoy!
Chapter 2

Background to structure editing

This chapter will give a brief overview of structure editing. It will also explain the concept of structure editors, their advantages and their typical problems.

2.1 Structured documents

In practice any document has some form of inherent structure. A simple text document can be seen as a sequence of characters, but it may often feel natural to refer to higher level building blocks of the text, such as words, lines and paragraphs. When editing text in word processors we also talk about sections and chapters, which are all considered parts of the document we are editing. Those parts constitute the document’s structure. The same applies to source code documents in which the parts are typically the different constructs of the source programming language such as assignments, if-statements and definitions.
The structure of most documents can be modelled by a tree, with the root element representing the whole document and branches representing parts and sub-parts. The leaf nodes will represent the smallest, atomic parts of the document, like letters for a text document and characters or perhaps other symbols for a source code document.

In many cases a tree representation of a document is not enough. A text document with cross references between different parts of the text, would typically have to be represented by a general graph instead of a simple tree. In source code documents it is common to refer to the same variable from many different places in the code. In order to model such a document appropriately, a DAG representation would be needed, so that each occurrence of the variable would refer to the same object representing the variable. However, this thesis will stick to the simple tree representation since it is powerful enough for most uses and there is no requirement for modelling this kind of multiple referencing to single structure objects.

2.2 Structure editors

The term structure editor is often used to emphasize that an editor provides editing facilities on higher level structures, that is structures that in turn may have sub-structures. For example a word processor that lets you swap two chapters would be a structure editor since the chapters would in turn consist of lower level parts (like paragraphs, words etc).

Structure editors typically offer a special view of the edited document, that may change during the course of editing. Figure 2.1 shows the main components of a structure editor — the document representation (depicted as a tree) and the document view (presented to the user), that are kept synchronized. The user can perform edit operations to the presentation or directly to the document tree, depending on the functionality offered by the editor.

One of the early attempts to build a structure editor is Mentor [3], constructed in the mid 1970’s. It is a collection of tools for editing structured information. It has a tree manipulation language, Mentol, which operates
2.2. Structure editors

Figure 2.1: The main components of structure editors.

on abstract syntax trees (AST). There is a set of commands to move a locator to different nodes in the AST, print and modify parts of it. It is possible to write patterns to match parts of the AST which can then be operated upon (printed, modified etc.). An editor for a specific document is obtained by feeding the system with syntax tables for the document and possibly define special commands for construction and modification. An editor for Pascal programs has been constructed this way.

Despite it’s old age, Mentor is rather interesting since the idea of using a tree manipulation language with pattern matching facilities is similar to the method described in this thesis.

One of the early ideas with structure editors for source code documents (programs), was to keep the syntax of the edited document correct all the time during editing. In the creation of The Cornell Program Synthesizer [4] it was established that programs were not text, but hierarchical compositions of structures. Therefore code fragments should be inserted top-down by selecting a placeholder for some (unexpanded) structure and expand it by inserting templates for new code fragments. In this way it was possible to ensure that no syntactic errors were introduced in the code. However, this would later be considered a disadvantage from a usability viewpoint (see section 2.2.2).

The Cornell Program Synthesizer was more than an editor. It was a whole programming environment (PE), with other facilities in addition to editing,
such as debugging. Because programs were maintained as structures, they
could be interpreted on the fly, and hence time expensive parsings were not
necessary in order to execute and debug programs. Other systems, such as
\textit{Gandalf} \cite{5}, also explored ways of integrating editors with execution and
debugging facilities.

During the 80’s, efforts to construct structure editors peaked. Now \textit{The
Synthesizer Generator} \cite{6} was constructed as the successor to \textit{The Cornell
Program Synthesizer}. It was constructed around the same basic ideas, but
had also functionality for semantic analysis, such as type checking. Like
many systems, it was actually an \textit{editor generator} rather than an editor.
An editor generator is a system for generating a whole class of editors given
some specification. The above mentioned \textit{Mentor} is also such a system.

Most structure editors have been constructed for source code editing alone.
The \textit{Proxima} \cite{7} editor is an attempt to build an editor for many different
kinds of documents. Some of it’s uses mentioned in addition to source code
editing are spreadsheet editing and word processing.

\subsection*{2.2.1 Advantages of structure editors}

Structure editors offer several advantages over pure text editors. Some of
those advantages are listed below:

\textbf{Graphical elements.} While a pure text editor is bound to the text pre-
sentation of the document and can therefore more or less only offer
a text view of it, a structure editor may add arbitrary graphical ele-
ments to its presentation to clarify parts of the document. In effect it
will be possible to separate representation of the document from it’s
presentation. There could be two or more presentations of the same
document simultaneously.

\textbf{Derived information.} A structure editor can derive information from
the document and present it in the view. One example would be type
information in a source code editor and information about misspelled
words.
2.2. Structure editors

Structural edit operations. A typical text editor offers edit operations on the character level. A structure editor can offer additional edit operations on other levels in the document, like on the sentence level or paragraph level in a text document editor.

The first two advantages discussed above are both concerning the presentation (i.e. the view) of the edited document. The various possibilities of how to present the document is perhaps the biggest advantage of structure editors. Some of these possibilities may be modelled in a pure text editor as well, like syntax highlighting, but the limitations will always be present: there is no possibility to add characters or graphical elements to the view of the document in a pure text editor.

The last advantage concerns the editing of the document and will be the main topic of this thesis.

2.2.2 Problems with structure editors

Structure editors have often been criticised for being clumsy to use [8, 9]. The pure structure editors would not allow the users to edit text in the straight-fashioned way they were used to. Instead they had to explicitly tell the editor what type of construct to insert next, often with some sort of drop down menu with choices of correct constructs given their current position in the document. This kind of interaction disrupts the user’s work flow, and editing may in some cases take longer than if the user was allowed to simply write down what she wants to express in a left to right manner as in an ordinary text editor. One example would be the simple arithmetic expression “a + 3−(-0.3*pi)”. To express this, the user of a pure structure editor would have to consult the pull-down menu at least four times; once for each operator in the expression.

The above mentioned problem leads us to a certain class of structure editors often referred to as hybrid editors. They are supposed to solve the problem by offering both textual as well as structural editing. The idea is that some constructs (like arithmetic expressions for example) may be edited in a textual fashion and then get their structure by parsing. A famous example of this kind of editor is The Synthesizer Generator [6].
2.2.3 Classes of structure editors

During the course of structure editor development, different ways of thinking about structure editing have emerged. Especially due to the discoveries of the problems above. Most structure editors built belongs to one of the following three classes of structure editors:

- Syntax directed
- Syntax recognizing
- Hybrid

The syntax directed editors are constructed around the concept of top-down editing. The editor continuously controls what the user can insert in the document tree by enforcing syntactic correct additions only.

Since syntax directed editors offered bad ergonomy to the user, syntax recognizing editors, such as Pan [10] and GSE [11], were built to overcome those problems. They are built from a totally different perspective than the syntax directed editors. Instead of manipulating the document structure, the user will now manipulate the textual representation from which the structure is generated by means of parsing. Unfortunately some of the advantages about structure editing is lost this way, since there can be no additional elements in the view except ones that can be parsed into the structure.

The hybrid editors are a compromise between the other two editor classes. They are typically structure editors, but offer syntax recognizing functionality for some parts of the document as explained in the previous section. Efforts have even been made to join existing text editors with structure editors [12].
Chapter 3

Formalising edit operations

This chapter forms the core of this thesis. It explains the formal model which will be used to specify editing operations along with the requirements put on it.

3.1 Requirements

In order to avoid disrupting the users work flow, the formalisation method described in this thesis should make it possible to specify editors that resembles ordinary text oriented editors as much as possible, but yet provide structure edit operations. Users should be able to edit text in a left to right manner instead of expanding placeholders by means of special commands, such as menu selections. Still, we want the structure of the document to be known by the editor, so a pure syntax-recognizing editor would not do. Given this, it is apparent that the formalisation model will have to define hybrid editors, where the boundary between structure editing and text editing is invisible to the user.
While pure structure editors are not able to deal with structural errors, editors specified by this model must be able to do so. Existing hybrid editors are usually able to cope with errors, but only at the lowest levels in the document tree. That means that any erroneous structure is simply represented by a sequence of characters, so that no structure can be located within another structure containing errors. But a good structure editor should be able keep structures intact, even if they happen to appear inside of a broken structure.

As an example, consider the following Pascal like code snippet:

```pascal
begin
  call P(3,(x+1));
  return 0;
end
```

Figure 3.1 shows a possible document tree for the code.

Suppose we happen to make some mistake, making the block structure syntactically illegal. Then the call statement on the second line would still be correct and it would be a pity if we did not have access to the structure editing facilities if we wanted to edit the call statement without touching the block structure. The new, erroneous, version of the code is shown here:

```pascal
begin
```
3.2 Overview

The aim of this work is to develop a formal way to specify how the structure of a tree based document changes during user editing. This will be accomplished by the use of the following components:

- A tree based document representation (document tree)
- Events
- Edit rules
- Edit functions

Each component will be detailed in this chapter. The idea with the model introduced in this thesis is that all user interactions generates events, which may be catched by edit rules. The edit rules specifies how the document tree changes for certain events. The actual changes will mainly be achieved by applying edit functions on the different parts of the document tree.

3.3 The document tree

As noted earlier, most text oriented documents can be represented with a tree structure. The formalisation model used in this thesis will actually
require the documents, that should be edited by the formalised editors, to be represenentable by tree structures.

A tree representing a document will consist of a root element representing the whole document, with sub-trees representing it’s sub-parts. The leaves will represent the symbols in the document, like for example letters, digits and punctuations. They will be referred to as atoms in this thesis. The nodes (except the leaves) will be referred to as structures from now on. Figure 3.2 shows an example of a document tree.

Throughout this thesis there will be statements like “a document has some structure”, which really means that the document is represented by the structure.

### 3.3.1 Up and down in the tree

As a convention, this thesis will use the notion of upwards in the tree to mean moving towards the root element, which then becomes the upmost element of the tree. Likewise, downwards will refer to moving towards the leaves of the tree, so the leaves form the downmost elements of the tree.
3.3. The document tree

3.3.2 Focus

In every interactive editor, there is some way to focus on some structure or object to relay edit operations to that object. Text oriented editors as well as word processors typically realise this by use of a carret or cursor which can be moved between characters in the document.

In a tree structure of a document, the notion of moving between characters is not well defined like it is in an ordinary, character based editor. As long as the cursor stays between two symbols, everything is fine. But when we get to an edge of a structure, some questions arise. In figure 3.3 there are two structurally identical trees. When drawing the figure it was decided that when the cursor sits in the end of a substructure, then the parent will see the cursor as if it was located just before or after that substructure. In the figure this is shown by the dotted lines going through the grayed cursor positions. In the left tree, the cursor sits after the last leave in the left sub-tree $T$. In the right tree, the cursor sits before the first leave in the right sub-tree $U$. These cursor positions are clearly different, but the parent node $S$ cannot tell if the cursor is within $T$ or $U$. And what will happen when the cursor is not in an end position of a node?

