Specification of a Language-Based Editor for XML documents using a Generator System

Benke József Dániel Matrikel-Nr.: 2964113 and
Halász Szilvia Matrikel-Nr.: 2964131

March 29, 2005

Project Report
International Master Program
Computational Logic

Chair for Foundations of Programming
Prof. Dr.-Ing. habil. Heiko Vogler
Institute of Theoretical Computer Science
Department of Computer Science
Dresden University of Technology

Supervisor: Dipl.-Inf. Enrico Bormann

Contents

Abstract

The objective of this project is to build a specification of XML-documents. This specification is used in generating a language-based editor with the help of the generator system [1]. The language used for the specification is Dresden Specification Language (DSL) [2]. The editor should be able to edit XML-documents that are structurally correct by means of context-free and context-sensitive syntax. The documents considered in this project must contain a DTD part [3] defining the syntactical structure of the actual XML part. Moreover, such a document can contain an XSL part [4] defining a transformation of the XML part into XHTML [5]. In the case that the whole XML-document is correct, we get XHTML code when executing the transformation.

1 Introduction

Today the eXtensible Markup Language (XML) [3] [7] is a widely accepted standard for exchanging and storing data. Many domain specific markup languages have been implemented in XML. Validating an XML-document means checking its structure against a specification which restricts the set of used tags and attributes and the way they are used together. A Document Type Definition (DTD) [3] [7] provides such a specification and is accepted most widely. The purpose of a DTD is to define the legal building blocks of an XML-document. It defines the document structure with a list of legal elements. A DTD can be declared inline in your XML-document, or as an external reference. The advantage of using DTDs is that a single DTD could be referenced by all the organization's applications. The defined structure of the data would be in a centralized resource, which means that any changes to the data structure definition would only need to be implemented in one place. All the applications that referenced the DTD would automatically use the new, updated structure.

When exchanging XML encoded data, having a document specification is important for interpreting the document's content because XML allows the introduction of new tag and attribute names at will. With XML we can use any tags we want, however the meaning of these tags is not automatically understood by the browser. As an example, `<table>` could mean an HTML table but also a piece of furniture. Because of the domain independent nature of XML, there is no standard way to display an XML-document. In order to display XML-documents, it is necessary to have a mechanism to describe how the document should be displayed. One of these mechanisms is Cascading Style Sheets (CSS) [7] [8], but XSL (eXtensible Stylesheet Language)[4] [8] is the preferred style sheet language of XML, and XSL is far more sophisticated than the CSS used by HTML.

The goal of this project is the specification of a language-based editor for XML-documents. The documents considered in this project must contain a DTD part defining the syntactical structure of the actual XML part. Moreover, such a document can contain an XSL part defining a transformation of the XML content into another markup language such as XHTML [5].

Language-based editors are editors which are aware of the syntactic rules of a specific programming language and provide a framework for the user to enter source code. Language-based editors can be programmed by hand, or can be generated using a generator system. The generated editor makes sure that the program edited is structurally correct according to the context-free syntax and checks for violations of the context-sensitive restrictions.

A generator system takes a specification of a particular formal (programming-) language as input and generates a language-based editor for that specified language. The person who specifies a formal (programming-) language as input for a generator system is called specifier, and the person who edits a program in the editor, which is output of a generator system is called user. A specifier should specify a formal (programming-) language in a certain specification language such that a generator system can understand it.

A specification of a language consists of a context-free syntax [6] and a context-sensitive syntax [6]. The context-free syntax is given by the rules of the Extended Backus-Naur Form (EBNF), which is a superset of that language. The context-sensitive syntax is specified as a list of rules (or restrictions) of that language to restrict the EBNF rules to the language. First we can formulate the restrictions into normal sentences. After it we describe these restrictions in a more formal way for a generator system that can work with this specification. The generator system takes this specification as input and generates a language-based editor for manipulating objects according to the given rules.

The generator system in [2] takes a specification of a language as input and generates a language-based editor for that language. It understands the specification language DSL [2], so the specification must be an element of DSL. With the Specification language DSL and the generator system we are able to build a language based editor for a particular language L by specifying it in DSL. This specification has to deal with the context-free syntax and context-sensitive restrictions of L. An important part of DSL is the attribute grammar formalism, furthermore it is integrated with an extension of MSO-logic over trees.

This report is structured as follows. After introducing the topic and the main tasks of our project in Chapter 2, the Chapters 3, 4 and 5 present the syntax of XML, as well as means of validating and transforming XML by means of DTD and XSL. Then, the steps of the specification in DSL are presented (Chapter 6) and a short manual for the user of the editor is given (Chapter 4).

2 Running example

In this chapter we present the running example used throughout this report. It is based on an XML grammar describing faculties, students (and their properties) of the Dresden University of Technology. For validating such data a DTD was developed. Furthermore, we also created an XSL stylesheet to transform such documents into XHTML.

2.1 XML with an inline DTD - Example

The following code presents an instance of our XML grammar. The DTD is included in the beginning of the document. Note that it is also possible to reference external DTDs.

This DTD allows to define the Dresden University of Technology and its faculties. Each faculty has a name, an address and at least one student visiting it. Students are characterized by their names, their matriculation number, the current semester of their studies, as well as by the name of the course they attend.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TUDresden [
<!ELEMENT TUDresden      Faculty*>
<!ELEMENT Faculty        (Facultyname, Facultyaddress, Student+)>
<!ELEMENT Facultyname    (#PCDATA)>
<!ELEMENT Facultyaddress (#PCDATA)>
<!ELEMENT Student        (Name, Semester)>
<!ELEMENT Name           (#PCDATA)>
<!ELEMENT Semester       (#PCDATA)>
<!ATTLIST Student    immatriculation  ID        #REQUIRED
                     interruptstudies (yes|no)  #IMPLIED
                     course           CDATA     #REQUIRED>
]>

<TUDresden>
  <Faculty>
    <Facultyname>Computer Science</Facultyname>
    <Facultyaddress>Hans Grundig Str. 25</Facultyaddress>
    <Student immatriculation="a1234"
             interruptstudies="yes"
             course="FMC">
      <Name>Benke Jozsef Daniel</Name>
      <Semester>5.</Semester>
    </Student>
    <Student immatriculation="c1339"
             course="IFPL">
      <Name>Martina Schneider</Name>
      <Semester>3.</Semester>
    </Student>
    </Student>
    <Student immatriculation="c3356"
             course="IFPL">
      <Name>Matthias Sammer</Name>
      <Semester>4.</Semester>
    </Student>
  </Faculty>
  <Faculty>
    <Facultyname>Mathematics</Facultyname>
    <Facultyaddress>Zellescher Weg 16</Facultyaddress>
    <Student immatriculation="a2345"
             course="Automata">
      <Name>Halasz Szilvia</Name>
      <Semester>5.</Semester>
    </Student>
  </Faculty>
</TUDresden>
```

2.2 XSL - Example

In order to display the data contained in our running example's XML instance, we defined an XSL stylesheet aiming at transforming that data to XHTML. The presentation is structured into nested XHTML tables.

```
<?xml version="1.0" encoding="UTF-8">
<xsl:stylesheet version="1.0" xmlns:xsl=
"html://www.w3.org/1999/XSL/Transform">
  <xsl:template match="TUDresden">
    <xsl:element name="html">
      <xsl:element name="body">
        <xsl:element name="table">
          <xsl:attribute name="border"> <xsl:text>2</xsl:text>
          </xsl:attribute>
          <xsl:apply-templates select="Faculty"/>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
  <xsl:template match="Faculty">
    <xsl:element name="tr">
      <xsl:element name="td">
        <xsl:element name="b"> <xsl:text>Faculty</xsl:text>
        </xsl:element>
      </xsl:element>
      <xsl:element name="td">
        <xsl:value-of select="Facultyname"/>
      </xsl:element>
    </xsl:element>
    <xsl:for-each select="Student">
    <xsl:element name="tr">
      <xsl:element name="td">
        <xsl:element name="b"> <xsl:text>Student</xsl:text>
        </xsl:element>
      </xsl:element>
      <xsl:element name="td">
        <xsl:text>name:</xsl:text>
        <xsl:value-of select="Name"/>
        <xsl:text>, semester:</xsl:text>
        <xsl:value-of select="Semester"/>
      </xsl:element>
    </xsl:element>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