Instead of using a cursor and move it between elements (as would be the case in a text editor), we use the notion of focus instead. Any symbol in the tree can have focus, but only one at a time. Figure 3.4 shows the same two trees as above, but with focused symbols instead of using a cursor.

Figure 3.3: The problem with cursor between nodes.
3.3.3 Simulating a cursor with focused symbols

The notion of focused symbols is a very natural and simple way to handle focus in a tree structure. But there is a subtlety to this: Consider the left tree in figure 3.4 were the symbols represent letters. How will this be presented on screen? Most naturally like depicted in this figure:

```
  a b c d
```

Here, letter b has focus, so it is presented with a mark behind it. This is a standard presentation for console applications, but many users may feel uncomfortable with this presentation since many text editors now use a thin cursor drawn *between* the characters instead. But in the tree representation there is no such thing as *between* elements in the tree. However, the use of a thin cursor can still be represented simply by stating that a marked character (symbol) should be presented by drawing a thin cursor *before* the focused character. So the situation above would be presented like this:

```
  a b c d
```
This may be more familiar to most users.

3.3.4 The invisible $ symbol

The biggest problem with cursor modelling using focused symbols instead of positions between them is there will always be one cursor position missing in the end. In practice this will have two implications:

- There will be no way to move behind the last symbol of a structure.
- If the document is empty, e.g. has no symbols, then there will be no symbol to focus and hence no cursor will be shown.

As an example, look at the left subtree of the left tree in figure 3.4. The last character is focused and will look like this on screen:

```
   a|b   c   d
```

What if we want to insert a character after the b? This is not doable since we cannot move behind the b character. Also, if we delete the b character we cannot longer focus it (since it is gone), so we would have to focus on a instead, but this would move the cursor in front of a, which would probably feel awkward. No text editor would do that.

In order to overcome those problems, we introduce a special end symbol, denoted $. This end symbol will always be present in an empty structure so that there will always be a symbol to focus to give us a cursor. In our example we would have the end symbol in the end of the structure all the time. Figure 3.5 shows how the end symbol is focused to make it possible to put the cursor after the b character. Most likely, the end symbol will not be presented on the screen, so it is not shown in the “On screen presentation” part of the figure.
3.4 Structures

The structures constitute the nodes in the document tree and represent different parts of the document, such as assignment statements in a source code document or paragraphs in a text document.

A structure is a tuple \((E, C, N)\) where \(E\) is the editor associated with this structure, \(C\) is the children and \(N\) it’s parent structure\(^1\).

For convenience, structures will be written in a rather compact notation. Consider the structure \((\text{Foo}, C, p)\) where \(\text{Foo}\) is a particular editor, \(C\) the children and \(p\) the parent. This structure would be written like “\(\text{Foo} [c_1 \ c_2 \ldots \ c_n]\)” instead, where \([c_1 \ c_2 \ldots \ c_n]\) is the sequence of children. Since this structure is associated with the \(\text{Foo}\) editor it would be called a \(\text{Foo}\) structure in this thesis.

The parent is not visible in the notation, so I’ll introduce the function \(P(S)\) to return the parent structure of \(S\).

\(^1\)In chapter 5 the structures will be expanded with additional elements.
3.5 Editors

An editor will in this thesis refer to an entity that captures events and in respond to those modifies structures. There will be different types of editors — each handling the editing of it’s own type of structure. In practice, this means that there will be a specific editor for each part of the document, e.g. one editor for assignment statements in source code documents and another editor for paragraphs in text documents etc. Hence a fully fleshed editor for a whole type of document will be composed of several small editors\(^2\). An editor for if-statements in a programming language would be an If editor. The structures it operates on would be associated with this editor and hence typically be called If structures.

Formally an editor can be described as a tuple \((R, F)\) where \(R\) is a set of edit rules and \(F\) is a set of edit functions. These rules and functions dictates the behaviour of the editor. They will be described in section 3.8.

Each editor may be responsible for the editing of several structures. In a source code document, for example, there may be an editor for if-statements

\(^2\)In [2] such entities are referred to as micro editors. I could have chosen the same notation in this work, but choose to call them just editors in order to keep it simple.
and this editor will then be associated with each if-structure in the document. Figure 3.6 shows a document tree being edited by three editors, where some structures refer to the same editor.

The structures are identified by their associations with the editors, so a structure may change editor during an edit session. This means that an if-statement could be changed into a while-statement, for example, by changing it’s editor reference to refer to a while-editor instead of an if-editor.

3.6 Modelling focus

In section 3.3.2 the concept of focus was explained. Any atom can be focused, but to realise this there must be some way to remember what atom currently has the focus. This can be done in different ways. One way would be to let each structure keep a reference to the subtree in which the focused atom is. The problem with this is that finding the current focused atom will require a traversal in the document tree, starting at the root going to the focused atom. This path may be rather long since the document tree could have a significant depth. Since events triggered by edit operations should be directed to the downmost structure (with focused items) first, the traversal may slow down operation of the editor significantly. Actually, this has not been explained yet — it will though in the next section.

Another solution would be to keep a global reference to the focused item. This method has two advantages; there will be no need to traverse the document tree to find the focused item, and it will be very simple to set the reference to refer to another item. But this also has some disturbing consequences. Each structure and atom will now have to be considered objects which in the case of the structures are subject to change. So, there must be a way to refer to those objects from several places at the same time. Because of this, it will be difficult to describe the model in a purely functional way, since it depends on side effects.

Despite it’s problems, this solution has been chosen for this model. The global reference to the focused item will be denoted $M$ from now on. This can refer to any atom in the document tree. So, if an atom $x$ is to be
3.7 Events

Events will typically be triggered by the user pressing a key or performing mouse gestures in order to edit the document. The events can be seen as messages that are sent to the document tree. In this thesis they will be referred to using a sans-serif font, like “erase” for example. Events may carry parameters which will be written after the event name within parenthesis, like “\texttt{ins(c)}”.

When an event is triggered, it is first sent to the bottom-most structure with focus. If that structure does not catch the event, it will be sent further to the structure’s parent structure and so on, until the event is either caught or reaches the root element in which case it is discarded. A structure is said to catch an event if it’s associated editor has an edit rule that catches the event. The structure to which an event has been sent will be referred to as the current structure.

The edit rules are explained in the next section.

3.8 Edit operation rules

The edit operation rules, or just edit rules for short, are used for catching events. An edit rule consist of three parts:

- An event name which is the name of the event triggering the rule.
- A pattern that is matched to the current structure.
- An action to specify what will happen to the structure and possibly other side effects.
An edit rule will catch an event if the current structure matches the edit rule’s pattern. When that happens, the actions of the edit rule will be performed, typically in order to alter the current structure.

As an example of an edit rule, take a look at the following rule from the rather simple Lex editor:

\[
\text{right} \quad \text{Lex}[\alpha @x y \beta] \rightarrow \text{Lex}[\alpha \ x \ y \ \beta] \ M = y
\]

This example demonstrates how the edit rules can be used to move the focus. The rule has event name \text{right}, the part between the event name and the arrow is the pattern and the part after the arrow is the updated \text{Lex}-structure followed by an action. Later the parts will be explained in detail, but for now we will briefly explain what is going on in the rule above. The rule can be read as follows:

Whenever a \text{right} event is sent to a \text{Lex}-structure with at least two subsequent symbols \(x\) and \(y\) somewhere in the structure and focus is on \(x\), then the structure will remain unchanged, but focus will move to the \(y\) element.

In order to exemplify it’s use, we need an example of a document to edit. Figure 3.7 shows a \text{Lex}-structure containing the character sequence “may” (followed by the end symbol). Now suppose the user presses the \text{right} key to generate a \text{right} event. The event would first be sent to the structure having focus, in this case the “a”-symbol. Since it is an atom, it won’t have any edit rule, so the event will be sent to the parent structure; the \text{Lex} structure. This structure has a focused symbol (the “a”-symbol) followed by another symbol (the “y”-symbol), so it will match the pattern of the rule and the action will be carried out. The action will let the structure stay unchanged and set focus on “y”-symbol.

### 3.8.1 Edit rule patterns

The idea with patterns in the edit rules is that a rule’s actions will not be executed if the current structure does not match the rule’s pattern. This way one editor can have several rules for the same event — if the first rule for that event does not match then the next rule is tested and so on. If
no matching rule is found for a given event, then the event is sent to the
current structure’s parent as explained earlier.

A pattern is built from the following components:

- $x$ matches any symbol and binds it to $x$.
  The same goes with $y$ and $z$.

- $S$ matches any structure and binds it to $S$.
  The same goes with $T \ldots W$.

- $X$ matches any symbol and any structure and binds it to $X$.
  The same goes with $Y$ and $Z$.

- $\alpha$ matches any sequence of structures and symbols, including
  the empty sequence (denoted $\epsilon$), and binds that sequence to
  $\alpha$.
  The same goes with $\beta \ldots \delta$.

- $S[P]$ matches any structure whose sequence of children matches
  the pattern expression $P$ and binds it’s type to $S$.

Sometimes it is necessary to match some particular symbol or structure.
For symbols this will be done by using it’s name in the pattern written in a
bold font. For example the symbol “a” would be written as “a” in a pattern.
Specific structures will be written like “Foo[P]” in pattern expressions where
$P$ is a pattern expression for the children. If we don’t care about the
children, but only want to match a particular type of structure, we can
just leave out the child sequence in the pattern like this: Foo. When two
or more structures of a specific type must occur in the pattern they will be subscripted like this: \( S[\text{Foo}_1 \text{ Foo}_2] \) meaning that \( \text{Foo}_1 \) and \( \text{Foo}_2 \) are two different Foo-structures.

### Matching focused elements

Most often we want to identify which element in a structure has the focus. Think about the example of the right edit rule for Lex structures as shown earlier:

\[
\text{right } \text{Lex}[\alpha @x y \beta] \rightarrow \text{Lex}[\alpha \ x \ y \ \beta] \ M = y
\]

Here, the @-sign in the pattern is a special pattern expression with the following definition:

- \(@x\) matches \(x\) if \(M = x\).
- \(@S\) matches \(x\) if \(M = x\) as well as \(S[\alpha @X \beta]\)

There are two more focus pattern expressions, \(F\) and \(L\) defined as follows:

- \(FS\) matches \(x\) if \(M = x\) as well as \(S[\mathcal{F}X \beta]\)
- \(LS\) matches \(x\) if \(M = x\) as well as \(S[\alpha \mathcal{L}X]\)

The intuition is that \(FS\) matches any structure whose first symbol has the focus while \(LS\) matches any structure whose last symbol has the focus. They both match a single symbol that has focus.