2.3 Result: XHTML code

Following code snippet presents the XHTML code generated by our XSL stylesheet from our XML instance document.

```
<!DOCTYPE TUDresden
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <body>
    <table border="2">
      <tr>
        <td> <b>Faculty</b> </td>
        <td>Computer Science</td>
      </tr>
      <tr>
        <td>
          <b>Student</b>
        </td>
       <td>name:Benke Jozsef Daniel, semester:5.</td>
      </tr>
      <tr>
        <td>
          <b>Student</b>
        </td>
        <td>name:Martina Schneider, semester:3.</td>
      </tr>
      <tr>
        <td>
          <b>Student</b>
        </td>
        <td>name:Matthias Sammer, semester:4.</td>
      </tr>
      <tr>
        <td> <b>Faculty</b> </td>
        <td>Mathematics</td>
      </tr>
      <tr>
        <td>
          <b>Student</b>
        </td>
        <td>name:Halasz Szilvia, semester:5.</td>
      </tr>
    </table>
  </body>
</html>
```

3 Syntax of XML-documents and XHTML code

An XML-document builds up from a DTD, an XML and an XSL part. The purpose of a Document Type Definition is to define the legal building blocks of an XML-document. It defines the document structure with a list of legal elements. A DTD can be declared inline in our XML-document, or as an external reference. We are focusing on XML-documents with an inline DTD definition. The purpose of an eXtensible Markup Language is to store and transfer data and to describe the data. The main purpose of XSL is to provide a way to transform or format the raw data of one XML-document into another form. If we have a complete and valid XML-document, we can transform it into XHTML code. The purpose of XHTML is to be a transitional language between HTML and XML. XHTML is a combination of XML and HTML. Now we detail the parts of the XML-documents.

3.1 Syntax of the DTD

Internal DOCTYPE declaration: If the DTD is included in our XML source file, it should be wrapped in a DOCTYPE definition with the following syntax:

```
<!DOCTYPE root-element [element-declarations]>
```

With a DTD, each of our XML files can carry a description of its own structure with it. Thus independent groups of people can agree to use a common DTD for interchanging data.

3.1.1 DTD - XML building blocks

The building blocks of XML-documents:
The main building blocks of the XML-documents are tags like `<body>....</body>`.
Seen from a DTD point of view, all XML-documents are made up by the following simple building blocks:

- **Elements:** Elements are the main building blocks of XML-documents. Examples of XML elements could be `"TUDresden"` and `"Faculty"`. Elements can contain text, other elements, or be empty.
- **Tags:** Tags are used to markup elements. A starting tag like `<Student>` marks up the beginning of an element, and an ending tag like `</Student>` marks up the end of an element.
- **Attributes:** Attributes are always placed inside the starting tag of an element. Attributes always come in name/value pairs, like `immatriculation="a2345"`.
- **Entities:** Entities are variables used to define common text. Entities can be used in the DTD to create abbreviations and special characters. Entities are given names, and are used by inserting the entity reference into a document. The XML parser replaces the entity reference, with the entity replacement text.

- **PCDATA(Parsable Character DATA):** PCDATA means parsed character data. Think of character data as the text found between the start tag and the end tag of an XML element. PCDATA is a text that will be parsed by the XML Parser - any markup text will be treated as markup.
- **CDATA:** CDATA also means character data. CDATA is text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded.

3.1.2 DTD - element declarations

Elements are declared in the DTD using a simple but strict syntax. An element declaration has the following syntax:

```
<!ELEMENT element-name element-content>
```

There are four ways to define the `element-content`:

- EMPTY : When an element is declared with the EMPTY keyword, it means that the element will not hold any information. This is generally used for special tags.
- ANY : When an element is declared with the ANY keyword, it means that the element can contain any information that the author wants it to.
- PCDATA : When an element is declared with the PCDATA keyword, it will hold the type of information Parsable Character DATA.
- With children : When an element is declared with the names of other elements in it, this defines a parent-child relationship.