### 3.8.2 Patterns in rules

In the edit rules, the variables in the pattern will refer to the same structures as those in it’s action. So in this example:

\[
\text{swapev } \text{Lex}[\alpha @x y \beta] \rightarrow \text{Lex}[\alpha \ y \ x \ \beta] \ M = y
\]

Each variable \(\alpha, \beta, x\) and \(y\) refer to the same structures on the right hand side of the arrow as to the left hand side. The values of the variables are their corresponding bindings achieved when matching the document structure to the pattern of the rule. So given the following document structure:
Lex[m a y $], $M = m$

which corresponds to the example given earlier, where $m, a, y$ and $\$$ are symbols representing the corresponding characters and $y$ has focus, we would get the following bindings when matching against the rule above:

$\alpha = \varepsilon, \beta = (y \$$), x = m, y = a$

So the resulting structure given by the right hand side of the rule, would be the following:

Lex[a m y $], M = a$

### 3.8.3 Edit rule actions

An edit rule can be seen as a structure transformation triggered by an event. The rule will transform every structure matching it’s pattern into the structure given by it’s action part. The action part of an edit rule specifies what will happen when the rule is triggered. The action part is a sequence of one or more actions separated by semicolons. The first action must always specify the resulting structure which must be of the same type as the one matched by the pattern.

### 3.9 Common edit operations

The edit operations are defined by the edit rules and each editor may define it’s own set of edit operations. But some edit operations are rather generic in their nature and it would be convenient if they were available for every structure in the document tree.

In order to provide maximum flexibility, the creator of editors should take as a common principle to try making as little assumptions about parent and child structures as possible. An editor should make no assumptions at all about parent structures, since a structure will not know what structure it will be a child of — any structure should be able to use any other structure as a child. Take a Lisp program document as example; a List-structure
could have another List-structure as a parent (lists can be nested inside other lists in Lisp programs) but if the List-structure is quoted it’s parent might be a Quote-structure instead. The Quote-structure may offer very different edit operations than that for List-structure, hence we can make no assumptions about the parent.

The child structures of a certain structure may be known. For an If-structure, for example, we could define an editor with edit operations such that we can ensure that it will always have a certain sequence of children, like [\textbf{IF } \text{Bexp} \text{ THEN } \text{Block} \text{ END}]. But in general it would be bad to make assumptions about the children of the children, since changes in a structure would then perhaps require changes in the editors for other structures (the ones who made assumptions about the changed structure). Also, in chapter 5 the model will be extended to handle error structures which means that we cannot even make assumptions about the children.

The above observations leads to the conclusion that we cannot make any assumptions about the availability of edit operations for child structures at all. This would really be a problem since edit operations often have to be applied to child structures (this will be evident later).

### 3.9.1 Edit functions

To overcome the problem that the availability of edit operations are not known, some standard operations are defined, that will be defined for every structure. These operations will be defined as functions, so that we can use them to construct new versions of structures by applying the functions to child structures. The function arguments are structure patterns, so the same functions can be defined for many different patterns.

The functions shown below are default operations, i.e. they are defined for every structure $S$:

\[
\begin{align*}
\text{first}(x) &= x \\
\text{first}(S[T \beta]) &= \text{first}(T) \\
\text{last}(x) &= x \\
\text{last}(S[\alpha T]) &= \text{last}(T)
\end{align*}
\]
split$(S[α \mathbin{@} x \ β]) = (S[α], S[x \ β])$

split$(S[α \mathbin{@} T \ β]) = (S[α \ L], S[R \ β])$ where $(L, R) = \text{split}(T)$

join$(S[α], S[x \ β]) = S[α \ x \ β]$

join$(S[α \ T], S[U \ β]) = S[α \ V \ β]$ where $V = \text{join}(T, U)$

Note that only split matches focused structures — it uses the focus to define the splitting point. This means that split can only be called in structures with focus.

When making new editors, those functions must always be provided, but each editor may give their own definitions of the functions. This will be exemplified in subsequent chapters.

In addition to the functions above, every structure should have a function that creates a new instance of the structure. In this thesis those functions will be denoted $\text{new}_S$ where $S$ is the type of the structure. The following four functions can be useful for many edit rules:

prepend$(T, S[α]) = S[T \ α]$

dropf$(S[x \ α]) = S[α]$

dropf$(S[T \ α]) = S[\text{dropf}(T) \ α]$

append$(T, S[α]) = S[α \ T]$

dropl$(S[α \ x]) = S[α]$

dropl$(S[α \ T]) = S[α \ \text{dropl}(T)]$

For some structures there is no logical way to describe those functions, so a function $\text{pendable}$? is introduced that returns True for every structure for which the above mentioned functions are defined and False otherwise. So each time one intends to apply one of those functions to a structure, $\text{pendable}$? should be called on the structure first, to see if the operation is possible.

\footnote{Unfortunately, I have not introduced any edit rules in this thesis that actually make use of those functions. However, due to their primitive nature, they are included anyway.}
3.9.2 Edit rules

Since the focus plays such an important role in the editors, the user must be able to move it between the structures in the document tree. Two events will be introduced for this purpose; left and right. These events should be caught by all editors and the following edit rules defines a standard behaviour for any structure $S$:

\[
\text{right } S[\alpha @X Y \beta] \rightarrow \star \ M = first(Y) \\
\text{left } S[\alpha X @Y \beta] \rightarrow \star \ M = last(X)
\]

Notice the use of the star ($\star$) used to depict that the structure remains unchanged by the rule — only the focus is moved. Also note that $X$ and $Y$ can be either symbols or structures, so focus is set to the last and first elements of the respective elements. The definition of those functions given in the previous section, states that $last(x)$ and $first(x)$ for any symbol $x$ is the symbol $x$ itself, while $last(S)$ and $first(S)$ for any structure $S$ is defined recursively as the first and last element of $S$ respectively.

3.10 Complementary edit operations

Sometimes it happens that two edit operations becomes complementary, so that they can undo each others effects. The left and right operations above are complements and likewise, are the split and join functions complements as defined above. If every edit operation had a complement, this could be used for implementing undo/redo functionality — just push every edit operation on a stack and when the user wants to undo, pop the appropriate number of edit operations and apply them again (in reversed order).

Unfortunately, every edit operation does not have a natural complement. Consider an edit rule for inserting an element before the focused element in a structure as defined here:

\[
\text{ins}(X) \quad S[\alpha @Y \beta] \rightarrow S[\alpha X Y \beta]
\]

We can easily define a complementary operation for this rule by removing the element before the focused one, defined as:
3.10. Complementary edit operations

\[ \text{del} \quad S[\alpha X @Y \beta] \rightarrow S[\alpha Y \beta] \]

But the del rule does not have a complementary rule since that would require storing the removed structure somewhere so that it could be restored later. Even though that solution would work in this case, there would be much more complicated situations when attributes and parsing are introduced to the model, as will be done later. The concept of undo/redo is indeed a very interesting one, but it has not been investigated further in this thesis.
Chapter 4

Editor examples

In this chapter I will present some examples of simple editors by giving their specification as edit functions and edit rules.

4.1 The Lex editor

This section will give the specification of a very simple editor, which will be used extensively by other more advanced editors. The purpose of the Lex editor is to edit lexical units. A lexical unit is a sequence of characters, so the structure of a Lex editor has no substructures — only atoms representing characters.

4.1.1 The edit functions

The first step to be taken when using a new editor, is to create it’s structure. The new function will take care of that:

\[ new_{Lex} = \text{Lex}[$\$]$ \]
4.1. The Lex editor

Not a very exciting function — it merely creates a Lex-structure containing nothing but the end-symbol.

The definition for split must be special for our Lex structure. Because every Lex structure must contain the end-symbol, a special split function must be defined to ensure this:

\[
\text{split}(\text{Lex}[\alpha \ @ x \ \beta]) = (\text{Lex}[\alpha \$], \text{Lex}[x \ \beta])
\]

Note the addition of the end-symbol in the left structure of the result. Also note that we don’t need the recursive version since a Lex structure cannot contain structures, but only atoms.

To ensure that we won’t get two end-symbols within the same Lex structure, the definition of join must be overloaded for Lex structures like this:

\[
\text{join}(\text{Lex}[\alpha \$, \text{Lex}[x \ \beta]) = \text{Lex}[\alpha \ x \ \beta]
\]

Here the end-symbol in the structure of the first parameter has been ignored in the result.

The other functions append, prepend etc are standard. They do not have to be overloaded for this structure.

4.1.2 The edit rules

We can move focus to the next or previous character in the structure by means of the following edit rules:

right  \quad \text{Lex}[\alpha \ @ x \ y \ \beta] \quad \rightarrow \quad \star \quad \mathcal{M} = y
left   \quad \text{Lex}[\alpha \ x \ @ y \ \beta] \quad \rightarrow \quad \star \quad \mathcal{M} = x

In order to add characters to the structure, we use an insertion rule:

ins\(c\)  \quad \text{Lex}[\alpha \ @ x \ \beta] \quad \rightarrow \quad \text{Lex}[\alpha \ c x \ \beta]

Note that \(c\) is a character and is inserted before \(x\) that will stay focused after the operation.

erase  \quad \text{Lex}[\alpha \ x \ @ y \ \beta] \quad \rightarrow \quad \text{Lex}[\alpha \ y \ \beta]
The erase rule above is the complement to the ins rule. This rule will undo the effect of the ins rule and vice versa.

Many text editors offer the possibility to erase the character in front of the cursor. This can be achieved with the following rule:

\[
\text{delete } \text{Lex}[α @x y β] \rightarrow \text{Lex}[α y β] \quad M = y
\]

Since we remove the character with the focus, we have to give some other character focus in order to keep the cursor. The succeeding character is a good choice.

Note that the only way to remove characters in the structure is by applying either erase or delete. A look at their rules will reveal that they will always leave the last character untouched — there is no way to remove the last character! This is desirable since the last character is the end symbol and should never be removed.

### 4.2 A Line editor

If you think about it, the Lex editor as exemplified above, really corresponds more or less exactly to an ordinary text editor: it’s structure is just a sequence of characters. We could almost say there is no structure at all in those documents (but even the flat structure is in fact a structure of course). But now it is time to introduce a slightly more interesting editor; one that has a little more advanced structure.

The Line editor will edit a structure consisting of a sequence of words. A word is simply a sequence of characters, but this is also the structure of the Lex editor described above, so now we can use that editor to edit the words. Hence the structure of the Line editor will be a sequence of Lex structures.

#### 4.2.1 The edit functions

Again, we need a function to create a structure for the editor. However, since we are interested in editing words (that is Lex structures), we should
not just create an empty Line structure, but a Line structure containing a new Lex structure. So our \( \text{new} \) function for creating the structure would look like this:

\[
\text{new}_{\text{Line}} = \text{Line}[\text{new}_{\text{Lex}}]
\]

The standard functions are the same as in the standard case, so no need to repeat or overload them here.

### 4.2.2 The edit rules

The role of the Line editor is to handle the creation and modification of a line of text, that is a sequence of words. The words themselves are handled by the Lex editor, so we won’t need to specify how to insert or delete characters — that is taken care of (by the Lex editor). Instead, we should only concern about handling the sequence of Lex structures.

The question is; how do we know when the user is starting to edit a new word? A rather natural way to start a new word would be to press the spacebar, since this would normally be the way to start a new word in a common text editor. So, suppose the spacebar key would generate a space event, then we could add the following edit rule:

\[
\text{space } \text{Line}[\alpha \odot \text{Lex } \beta] \rightarrow \text{Line}[\alpha \ L \ R \ \beta] \quad \mathcal{M} = \text{first}(R) \\
\text{where } (L, R) = \text{split}(\text{Lex})
\]

This may seem a bit awkward; why are we splitting the Lex structure in two? The idea is that the user may have moved the cursor to somewhere in the middle of a word, that is somewhere in the middle of the Lex structure above. Then if the user presses spacebar she would expect the word to be split in two. This will still work if the focus is on the last or the first character of the word. If it is on the last character we would get all characters in the left word and a new word to the right containing nothing but the end-symbol. The case when focus is on the first character is analogous. A look at the split function for the Lex structure will reveal the described behaviour.

Also note that the focus will be moved to the first character in the right word (Lex structure). This might not seem obvious — we could have set
focus on the last character of the left word as well, but try position the
cursor within a word in your favourite text editor and press \texttt{spacebar} and
see where the cursor goes. Most likely it will be placed in front of the right
word and \textit{not} after the left word.

Given the edit rule above we can create new words and splitting existing
ones. However, we cannot remove any words or join them, so we need someedit rule for this. The question to ask here is “what would cause words to
disappear or being smashed together?”. The answer is perhaps obvious;
deletion. So let’s add an \texttt{erase} rule in the spirit of the \texttt{erase} rule for our \texttt{Lex}
structure:

$$
\texttt{erase \ Line[}\alpha \ \texttt{Lex}_1 \ \mathcal{F}\texttt{Lex}_2 \ \beta \ ] \ \rightarrow \ \texttt{Line[}\alpha \ \texttt{join(Lex}_1, \texttt{Lex}_2) \ \beta \ ]
$$

First note the use of subscripts on the \texttt{Lex} structures; they are just in
place to make it possible to unambiguously refer to the different structures.
What we really do is erasing the space between the words. Remember the
definition of \(\mathcal{F}\) and \(\mathcal{L}\) from 3.8.1: \(\mathcal{F}\texttt{Lex}\) matches any \texttt{Lex}-structure whose
\texttt{first} symbol has focus, and \(\mathcal{L}\texttt{Lex}\) matches any \texttt{Lex}-structure whose \texttt{last}
symbol has focus.

Now it would be tempting to add a corresponding \texttt{delete} rule to join two \texttt{Lex}
structures from the left, as has been done in the rule below. But beware;
the rule will have an unwanted side-effect!

$$
\texttt{delete \ Line[}\alpha \ \mathcal{L}\texttt{Lex}_1 \ \texttt{Lex}_2 \ \beta \ ] \ \rightarrow \ \texttt{Line[}\alpha \ \texttt{join(Lex}_1, \texttt{Lex}_2) \ \beta \ ] \ \texttt{Bad rule}!!
$$

It may be tricky to spot the problem here. First think about what atom
will be focused in this rule; it will be the last atom of the left \texttt{Lex}\ structure.
But the last atom of a \texttt{Lex} structure is the end-symbol, and according
to the definition of \texttt{join} for \texttt{Lex} structures, this symbol will be removed!
Hence we will lose the cursor if we do not explicitly set the focus to some
other symbol. The best choice would be the \texttt{first} symbol in the right \texttt{Lex}
structure, so the correct version of the edit rule above would be:

$$
\texttt{delete \ Line[}\alpha \ \mathcal{L}\texttt{Lex}_1 \ \texttt{Lex}_2 \ \beta \ ] \ \rightarrow \ \texttt{Line[}\alpha \ \texttt{join(Lex}_1, \texttt{Lex}_2) \ \beta \ ] \\
\mathcal{M} = \texttt{first(Lex}_2\)
$$

To be safe, we should have done this for the \texttt{erase} rule as well.
As of now, we could use the two editors described above in order to edit a sequence of characters. In the next example I will describe a full-blown text structure editor, then it will become apparent why the editor in this example was called a Line editor.

4.3 A Text editor

A text document can be described as a sequence of text lines which in turn is a sequence of words. It would of course be possible to go even further by introducing headings and sections etc. But then it would explode into several constructs of a full-blown word-processor-document with formatted text environments like bulleted lists, tables, figures and whatnot. A text in this context will refer to an unformatted piece of text, i.e no font changes or the like, only plain text. Paragraphs may be added, but I’ll skip that for now — only lines of text will be used in this example.

As mentioned, the structure of the Text editor will be a sequence of lines. Now it should be apparent that we can reuse our Line editor described above to edit the single lines of the text. Still we need an editor to handle the sequence of Line structures — the Text editor.

4.3.1 The edit functions

Creating a new Text structure would be done in exactly the same way as creating a Line structure:

\[ new_{Text} = Text[new_{Line}] \]

4.3.2 The edit rules

New lines would normally be created by pressing the enter key, so we can add the following edit rule to reflect this:

\[ \text{enter} \quad Text[\alpha \oplus S \beta] \rightarrow Text[\alpha \ L \ R \ \beta] \quad M = first(R) \]

where \((L, R) = split(S)\)
And let’s accompany this with some deletion rules as well:

\[
\text{erase } \text{Text}[^a S_1 F S_2 \beta] \rightarrow \text{Text}[^a \text{join}(S_1, S_2) \beta] \quad M = \text{first}(S_2)
\]

\[
\text{delete } \text{Text}[^a L S_1 S_2 \beta] \rightarrow \text{Text}[^a \text{join}(S_1, S_2) \beta] \quad M = \text{first}(S_2)
\]

Since \text{join} is generally only defined for structures of the same type, we would be in trouble if \( S_1 \) and \( S_2 \) in the rules above would be of different types. However, as long as we don’t let other structures than \text{Line} structures be inserted in the structure, it should not be a problem.

4.4 A comma separated list

The above examples described editors whose structure were simple sequences of some element type. Now it is time to introduce an editor with a little more complicated structure than just a sequence of elements of the same type. The structure for this example is one that is common in many programming languages; a comma separated list inside a pair of parentheses. This structure is for example used for parameter lists in procedure calls, where the items are formal or actual parameters. The type name of the structure will be \text{List}, and the items will simply be \text{Lex} structures.

4.4.1 Edit functions

The \text{List} editor will have a little more advanced structure and the \text{new} function for the list will serve as a template for a the structure by inserting the surrounding parentheses:

\[
\text{new}_{\text{List}} = \text{List}[L P \text{new}_{\text{Lex}} R P]
\]

The \text{LP} and \text{RP} symbols represent left and a right parenthesis respectively. Because we have special symbols at the ends of the structure, we need to redefine some of the edit functions for the \text{List} structure to make sure that we always keep those symbols in the edges and that they won’t reappear on other places inside the structure. We must also make sure that the
separating commas, which will be denoted by COM symbols, will stay between items and not get duplicated or disappear.

\[
\text{split(} \text{List}[\alpha @ S \beta]) = (\text{List}[\alpha L \text{RP}], \text{List}[\text{LP} R \beta])
\]

where \((L, R) = \text{split}(S)\)

\[
\text{join(} \text{List}[\alpha \text{RP}], \text{List}[\text{LP} \beta]) = \text{List}[\alpha \text{COM} \beta]
\]

The definition of \text{split} makes sure that new structures will have the required parenthesis, provided that the original structure is correct. The definition of \text{join} makes sure that there is a separator (COM) between the elements in the new list and that extra parentheses are removed.

Note the asymmetry between \text{split} and \text{join}, they are not able to undo each-other’s operations, since \text{join} is not recursively defined. This will keep us out of trouble if there happens to be different structures in the List structure that cannot be joined.

Also, since there is no point in focus the parentheses we will redefine \text{first} and \text{last} for this structure:

\[
\text{first(} \text{List}[\alpha \text{LP} S \beta]) = \text{first}(S)
\]
\[
\text{last(} \text{List}[\alpha S \text{RP}]) = \text{last}(S)
\]

4.4.2 Edit rules

Like before, the question here is how new list items will be created. The most natural way for the user would be to create new items by typing the item separator. If we assume that our separator COM will be presented as a comma (,) we would have to catch the ins(c) event when c is the comma character. But there is a problem with this; since all events are sent to the children first, the ins event will be captured by the Lex structure and handled there, so this event will never reach our List structure. In order to be able to handle this, we would have to make sure that the Lex structure never catches the event somehow.

The solution is to extend the model a little, by introducing delimiters. The idea with delimiters is that each editor can specify a set of delimiter characters, and events with those characters as parameter will never be
captured by any Lex structure below the structure associated with the editor.

So, in our case we add the comma character to the set of delimiters for our List structure. It would now be possible to define the edit rule:

\[ \text{ins}(d) \quad \text{List}[\alpha @ S \beta] \rightarrow \text{List}[\alpha L \text{COM} R \beta] \]

where \((L, R) = \text{split}(S)\)

Where \(d\) in this case denotes the comma character.

And, for removal of items, we define the following rules:

\[ \text{erase} \quad \text{List}[\alpha S \text{COM} FT \beta] \rightarrow \text{List}[\alpha \text{join}(S, T) \beta] M = \text{first}(T) \]

\[ \text{delete} \quad \text{List}[\alpha LS \text{COM} T \beta] \rightarrow \text{List}[\alpha \text{join}(S, T) \beta] M = \text{first}(T) \]

Again, since we don’t want to focus the parentheses or any separator, we’ll define the following rules:

\[ \text{right} \quad \text{List}[\alpha LS \text{COM} T \beta] \rightarrow \text{List}[\alpha S \text{COM} T \beta] M = \text{first}(T) \]

\[ \text{left} \quad \text{List}[\alpha S \text{COM} FT \beta] \rightarrow \text{List}[\alpha S \text{COM} T \beta] M = \text{last}(S) \]

They will simply skip the parentheses and separators.
Chapter 5

Extensions and difficulties

In this chapter, the model will be extended to deal with practical issues that will arise in structure editors. Some ideas are presented for how to handle comments and selection. Attributes and alternations will be discussed a bit more extensively.