When an element is declared with children, it will also define how the children can be used inside an XML-document and also the order that they are allowed to appear in an XML document. There are several ways for defining children elements:

exactly one occurrence	: `child-element`
minimum one occurrence	: `child-element+`
zero or more occurrence	: `child-element*`
zero or one occurrence	: `child-element?`
sequence occurrences	: `child-element1,child-element2,...`
alternative occurrences	: `child-element1|child-element2|...`

Some examples:

- Declaring an element having character data

```
<!ELEMENT Facultyname (#PCDATA)>
```

- Declaring zero or more occurrences of the same element

  ```
  <!ELEMENT TUDresden   Faculty*>
  ```

- Declaring mixed content

  ```
  <!ELEMENT Faculty (Facultyname, Facultyaddress, Student+)>
  ```

3.1.3 DTD - attribute declarations

An attribute declaration has the following syntax:

```
<!ATTLIST element-name
   attribute-name1 attribute-type1 default-value1
   attribute-name2 attribute-type2 default-value2
   ...>
```

- The attribute-type can have the following values:

Value	**Explanation**
CDATA	the value is character data
(*en*1|*en*2|..)	the value must be one from an enumerated list
ID	the value is a unique id
IDREF	the value is the id of another element
IDREFS	the value is a list of other ids
NMTOKEN	the value is a valid XML name
NMTOKENS	the value is a list of valid XML names
ENTITY	the value is an entity
ENTITIES	the value is a list of entities

- The default-value can have the following values:

Value	**The default value of the attribute**
#REQUIRED	the attribute value must be included in the element
#IMPLIED	the attribute does not have to be included
#FIXED "value"	the attribute is required and is set to a fixed value
"attr-value"	default value of the attribute

In this project we are concentrating on the attribute-types CDATA, `(en1|en2|..)`, ID, NMTOKEN and ENTITY and attribute-values #REQUIRED, #IMPLIED and #FIXED "value".

3.1.4 DTD - entities

Entities are variables used to define shortcuts to common text. They can be declared internal, or external.

- Internal Entity Declaration

  ```
  <!ENTITY entity-name "entity-value">
  ```

- External Entity Declaration

  ```
  <!ENTITY entity-name SYSTEM "URI/URL">
  ```

In this project we are only dealing with internal entity declarations, because in the validation we are verifying only local resources.

3.2 Syntax of XML

3.2.1 XML - elements

Like in the next example, the XML declaration should always be included as the first line in the document. It defines the XML version of the document. In this case the document conforms to the 1.0 specification of XML:

```
<?xml version="1.0"?>
```

The next line defines the first element of the document (the root element):

```
<TUDresden>
```

The next line defines a child element of the root (Faculty):

```
<Faculty> ... </Faculty>
```

The last line defines the end of the root element:

```
</TUDresden>
```

All XML elements must have a closing tag:
In XML all elements must have a closing tag like this:

```
<Student>Benke Jozsef Daniel</Student>
```

or, if the element does not contain any text or subnodes but attributes, it looks like this:

```
<Student course = "FMC" />
```

All XML documents must have a root tag:
All XML documents must contain a single tag pair to define the root element. All other elements must be nested within the root element. All elements can have sub (children) elements. Sub elements must be in pairs and correctly nested within their parent element:

```
<TUDresden>
     <Faculty>
          <Facultyname>
          </Facultyname>
          ...
     </Faculty>
</TUDresden>
```

XML tags are case sensitive:
XML tags are case sensitive. The tag `<Student>` is different from the tag `<student>`. Opening and closing tags must therefore be written with the same case.

3.2.2 XML - attributes

Usually, or most common, attributes are used to provide information that is not a part of the content of the XML document. XML attributes are normally used to describe XML elements or to provide additional information about elements. XML elements can have attributes in name/value pairs just like in HTML. In XML the attribute value must always be quoted. Attributes are always contained within the start tag of an element.
Examples:

```
<Student immatriculation="a2345"
         course="Automata">
...
</Student>
```

3.3 Syntax of XSL

3.3.1 Purpose of XSL

The extensible Stylesheet Language (XSL)[4] [8] is a language for expressing style sheets. The specification contains three parts: XSL Transformations (XSLT), XSL Formatting Objects (XSL-FO) and XPath. We are only dealing with XPath and XSLT.