5.1 Lexical editors and subtyping

In chapter 4 the Lex editor was defined and was then used in every other example editor throughout that chapter. The Lex editor is somewhat special in that it’s structure does only contain symbols — no other structures. This property makes it a valuable tool for representing lexical elements in a document and most documents contain a lots of those. In a text document, every word will typically be represented by a Lex structure as demonstrated in the examples. Also, a source code document will contain many lexical elements such as identifiers and keywords.

In a source code document there are several different types of lexical elements; variables, numbers, strings etc. and although they are all lexical
elements they do not necessarily share all properties. An editor for num-
bers would for example be very different than one for identifiers. It would
be nice to be able to type a number lexical as a number and an identifier
lexical as an identifier, but still be able to refer to any lexical structure. All
lexical structure can be said to belong to the same class of structure; the
Lex structure.

To be able to type lexical elements differently and still be able to refer to
general Lex structures (in edit rule patterns for example) we introduce the
concept of subtypes to the formal model. We can define new structures
with their own type and then declare them as subtypes of the Lex type.
The following notation will be used to define a subtype from the Lex type:

\[ \text{Num} < \text{Lex} \]

So, with this declaration any occurrence of Lex in a pattern will not only
match a Lex structure but also a Num structure. Moreover the Num struc-
ture can have it’s own editor functions and rules. The only requirement is
that the Num structure will only contain symbols, just like the Lex struc-
ture. That is the one common property of the Lex class of editors —
anything else may be different.

5.1.1 Example

Just to clarify the use of subtyped Lex editors, I will present two examples of
Lex editors, namely the ones mentioned earlier; one Num editor for editing
numbers and one Ident editor for editing identifiers.

A Num editor

A new Num structure would be created exactly like an ordinary Lex struc-
ture:

\[ \text{new}_{\text{Num}} = \text{Num}\llbracket $\rrbracket \]

The only thing that makes the Num structure special is that it is supposed
to contain only digits, so the only thing we have to redefine for this editor
is the ins edit rule so that only digits are inserted:
\text{ins}(d) \quad \text{Num}[\alpha \ @x \ \beta] \rightarrow \ \text{Num}[\alpha \ d \ x \ \beta] \text{ where } d \text{ is a digit.}

Really simple — everything else may be the same as for the standard \text{Lex} editor.

\textbf{A Ident editor}

We create the structure with the function:

\[
\text{newIdent} = \text{Ident}[$$
\]

Many programming languages define the syntax of identifiers as beginning with an english letter followed by a sequence of letters and possibly digits. Other characters may be allowed as well, but for this example we will go with this structure. The important thing is that the first character be always an english letter while the rest might be any letter or digit. For this we define two \text{ins} rules:

\text{ins}(l) \quad \text{Ident}[\alpha \ @x \ \beta] \rightarrow \ \text{Ident}[\alpha \ l \ x \ \beta] \text{ when } l \text{ is an english letter } (a - z).

\text{ins}(d) \quad \text{Ident}[\alpha \ x \ @y\beta] \rightarrow \ \text{Ident}[\alpha \ x \ d \ y \ \beta] \text{ when } d \text{ is a digit.}

The first rule will let you insert a letter anywhere in the structure, while the second rule, for insertion of digits, is only applicable when there is at least one other character in front of it. As for the \text{Num} editor, rules for deletion will stay the same as for the standard \text{Lex} editor.

There is one little problem with this editor. Although it won’t be possible to insert anything else but a letter in the first position, a digit can actually appear there anyway! This will happen when there is only one letter at the beginning of the structure and this is removed. For example, the structure \text{Ident}[a \ @2 \ c] will after an \text{erase} event look like \text{Ident}[@2 \ c]. Clearly this structure is not a correct \text{Ident} structure, since it starts with a digit. This will also happen if we split an \text{Ident} structure with focus on a digit. Consider a (correct) \text{Ident} structure matching the following pattern:

\text{Ident}[\alpha \ @x \ \beta] \text{ where } x \text{ is a digit}

If we apply that structure to \text{split} we would get the following structures:

\text{Ident}[@] \text{ and } \text{Ident}[x \ \beta]
The left structure would be correct (provided that the original structure was correct), but the right structure will now start with a digit and hence not be a legal \texttt{Ident} structure, although it has been typed as one.

In section 5.5 this problem is generalised and a possible solution is presented.

## 5.2 Attributes

One of the main strengths of structure editing is that the different structures in the document can have extra data associated. In a text document, for example, we may be able to set font, size and color attributes for different parts of the text. This information belongs to the edited document and must be stored therein, that is if we store the document on file and open it some other time, we want this information to be restored from the file.

It is important to differ between \textit{implicit data} and \textit{explicit data}. Implicit data is common for every structure of a certain type and will not be stored within the structures. Explicit data on the other hand, may be unique for every instance of a certain structure type and will therefore be stored within the structures.

To exemplify the difference between implicit and explicit data, consider a source code editor with a \texttt{Keyword} structure for keywords. This editor might present all keywords in a bold font by rendering all characters in the \texttt{Keyword} structures in bold. This behaviour lies within the presentation and hence do not have to be explicit in the document, i.e we don’t have to store this information within each \texttt{Keyword} structure. This is an example of implicit data.

In a text editor on the other hand, we will edit words which may be drawn in different fonts and colors. Since this information cannot be derived from the structure (we cannot just look at a \texttt{Word} structure and tell what font it should have) it must be stored within the structure itself, so that it can be restored in another edit session. Hence information about fonts and colors in this case, would constitute explicit data.
5.2. Attributes

Only explicit data will be considered in this section. Implicit data is not really of interest for this thesis, since it does not have anything to do with the structure of the document — only with the presentation.

Now consider the Text editor presented in chapter 4. It can be used for editing simple texts but it cannot handle font or color changes. Suppose we want an editor which can handle this. One way to do this would be by adding new structures; Bold, Italic, Red etc. and then let them hold lexicals (words). While this works well for bold and italic settings, it would require one new structure for each possible color, which would yield arbitrarily many structures. A nice solution to this problem would be to use one single Color structure and have that hold an attribute to determine the color.

In order to do this extend the model so that each structure also has a set of attributes. This set of attributes will be denoted as a comma separated collection of attributes enclosed in curly brackets, like this:

\[
\{\text{attrib}_1 = \text{val}_1, \text{attrib}_2 = \text{val}_2, \ldots, \text{attrib}_n = \text{val}_n\}
\]

Each attribute will have an associated value, which can be expressed using the "attribute = value" notation above.

A structure with an associated attribute set will be denoted like this:

\[
S[\ldots]\{\text{attrib}_1 = \text{val}_1, \text{attrib}_2 = \text{val}_2, \ldots, \text{attrib}_n = \text{val}_n\}
\]

Now, let’s introduce “ring” (∘) as a union operator on attribute sets to make it possible to add new attributes to structures. So the following relation holds:

\[
S[\ldots]\{a = x, b = y\} \circ \{a = u, c = z\} = S[\ldots]\{a = u, b = y, c = z\}
\]

1The ring operator is actually a function \((S, A) \rightarrow S\) where \(A\) is an attribute set and \(S\) a structure (possibly with attributes). It will be used as an operator in this thesis for convenience.
So, the right attribute set will overload any attributes in the left set.

A structure can be queried for an attribute using a dot-notation. So if we want to know the value of attribute \( a \) for \( S \), then we write \( S.a \).

There is also an operator to remove attributes from structures. This will be denoted \( / \) and can be used as follows:

\[
S[\cdots]/a
\]

This expression means that attribute \( a \) is removed from \( S \). Removing an attribute from a structure which does not have the attribute has no effect.

Attribute queries are forwarded to parents if necessary, so the following relation holds:

\[
(S[\cdots]f\{a = x\}/a).a = \mathcal{P}(S).a
\]

(Remember that \( \mathcal{P}(S) \) is the parent of \( S \).)

The equation means that if we set the attribute \( a \) to \( x \) in \( S \) and removes it, then if we query \( S \) for \( a \) we will get the value of \( a \) for the parent of \( S \) (i.e.\( \mathcal{P}(S) \)).

In order to extract the whole attribute set from a structure, define the function \( \mathcal{A} : S \to A \) where \( S \) denotes structures and \( A \) denotes attribute sets. Then the following holds:

\[
\mathcal{A}(S[\cdots]\{a = x, \ b = y\}) = \{a, \ b\}
\]

### 5.2.1 Using attributes in edit rules

Attributes will be manipulated in the edit rules. A structure can be given additional attributes in an edit rule action. Attributes can also be
used in patterns in order to make it possible to match structures with the given attributes. However, when used in patterns, any attribute set will not necessarily denote all attributes in the structure. The pattern $S[a] \{a = x, b = y\}$ will match any structure with attributes $a$ and $b$ — even if it contains other attributes as well. So, for example, would it match $\text{Foo}\{\cdots\}\{a = \cdots, b = \cdots, c = \cdots\}$ despite that $\text{Foo}$ has an extra attribute $c$.

The following edit rule exemplifies the use of attributes in the pattern as well as in the action part.

$$f,\text{oom} \quad \text{Ost}\{\alpha\text{T}\{a = x\}\beta\} \rightarrow \text{Ost}\{\alpha\text{T}\{a\beta\}a\{b = x\}\}$$

This (admittedly silly) example will be caught by any $\text{Ost}$ structure that contains a $\text{T}$ structure with an attribute $a$. If the document matches this structure, $x$ is bound to the value of $a$ and in the action we remove $a$ from the $\text{T}$ structure while adding a $b$ attribute to the $\text{Ost}$ structure with the value bound to $x$. In effect this rule will copy the value of $\text{T}.a$ to $\text{Ost}.b$ while removing $a$ from $\text{T}$.

Section 5.2.3 provides a more serious example of the use of attributes.

### 5.2.2 Considerations

It may appear to be simple to extend the model with structure attributes. But there are several issues to consider. The hot question is: What should happen if a structure is queried for an attribute it doesn’t have? There are three alternative choices.

- Provide some kind of nil value.
- Error — it should not happen.
- Forward the query to the parent.

Each possibility has it’s own advantages and disadvantages. If one uses the error solution, one has to make sure that no structures will ever be
queried for attributes they don’t have. On the other hand there will be no surprises, since all attributes have to be explicitly defined for each structure. However, I think this is the worst solution of the three since it requires full knowledge of all possible parent/child combinations in order to avoid errors.