XPath: XPath is a set of rules for defining parts of an XML-document. It allows you to select specific elements or nodes in an XML-document. You can locate the data you need access to in an XML file using XPath statements. XPath is key to XSL because it allows you to select the parts of the XML-document you want output or transform to another

format.
XPath defines paths to access specified elements within an XML file. For example, you can explicitly call a particular element by its name and find its attribute if necessary. Considering our running example, we can explicitly select the `<Name>` of all student element with the following XPath statement:

```
/TUDresden/Faculty/Student/Name
```

XSL Transformations (XSLT): The aim of XSL Transformations is to transform an XML-document into another text-based document. The most common use for XSLT is to transform an XML-document into HTML or XHTML. This allows the document writer to customize the output of the transformed file into something different than the input XML data file. The document could leave out elements in the transformation, create new elements, sort elements, rearrange elements, and even perform checks on the elements.
Let us see how to transform an XML-document into XHTML using XSLT.
The root element that declares the document to be an XSL style sheet is `<xsl:stylesheet>`. Every XSL document has to contain the correct Style Sheet Declaration:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

3.3.2 Syntax of XSLT elements

We will see the syntax of the XSLT elements. We concentrate on the most important elements, such as: `apply-templates, attribute, choose, element, for-each, if, sort, stylesheet, templates, text` and `value-of`.
Almost every XSLT element can have zero or more sub XSL-elements, for example denoted by `Content:(xsl:sort)*`. It means, that the given XSL-element can contain zero or more `<xsl:sort>` elements. We do not discuss every possibility for subnodes, just the most important.

Stylesheet: Defines the root element of a style sheet.

```
<xsl:stylesheet id="name" version="version">

     Content:(<xsl:template>*)

</xsl:stylesheet>
```

The `name` is a unique id for the style sheet. The `version` specifies the XSLT version of the style sheet.

Template: Rules to apply when a specified node is matched.

```
<xsl:template match="pattern" name="name">

  Content:template+

</xsl:template>
```

The `pattern` identifies in XPath the source node or nodes to which the rule applies. If this attribute is omitted there must be a name attribute. The `name` specifies a name for the template. If this attribute is omitted there must be a match attribute.

Element: It creates an element node in the output document.

```
<xsl:element name="name">

  Content:template+

</xsl:element>
```

The `name` specifies the name of the element to be created.

Attributes: The `<xsl:attribute>` element is used to add attributes to a created element.

```
<xsl:attribute name="attributename">

  Content:template+

</xsl:attribute>
```

The `attributename` specifies the name of the attribute.

Text: It writes literal text to the output.

```
<xsl:text>

  Content:#PCDATA

</xsl:text>
```

Value-of: It extracts the value of a selected node.

```
<xsl:value-of select="expression"/>
```

An `expression` is an XPath expression that specifies which node/attribute to extract the value from.

For-each: It loops through each node in a specified node set.

```
<xsl:for-each select="expression">

  Content:(xsl:sort*,template+)

</xsl:for-each>
```

An `expression` specifies the nodes to be processed.

Sort: It sorts the output in alphabetical order.

```
<xsl:sort select="expression" order="ascending"|"descending"/>
```

An `expression` is an XPath expression that specifies the nodes to be sorted. The alternatives of the attribute ”order” are `ascending` and `descending` for the order of strings.

If: It contains a template that will be applied only if a specified condition is true.

```
<xsl:if test="expression">

  Content:template+

</xsl:if>
```

An `expression` specifies the condition to be tested.