The nil value solution is more stable since it will always work (but the behaviour may not be as intended). It is easy to tell if a structure has a certain attribute; just query the structure and see if it returns nil. The disadvantage with this approach is that it will hide attributes from parents. Consider the case when a structure S with parent P is queried for attribute a. Suppose that we want S.a to return the value of P.a instead of providing it’s own value. This can not be achieved with this method, since S will return nil if it does not have it’s own a. It won’t work if S is given attribute a with the same value as P.a either, because as soon as the value of P.a changes, it will not be same as S.a anymore.\footnote{Actually, this would work if references were introduced into the model. Then we could make S.a refer to the value of P.a. Moreover, we could make S.a refer to any structure’s attribute — not only the P’s. Although this would be the most flexible and powerful model, references are a bit tricky to describe formally, so they won’t be used in this model.}

The last method suggests that if S is queried for an attribute it doesn’t have, say a, then it’s parent will (automatically) be queried for a instead. This method makes it possible for a structure to return the value it’s parent’s attributes by not having those attributes itself. Combined with the possibility for a structure to remove an attribute it already has, this method should provide enough flexibility. On the other hand, there is no way to tell whether a structure does have a certain attribute by just query for it and watch the result. However, this functionality could be provided with a hasAttrib(a) function if needed.\footnote{This is better than using references, since it does not require the editor writer to explicitly make references to parent attributes. That is a good thing to avoid since the model assume that any structure generally does not know anything about it’s parent and hence should not make any assumptions about parent attributes.}

Actually, the forward method described above is not really a complete solution. It’s recursive nature will force us to decide what will happen if the root structure is queried for an attribute it does not have. Since the root structure has no parent, it is not possible to forward the query,
so again the decision has to be made between the Nil value and an error condition (but only for the root structure). As mentioned above, the Nil value method is more stable since it will always return a well-defined value — no matter what happens. This method will be used in this model, but since every possible attribute can (in theory) get a Nil value, it will be up to the editor writer to make an interpretation of the Nil values for every attribute.

### 5.2.3 An FText editor

In order to create a text editor which allows font and color changes in the text, we will use attributes to handle information about fonts and colors. Let’s create a Word structure to contain each word and associated font, size and color attributes. Two more structures will be used to represent the text; Par for paragraphs and FText for the whole text. The relation between the structures can be described in EBNF-like notation as below:

\[
\text{FText} = \text{Par}^* \\
\text{Par} = \text{Word}^*
\]

The FText structure should contain default attributes for the whole text, so its new function will look like this:

\[
\text{new}_{\text{FText}} = \text{FText}[\text{new}_{\text{Par}}] \circ \{\text{font} = \cdots, \text{size} = \cdots, \text{color} = \cdots\}
\]

The values are not interesting for now, that’s why they are written as (⋯). They could be set according to user preferences or be hard coded in the implementation of the FText editor. The edit rules and functions for this editor are analogous to the ones for the Line editor (see section 4.2), so they are printed here without explanation.

\[
\begin{align*}
\text{enter} & \quad \text{FText}[\alpha @\text{Par } \beta] \rightarrow \text{FText}[\alpha L R \beta] \quad M = \text{first}(R) \\
& \quad \text{where } (L, R) = \text{split}(\text{Par}) \\
\text{erase} & \quad \text{FText}[\alpha \text{Par}_1 \not= \text{Par}_2 \beta] \rightarrow \text{FText}[\alpha \text{join}(\text{Par}_1, \text{Par}_2) \beta] \\
\text{delete} & \quad \text{FText}[\alpha \not= \text{Par}_1 \text{Par}_2 \beta] \rightarrow \text{FText}[\alpha \text{join}(\text{Par}_1, \text{Par}_2) \beta]
\end{align*}
\]
The join and split functions over Par structures have their standard definitions, so no need to repeat them here.

The Par structure does not need any attributes, so it will be created without any. It’s edit rules and functions are almost the same as the ones for the FText structure.

\[
\text{new}_{\text{Par}} = \text{Par}[\text{new}_{\text{Word}}]
\]

\[
\text{space } \text{Par}[\alpha \@\text{Word }\beta] \rightarrow \text{Par}[\alpha \ L \ R \ \beta] \quad \mathcal{M} = \text{first}(R)
\]

where \((L, R) = \text{split}(\text{Word})\)

The erase and delete rules are exactly the same as for FText with Word substituted for Par, so they will not be repeated here.

It is the Word structure that is the most interesting structure in this example. It will have no attributes from the beginning, but if the user wishes to set i.e. the size of a word, then that word’s Word structure will get it’s own size attribute. Let’s introduce the following events:

setFont\((f)\), setCol\((c)\)

to set font and color attributes respectively.

The Word structure will be very much like a Lex structure, so I won’t repeat the simple edit rules or the edit functions for it. However, the rules for handling the new events are presented here:

\[
\text{setFont}(f) \quad \text{Word}[\alpha] \rightarrow \text{Word}[\alpha] \circ \{\text{font} = f\}
\]

\[
\text{setCol}(c) \quad \text{Word}[\alpha] \rightarrow \text{Word}[\alpha] \circ \{\text{col} = c\}
\]

An interesting issue to consider is what will happen with the attributes when we join two Word structures. The structures may have different values for the same attribute or one structure could have an attribute which the other doesn’t have. The join function will define the attributes of the resulting structure. In this example we will let the right Word structure define the value of common attributes. The resulting structure will get all attributes from both the left and the right Word structures:

\[
\text{join}(\text{Word}_1[\alpha \ $], \text{Word}_2[\beta \ $]) = \text{Word}[\alpha \ \beta \ $] \circ A(\text{Word}_1) \circ A(\text{Word}_2)
\]
Note that since $\text{Word}_2$ is to the right of $\text{Word}_1$ in the attribute merge any attributes that are in both structures will get the values from $\text{Word}_2$.

When splitting structures there is generally no need to think about what attributes should be used — just using all attributes from the splitted structure appear to be reasonable:

$$\text{split}(\text{Word}_1[\alpha \circ x \beta]) = (\text{Word}[\alpha \emptyset] \circ A, \text{Word}[\circ x \beta] \circ A)$$

where $A = \mathcal{A}(\text{Word}_1)$

To make it possible to revert back to “standard” color (as defined by the FText structure), add an unsetCol event for this. The rule for this event uses the $/-$-operator to remove the col attribute from the structure:

$$\text{unsetCol}(c) \text{ Word}[\alpha] \rightarrow \text{Word}[\alpha]/\text{col}$$

The same can of course be done to unset the font attribute. Font size attributes can be handled in the same way.

## 5.3 Comments

Many different kinds of document formats offer the possibility to insert comments into the document. Comments are most common in source code documents but may appear in other documents as well\textsuperscript{4}. Unfortunately, since they may appear almost anywhere in the document, they are generally hard to cope with in structure editors.

The most natural way to incorporate comments in the formal model would be to provide a special comment-structure. However, it would then be difficult to write structure patterns for structures when comments can appear anywhere. A possible solution to that would be to restrict comments to certain places only, like for example as the last element in structures.

Most often, comments are used for commenting some part in the document. Since parts are represented as structures in the document tree, one may recognize that the user typically wants to comment different structures.

---

\textsuperscript{4}In fact, one may think that every document should provide the ability to insert comments.
Extensions and difficulties

Figure 5.1: A document tree where the gray areas depicts the components in a selected area.

Given this observation one may take another approach to modelling comments — we can extend the model to include a special comment field in each structure. This field may now be used for holding a comment which is associated with the structure. Since the comment will now be separated from the structure of the document, it will not get in the way during editing and can be presented as “floating” above the document in the presentation.

5.4 Selection

Most text oriented editors give the user the possibility to select sequences of characters in the document. Selected text can then be cut or copied into a clipboard buffer from where it can be pasted into the document again, possibly at some other position in the text. Many structure editors offer selection functionality too, but usually in a more restricted manner. Text selection, or rather structure selection, has not been introduced in the formalism proposed in this thesis, but for future addition, the associated problems will be discussed in this section.

Support for selection of arbitrary character sequences is hard to achieve in a structure editor. Consider the document tree depicted in figure 5.1 where the A, B and C structures are representing Lex-structures. If the user was allowed to select any subsequent characters, she might produce the
selection over several Lex-structures as shown in the figure. The selection starts somewhere in the A structure and ends somewhere in the C structure, selecting all characters in between, including the whole B structure (the gray area in the figure).

With the selected area in the figure, if the user issues a cut operation, what should be put into the clipboard? Since the cut spans over several structures (A, B and C) we would have put all those structures, or rather the whole C structure and the cut halves of A and B into the clipboard buffer. Now, suppose the user wants to paste the cut area back into the document again, in the same place as before. This should, logically be possible, since it would be in any ordinary editor. However, it won’t be possible here, since we cannot know where to put the cut structures — they came from different places.

A natural solution would be to restrict how the selections can be made. We can limit the selection to either whole subtrees or parts of Lex-structures. In that case, we are guaranteed to get either a sequence of characters (which can be pasted into any Lex-structure) or a whole subtree (with a single root structure). This can be enforced by automatically enlarging the selection to cover whole subtrees if necessary. This would make the selection made in figure 5.1 to also include structure S and T as depicted in figure 5.2.
5.5 Alternation

What would life be without choice? Really simple but probably also very boring. Choice is what makes things interesting, but it is also what makes things complicated — it isn’t always obvious what alternative to choose. Unfortunately this also holds for documents and in particular for source code documents. The editor examples in chapter 4 are simple much because their structures are simple. And this simplicity is in turn caused by the very limited alternatives given for the structure of the documents edited. It is always clear in those examples what kind of structure will be edited next. However, this is most often not the case. Many documents may have structures with alternating substructures — some positions in the document may be populated by structures which can be chosen from a set of different structures. This is especially true for programming languages whose structures typically require rather complicated grammars with several (and often nested) alternates to explain.

The approach taken in several structure editors to overcome this problem is simply not to let the user enter anything before she has explicitly stated what structure she wishes to insert and edit. This top-down approach has already been mentioned as clumsy and would violate the requirement of as free editing as possible using a bottom-up approach. Another solution would be to let the user edit the presentation (as in a plain text editor) and then update the document tree to reflect the changes. This method has been developed in the Proxima editor [7].

5.5.1 The simplest case of alternation

One simple case of alternation would be when every alternative is a lexical unit. This means they could be represented by Lex structures with the model presented in this thesis. Consider a sequence document structure were each item can be either a number or an identifier which can be represented by the respective Num and Ident structures introduced earlier. Since we don’t know if the currently edited structure will be a Num or an Ident structure until the user has finished editing it, we cannot tell whether to
create a new `Num` or a new `Ident` structure each time the user will begin editing a new item in the sequence. A simple, yet powerful solution in this case would be to always create “neutral” new `Lex` structure each time the user starts editing a new item. Then as soon as the user has finished editing the current item and moves to the next, we can decide if the item is a `Num` or a `Ident` structure and type it accordingly. If the structure doesn’t match any of the two, we can let it remain as an erroneous `Lex` structure and let the user deal with this in the future.