Choose: It is used in conjunction with `<when>` and `<otherwise>` to express multiple conditional tests. If no `<xsl:when>` is true, the content of `<xsl:otherwise>` is processed. If no `<xsl:when>` is true, and no `<xsl:otherwise>` element is present, nothing is created.

```
<xsl:choose>

  Content:(xsl:when+,xsl:otherwise?)

</xsl:choose>
```

The `<xsl:when>` element evaluates an expression, and if it returns true, an action is performed.

```
<xsl:when test="boolean-expression">

  Content: template+

</xsl:when>
```

The `boolean-expression` is required, it specifies a Boolean expression to be tested.
The `<xsl:otherwise>` element specifies a default action for the `<choose>` element

```
<xsl:otherwise>

  Content:template+

</xsl:otherwise>
```

It has no attributes.

Apply-templates: It applies a template rule to the current element.

```
<xsl:apply-templates select="expression"/>
```

An `expression` specifies the nodes to be processed.

3.4 Usage of XML and XSL together

Since XML does not use predefined tags (we can use any tags we like) the meanings of these tags are not well understood. For example the `<table>` could mean an HTML table or a piece of furniture, and a browser does not know how to display it. There must be something in addition to the XML-document that describes how the document should be displayed, and that is XSL. XSL stands for eXtensible Stylesheet Language. XSL consists of more parts, but we focus on the most important part of the XSL standards, XSLT (XSL Transformations) which is a language for transforming XML-documents.

Think of XSL as a set of languages that can transform XML into XHTML, filter and sort XML data, define parts of an XML-document, format XML data based on the data value, like displaying negative numbers in red, and output XML data to different media, like screens and paper.

3.5 About XHTML - eXtensible HyperText Markup Language

3.5.1 Syntax of XHTML

XHTML[5] [8] is a combination of HTML and XML. XHTML consists of all the elements in HTML combined with the syntax of XML. XML and HTML were designed with different goals:

- XML was designed to describe data and to focus on what data is.
- HTML was designed to display data and to focus on how data looks like.

The syntax of an XHTML document is very similar to a HTML document but it has more restrictions. We mention some more XHTML syntax rules:

- XHTML elements must be properly nested within each other, i.e. the innermost element must be closed before one can close any other elements. For example: `<a><b>...</b></a>`
- XHTML documents must be well-formed, i.e. all XHTML elements must be nested within the `<html>` root element. All other elements can have sub (children) elements. Sub elements must be in pairs and correctly nested within their parent element.
- All XHTML elements must be closed. Non-empty elements must have an end tag. Empty elements must either have an end tag or the start tag must end with `/>`.

- Attribute values must be quoted such that:
  `<Student immatriculation = "a2345">`
- Attribute minimization is forbidden. The correct form is like in the following example:
  correct: `<Student immatriculation = "a2345">`
  wrong: `<Student immatriculation>`
- The XHTML DTD defines mandatory elements, i.e. all XHTML documents must have a DOCTYPE declaration. The html and body elements must be present, and the title must be present inside the head element. The minimum XHTML document template is the following:

  ```
  <!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  <html>
    <body>
      Body text
    </body>
  </html>
  ```

3.5.2 A simple example

Let us take a quick look at a simple XML node, some XSL elements and their result, i.e. an XHTML node.

XML:

```
<Facultyname>Computer Science</Facultyname>
```

XSL:

```
<xsl:element name = "td">
  <xsl:value-of select = "Facultyname"/>
</xsl:element>
```

XHTML:

```
<td>Computer Science</td>
```

We have chosen a simplified example to represent how we can make an XHTML node from an XML node with help of XSL elements. First, we consider the XSL elements. The value of the attribute `"name"` of the element `<xsl:element>` is transmitted to the XHTML node. It will be a tag in the XHTML document, in our case: `<td>`. Both in the XSL and in the XHTML document the element `"td"` has the open and the closed tag. Secondly, we consider the `<xsl:value-of>` element, that is we have to look for the tag - given in the `"select"` attribute - in the XML-document. If we find such a tag in the XML-document then we transmit it to an XHTML document. It will be the text in the XHTML document, in our case: `Computer Science`.