In order to decide what type the structure should have, extend the model to include regular expressions that describes the different `Lex` structures:

```
Num = /\[0-9\]+/  
Ident = /\[a-zA-Z\]\[a-zA-Z0-9\]*/
```

Any `Lex` structure can now be tested for a match against the regular expressions and be typed with the correct subtype. This function will be called `getlex` and it’s signature would be:

```
getlex :: Lex → Lex
```

### Example of the use of `getlex`

The use of `getlex` is exemplified in the following `Seq` editor, which is just a sequence of `Num` and `Ident` structures:

```
space Seq[α @Lex β] → Seq[α getlex(L) getlex(R) β]
where (L, R) = split(Lex)
```

New `Seq` structures are created with the following function:

```
new_{Seq} = Seq[new_{Lex}]
```

If a `Seq` structure get split we should make sure that it’s elements are `getlex:ed` as needed by defining `split` for this structure as follows:

```
split(Seq[α @S β]) = (Seq[α getlex(L)], Seq[getlex(R) β])
where (L, R) = split(S)
```
5.5.2 The general case of alternation

In general, other structures than lexical ones can be alternating. Consider the following description of a structure for some statements in a C-like programming language:

\[
\text{Stmt} = \text{If} \mid \text{While} \mid \ldots \\
\text{If} = \text{IF} \ Bexp \ \text{Block} \\
\text{While} = \text{WHILE} \ Bexp \ \text{Block}
\]

A statement can be many things, like an if-statement or a while-statement as indicated in the description above. The statements are represented by the \text{Stmt} structure which is an alternate structure. As we will see in this section, this kind of general alternation causes several difficulties.

The first problem to consider is what structure will be created when a creating a new \text{Stmt} structure. If \text{Stmt} is really an actual structure in the document tree, it should have one child, either \text{If} or \text{While} or something else as indicated by the dots in the description above. However, it is impossible to tell which of the statement structures the user is intending to edit. Maybe the user intends to write an if-statement or maybe she intends a while-statement.

One possible solution would be to use some kind of default structure, which can be created instead of \text{If} or \text{While} letting the user edit text in that. Then we could let the editor check the structure each time it is changed and match it against the alternates. As soon as a match has been discovered, the structure can be tagged with the right type (i.e. associated with the correct editor).

The solution is simple but has a flaw. The user might want to change an already tagged structure into some other structure by editing. While doing this, the original structure will generally break until it can be recognized as the intended structure. In the description above, consider changing an if-statement to a while-statement. Since they are structurally very alike, it would be natural for the user to erase the \text{If} symbol and replace it with a new \text{While} symbol in order to change the if-statement to a while-statement. But
as soon as the \text{If} symbol is removed, the \text{If}-structure is broken. In general the problem can be depicted like this:

$$S \rightarrow E \rightarrow T$$

Where $S$ is the source structure which the user wants to edit into the $T$ structure. While doing so, the structure will (most probably) pass through a state when it does not match any type of structure in the document, depicted $E$ above. Because of this problem, it is evident that some means of dealing with errors is needed.

### 5.5.3 Dealing with errors

If structures in the document tree can break, some method is needed to ensure that broken structures can be recognized as document structures again, after some edit operations. Since the user must be able to edit broken structures as well as others, edit rules are needed for them as well.

The solution would be to introduce a default editor, called $D$, and define it’s edit rules and functions as the ones in section 3.9. Now, as soon as some structure breaks, it will be transformed into a $D$ structure. Since the $D$ structures must be recognized as other structures if there is a match, a parser is needed. The idea is that every time a \text{Lex} structure within a subtree of a $D$ structure is changed, the structure will parsed.

Using this model, transforming a structure $S$ into another structure $T$ would yield a transformation $S \rightarrow D \rightarrow T$.

**Parser requirements**

Since parsing is a large and well researched subject in itself, I will not go into any parsing technique in this thesis (see [13, 14] for references of parsers for structure editors and incremental parsers). However, some requirements must be put on the parser:
• It must preserve the $D$ tree if the parsing fails.
• If the parsing is successful, every symbol must be preserved.

The first requirement ensures that the parsing will not destroy the current structure, even if it has errors. Consider the example of changing if-statement to while-statement again. When the user removes the If symbol, the If structure is broken and hence becomes a $D$ structure. Now, if the user perhaps replaces the Bexp structure for some other Bexp structure, the $D$ structure has to be parsed. Since the $D$ structure has not changed the parsing will fail. But we still want to Bexp and Block structures to be left intact, since they are still correct.

The second requirement stems from the subtlety of referencing symbols. Since the focus is referencing some symbol, the same symbol must be preserved even after parsing — otherwise the focus will be lost. This was discussed in section 3.6.

5.6 Classes of editors

One may identify some similar structures when creating editors. Structures such as lists of items have appeared in several examples in this thesis, such as the Text and the Line editors in chapter 4. The Text editor was defined almost exactly like the Line editor and in fact, any editor for a structure that is a sequence of items can be defined in the same way. Hence, a whole class\textsuperscript{5} of editors has been identified.

Other classes can also be identified and some of them are listed in the following table:

<table>
<thead>
<tr>
<th>Editor class</th>
<th>Structure description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Seq = Item$^+$</td>
</tr>
<tr>
<td>List</td>
<td>List = LD Item$^*$ RD</td>
</tr>
<tr>
<td>SepList</td>
<td>List = LD Item (SEP</td>
</tr>
</tbody>
</table>

\textsuperscript{5}The term class here should not be confused with the classification of structure editors into syntax directed, syntax recognizing and hybrid editors. That is another kind of classification.
The symbols $\text{LD}$, $\text{RD}$ and $\text{SEP}$ can be seen as meta symbols for separator symbols and “Item” as the structure of the child elements. It would now be possible to define standard edit operations for those structures and use them as starting points when defining new editors.

The idea of using special grammars for structure editors is not new. Some systems have a limited set of grammar productions which may be used to define editors (see [15] for details). The model proposed in this thesis won’t enforce such restrictions however.
Chapter 6

Conclusions

Structure editors have been under development for more than twenty years now. So far no structure editor has really made it to the mainstream uses, but their advantages over pure text editors are so significant that their future is very promising. The use of the $D$-structures to handle broken structures and alternations, provides continuous structure editing — even when the document tree does not correspond to a syntactical correct document. I believe this approach can make structure editors as flexible as pure text editors while offering all the power of structure editing.

In order to achieve constant editing flexibility, this model proposes the use of parsers in order to repair broken structures. However, for some documents it should be possible to define editors without the need for a parser. We did so for the Text-editor in chapter 4, but also some source code documents, especially Lisp code, has a rather simple structure. So it should be possible to make a structure editor for Lisp code as well, without a parser. This has not been investigated further in this thesis though, since parsers will generally be needed for source code editing.

Some issues have not been brought up in this thesis, but may be important none the less. One such issue is the problems involved with creating structure editors that can handle infix operators with priorities (such as
arithmetic expressions). Such recursive structures may be complex to handle in a general structure editor. Perhaps structures such as arithmetic expressions could be handled as a sequence of numbers/identifiers and operators instead of keeping the complete recursive structure known to the editor. This issue is left open for future work.

The formalism is very powerful since the structure matching facilities makes it possible to handle arbitrary structures. In theory, editors for any type of structured document should be possible to define using this formalism. The introduction of attributes plays a significant role for this, although their use have not been given a lot of focus in this document.

Unfortunately it is hard to evaluate the formalism and the editor specifications when there exists no implementation. The formalism can be evaluated to some extent by using it to define editors, but the editors can not be evaluated until an implementation exists.
Appendix A

Editor specifications

A.1 Standard editors

The following editors are available in order to make it easier to create new editors. Hence, new editors can be built based on those. In order to save space, the edit functions and edit rules are compiled in tables.

A.1.1 The default editor $\mathcal{D}$

The default editor is an editor for garbage structures, i.e. broken structures. Each time it changes, it should be parsed using the \textit{parse} function, in order to make sure that document structures are recognized if possible.
### A.1. Standard editors

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$new_{D}$</td>
<td>$D[new_{Lex}]$</td>
</tr>
<tr>
<td>$pendable?(D)$</td>
<td>$True$</td>
</tr>
<tr>
<td>$first(D[X \beta])$</td>
<td>$first(X)$</td>
</tr>
<tr>
<td>$last(D[\alpha X])$</td>
<td>$last(X)$</td>
</tr>
<tr>
<td>$split(D[\alpha @x \beta])$</td>
<td>$(parse(D[\alpha]), parse(D[x \beta]))$</td>
</tr>
<tr>
<td>$split(D[\alpha @S \beta])$</td>
<td>$(parse(D[\alpha L]), parse(D[R \beta]))$</td>
</tr>
<tr>
<td></td>
<td>where $(L, R) = split(S)$</td>
</tr>
<tr>
<td>$join(D[\alpha], D[\beta])$</td>
<td>$parse(D[\alpha \beta])$</td>
</tr>
<tr>
<td>$prepend(X, D[\alpha])$</td>
<td>$parse(D[X \alpha])$</td>
</tr>
<tr>
<td>$append(X, D[\alpha])$</td>
<td>$parse(D[\alpha X])$</td>
</tr>
<tr>
<td>$dropf(D[X])$</td>
<td>$new_{D}$</td>
</tr>
<tr>
<td>$dropf(D[X \beta])$</td>
<td>$parse(D[\beta])$</td>
</tr>
<tr>
<td>$dropl(D[X])$</td>
<td>$new_{D}$</td>
</tr>
<tr>
<td>$dropl(D[\alpha X])$</td>
<td>$parse(D[\alpha])$</td>
</tr>
</tbody>
</table>

The use of $prepend$, $append$ and $pendable?$ can be illustrated by introducing the following functions:

$$mover(D[\alpha @X Y \beta]) = D[\alpha prepend(X, Y)] \text{ when } pendable?(Y)$$

$$D[\alpha Y X \beta] \text{ otherwise}$$

$$movel(D[\alpha Y @X \beta]) = D[\alpha append(X, Y)] \text{ when } pendable?(Y)$$

$$D[\alpha X Y \beta] \text{ otherwise}$$

So, $mover$ moves the focused element to the right and $movel$ moves the focused element to the left. Notice the use of $pendable?$ to see whether the structure should change or not.
A.1.2 The lexical editor

There are only symbols in the Lex-structures.

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( new_{Lex} )</td>
<td>( \text{Lex}[$] )</td>
</tr>
<tr>
<td>( \text{pendable?}(\text{Lex}) )</td>
<td>( \text{False} )</td>
</tr>
<tr>
<td>( \text{first}(\text{Lex}[x \beta]) )</td>
<td>( x )</td>
</tr>
<tr>
<td>( \text{last}(\text{Lex}[\alpha x]) )</td>
<td>( x ) (always ( $ ))</td>
</tr>
<tr>
<td>( \text{split}(\text{Lex}[\alpha @x \beta]) )</td>
<td>( (\text{Lex}[\alpha $], \text{Lex}[x \beta]) )</td>
</tr>
<tr>
<td>( \text{join}(\text{Lex}_1[\alpha $, \text{Lex}_2[\beta]) )</td>
<td>( \text{Lex}[\alpha \beta] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>( \text{Lex}[\alpha x @y \beta] )</td>
<td>( \mathcal{M} = x )</td>
</tr>
<tr>
<td>right</td>
<td>( \text{Lex}[\alpha @x y \beta] )</td>
<td>( \mathcal{M} = y )</td>
</tr>
<tr>
<td>ins(x)</td>
<td>( \text{Lex}[\alpha @y \beta] )</td>
<td>( \text{Lex}[\alpha x y \beta] )</td>
</tr>
<tr>
<td>erase</td>
<td>( \text{Lex}[\alpha x @y \beta] )</td>
<td>( \text{Lex}[\alpha y \beta] )</td>
</tr>
<tr>
<td>delete</td>
<td>( \text{Lex}[\alpha x @y \beta] )</td>
<td>( \text{Lex}[\alpha x \beta], \mathcal{M} = x )</td>
</tr>
</tbody>
</table>
A.1.3 The sequence editor Seq

A sequence has the following structure:

\[ \text{Seq} = \text{Item}^+ \]

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(new_{\text{Seq}})</td>
<td>(\text{Seq}[\text{new}_{\text{Item}}])</td>
</tr>
<tr>
<td>(\text{pendable?}(\text{Seq}))</td>
<td>(\text{True})</td>
</tr>
<tr>
<td>(\text{first}(\text{Seq}[\alpha \ X \beta]))</td>
<td>(\text{first}(X))</td>
</tr>
<tr>
<td>(\text{last}(\text{Seq}[\alpha \ X]))</td>
<td>(\text{last}(X))</td>
</tr>
<tr>
<td>(\text{split}(\text{Seq}[\alpha @x \beta]))</td>
<td>((\text{Seq}[\alpha], \text{Seq}[x \beta]))</td>
</tr>
<tr>
<td>(\text{split}(\text{Seq}[\alpha @S \beta]))</td>
<td>((\text{Seq}[\alpha L], \text{Seq}[R \beta]))</td>
</tr>
<tr>
<td>(\text{join}(\text{Seq}[\alpha], \text{Seq}[\beta]))</td>
<td>(\text{Seq}[\alpha \beta])</td>
</tr>
<tr>
<td>(\text{append}(X, \text{Seq}[\alpha]))</td>
<td>(\text{Seq}[\alpha X])</td>
</tr>
<tr>
<td>(\text{prepend}(X, \text{Seq}[\alpha]))</td>
<td>(\text{Seq}[X \alpha])</td>
</tr>
<tr>
<td>(\text{dropf}(\text{Seq}[X]))</td>
<td>(\text{Seq}[\text{new}_{\text{Item}}])</td>
</tr>
<tr>
<td>(\text{dropf}(\text{Seq}[X \beta]))</td>
<td>(\text{Seq}[\beta])</td>
</tr>
<tr>
<td>(\text{dropl}(\text{Seq}[X]))</td>
<td>(\text{Seq}[\text{new}_{\text{Item}}])</td>
</tr>
<tr>
<td>(\text{dropl}(\text{Seq}[\alpha X]))</td>
<td>(\text{Seq}[\alpha])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>(\text{Seq}[\alpha @X Y \beta])</td>
<td>(\text{Seq}[\alpha X Y \beta], M = \text{first}(Y))</td>
</tr>
<tr>
<td>left</td>
<td>(\text{Seq}[\alpha X @Y \beta])</td>
<td>(\text{Seq}[\alpha X Y \beta], M = \text{last}(X))</td>
</tr>
<tr>
<td>space</td>
<td>(\text{Seq}[\alpha @S \beta])</td>
<td>(\text{Seq}[\alpha L R \beta], M = \text{first}(R))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where ((L, R) = \text{split}(S))</td>
</tr>
<tr>
<td>space</td>
<td>(\text{Seq}[\alpha @x \beta])</td>
<td>(\text{Seq}[\alpha \text{new}_{\text{Item}} x \beta])</td>
</tr>
<tr>
<td>erase</td>
<td>(\text{Seq}[\alpha X @x \beta])</td>
<td>(\text{Seq}[\alpha x \beta])</td>
</tr>
<tr>
<td>erase</td>
<td>(\text{Seq}[\alpha S @T \beta])</td>
<td>(\text{Seq}[\alpha \text{join}(S, T) \beta])</td>
</tr>
</tbody>
</table>
A.1.4 The list editor List

A List is a sequence of items, with a starting and an ending delimiter. The structure can be described as follows, where LD and RD denotes the delimiter symbols:

\[
\text{List} = \text{LD Item}^* \text{RD}
\]

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_List</td>
<td>\text{List[LD new_Item RD]}</td>
</tr>
<tr>
<td>pendable?(List)</td>
<td>\text{True}</td>
</tr>
<tr>
<td>first(List[LD X \beta RD])</td>
<td>\text{first}(X)</td>
</tr>
<tr>
<td>last(List[LD \alpha X RD])</td>
<td>\text{last}(X)</td>
</tr>
<tr>
<td>split(List[LD \alpha @x \beta RD])</td>
<td>(\text{List}[LD \alpha RD], \text{List}[LD x \beta RD])</td>
</tr>
<tr>
<td>split(List[LD \alpha @S \beta RD])</td>
<td>(\text{List}[LD \alpha L RD], \text{List}[LD R \beta RD])</td>
</tr>
<tr>
<td>where (L, R) = split(S)</td>
<td></td>
</tr>
<tr>
<td>join(List[LD \alpha RD], List[LD \beta RD])</td>
<td>\text{List[LD \alpha \beta RD]}</td>
</tr>
<tr>
<td>append(X, List[LD \alpha RD])</td>
<td>\text{List}[LD \alpha X RD]</td>
</tr>
<tr>
<td>prepend(X, List[LD \alpha RD])</td>
<td>\text{List}[LD X \alpha RD]</td>
</tr>
<tr>
<td>dropf(List[LD X RD])</td>
<td>\text{List[LD new_Item RD]}</td>
</tr>
<tr>
<td>dropf(List[LD X \beta RD])</td>
<td>\text{List[LD \beta RD]}</td>
</tr>
<tr>
<td>dropf(List[LD X RD])</td>
<td>\text{List[LD new_Item RD]}</td>
</tr>
<tr>
<td>dropf(List[LD \alpha X RD])</td>
<td>\text{List[LD \alpha RD]}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
<td>Seq[LD \alpha @X Y \beta RD]</td>
<td>Seq[LD \alpha X Y \beta RD] M = first(Y)</td>
</tr>
<tr>
<td>left</td>
<td>Seq[LD \alpha @X \beta RD]</td>
<td>Seq[LD \alpha X \beta RD] M = last(X)</td>
</tr>
<tr>
<td>space</td>
<td>Seq[LD \alpha @S \beta RD]</td>
<td>Seq[LD \alpha L R \beta RD] M = first(R) where (L, R) = split(S)</td>
</tr>
<tr>
<td>space</td>
<td>Seq[LD \alpha @x \beta RD]</td>
<td>Seq[LD \alpha new_Item x \beta RD]</td>
</tr>
<tr>
<td>erase</td>
<td>Seq[LD \alpha @x \beta RD]</td>
<td>Seq[LD \alpha x \beta RD]</td>
</tr>
<tr>
<td>erase</td>
<td>Seq[LD \alpha @T \beta RD]</td>
<td>Seq[LD \alpha join(S, T) \beta RD]</td>
</tr>
</tbody>
</table>
A.2 A Lisp editor

This is a specification of an editor for documents of a simple Lisp-language. The structure of the document can be described as follows:

\[
\begin{align*}
\text{Lisp} &= \text{Exp}^* \\
\text{Exp} &= \text{List} \mid \text{Quote} \mid \text{String} \mid \text{Id} \mid \text{Num} \\
\text{List} &= (\text{Exp}^*) \\
\text{Quote} &= (\text{Exp}) \\
\text{String} &= (\text{Exp}) \\
\text{Id} &= /[:idChar:]^+/ \\
\text{Num} &= /[:digit:]^+/ \\
\end{align*}
\]

The terminal symbols LP, RP, QM and DQ denote the delimiters ‘(’, ‘)’, ‘‘)’ and ‘‘)’ respectively.

The Lisp-structure is a sequence, so it can be controlled by the standard Seq-editor. Likewise, the List-structure is the same as the default List-structure, with LP and RP for LD and RD, so that can be controlled by the default List-editor. The Exp structure is an alternation structure, so the D structure can be used in it’s place.

The only editors that have to be built for the complete Lisp editor, are the String and Quote editors.

String

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>newString(String)</td>
<td>String[\text{DQ } newLex \text{DQ}]</td>
</tr>
<tr>
<td>pendable?(String)</td>
<td>False</td>
</tr>
<tr>
<td>first(String[\text{DQ } S \text{DQ}])</td>
<td>first(S)</td>
</tr>
<tr>
<td>last(String[\text{DQ } S \text{DQ}])</td>
<td>last(S)</td>
</tr>
<tr>
<td>split(String[\text{DQ } @S \text{DQ}])</td>
<td>(String[\text{DQ } L \text{DQ}], String[\text{DQ } R \text{DQ}]) where ((L, R) = split(S))</td>
</tr>
<tr>
<td>join(String[\text{DQ } S_1 \text{DQ}], String[\text{DQ } S_2 \text{DQ}])</td>
<td>String[\text{DQ } join(S_1, S_2) \text{DQ}]</td>
</tr>
</tbody>
</table>
### Quote

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>newQuote</td>
<td><strong>Quote</strong>[QM newLex]</td>
</tr>
<tr>
<td>pendable?(Quote[QM S])</td>
<td><strong>pendable</strong>?(S)</td>
</tr>
<tr>
<td>first(Quote[QM S])</td>
<td><strong>first</strong>(S)</td>
</tr>
<tr>
<td>last(Quote[QM S])</td>
<td><strong>last</strong>(S)</td>
</tr>
<tr>
<td>split(Quote[QM @S])</td>
<td>(Quote[QM L], Quote[QM R]) where (L, R) = split(S)</td>
</tr>
<tr>
<td>join(Quote[QM S₁], Quote[QM S₂])</td>
<td><strong>Quote</strong>[QM join(S₁, S₂)]</td>
</tr>
<tr>
<td>append(X, Quote[QM S])</td>
<td><strong>Quote</strong>[QM append(X, S)]</td>
</tr>
<tr>
<td>prepend(X, Quote[QM S])</td>
<td><strong>Quote</strong>[QM prepend(X, S)]</td>
</tr>
<tr>
<td>dropf(Quote[QM S])</td>
<td><strong>Quote</strong>[QM dropf(S)]</td>
</tr>
<tr>
<td>dropl(Quote[QM S])</td>
<td><strong>Quote</strong>[QM dropl(S)]</td>
</tr>
</tbody>
</table>
Bibliography


Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Johan Holmquist
Linköping, 25th October 2005