3.6 Context-Free Syntax of XML-documents

In the previous sections we had a look at the syntax of the XML-documents, generally. Now we use a formal way to describe the syntax of the XML-documents. The context-free syntax is given by the following rules of an EBNF-definition, where the non-terminal symbols are enclosed between `<` and `>`, the terminal symbols are enclosed between " and ", and `<Root>` is the start symbol.

```
<Root>
  ::= <DtdXmlXsl>

<DtdXmlXsl>
  ::= "<?xml version = \"1.0\" encoding = UTF-8?>"
      <DtdRoot> <XmlRoot> <XslRoot>
      | <XhtmlRoot>

<Identifier>
  ::= ('a' | ... | 'Z') {'0' | ... | '9' | 'a' | ... | 'Z' | '_'}

<String>
  ::= '0' | ... | '9' | 'a' | ... | 'Z' | '_' | '&' | ';' | '?'
      | '!' | ':'| ',' | '.'

<QuotedString>
  ::= ' \" ' <String> ' \" '
```

The DTD-part of the EBNF-definition:

```
<DtdRoot>
  ::= "<!DOCTYPE  " <Identifier>  "["
      <EntListOpt> <ElemList> <AttrListOpt>
      "]>"
      | "DTD"

<EntListOpt>
  ::= [<EntList>]

<EntList>
  ::= <Ent> {<Ent>}

<Ent>
  ::= "<!ENTITY " <Identifier> <QuotedString> " >"

<ElemList>
  ::= <Elem> {<Elem>}

<Elem>
```

```
  ::= "<!ELEMENT " <Identifier> <TagStruc> " >"

<AttrListOpt>
  ::= [<AttrList>]

<AttrList>
  ::= <Attr> {<Attr>}

<Attr>
  ::= "<!ATTLIST " <Identifier> <AttrElems> ">"

<AttrElems>
  ::= <AttrEl> {<AttrEl>}

<AttrEl>
  ::= <Identifier> <DataType>

<DataType>
  ::= "CDATA" <DefValue> | "ID" <DefValueID> | "ENTITY" <DefValue>
      | "NMTOKEN" <DefValue> | "(" <Type_AltList> ")" <DefValue>

<DefValueID>
  ::= "#REQUIRED" | "#IMPLIED"

<Type_AltList>
  ::= <String> | <String> { | <String>}

<DefValue>
  ::= "#REQUIRED" | "#FIXED" <QuotedString>
      | "#IMPLIED"

<TagStruc>
  ::= <CombineComplex> | <Given>

<Given>
  ::= "ANY" | "EMPTY"

<CombineComplex>
  ::= "(#PCDATA)" | <CombineSimple> | <CombineSimple> "?"
      | <CombineSimple> "*" | <CombineSimple> "+"

<CombineSimple>
  ::= "(" <COS_AltList> ")" | "(" <COS_SeqList> ")" | <Identifier>

<COS_AltList>
  ::= <CombineComplex> "|" <CombineComplex> {"|" <CombineComplex>}
```

```
<COS_SeqList>
  ::= <CombineComplex> "," <CombineComplex> {"," <CombineComplex>}
```

The XML-part of the EBNF-definition:

```
<XmlRoot>
  ::= "<" <Identifier> <XmlAttrListOpt> ">" <XmlText>
      <XmlNodeList> "</" <Identifier> ">"
      | "XML"

<XmlAttrListOpt>
  ::= [<XmlAttrList>]

<XmlAttrList>
  ::= <Identifier> " = " <QuotedString>
      {<Identifier> " = " <QuotedString>}

<XmlNodeList>
  ::= {<XmlNode>}

<XmlNode>
  ::= "<" <Identifier> <XmlAttrList> "/>"
      | "<" <Identifier> <XmlAttrListOpt> ">" <XmlText>
        <XmlNodeList> "</" <Identifier> ">"

<XmlText>
  ::= {<String>} | "&" <Identifier>
```

The XSL-part of the EBNF-definition:

```
<XslRoot>
  ::= "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>
      <xsl:stylesheet version = \"1.0\"
      xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">"
      <XslTemplateList> "</xsl:stylesheet>"
      | "XSL"

<XslTemplateList>
  ::= <XslTemplate> {<XslTemplate>}

<XslTemplate>
  ::= "<xsl:template " <XslTemplateAttrMatch> ">"
      <XslNodeListM> "</xsl:template>"
```

```
<XslTemplateAttrMatch>
  ::= "match = \" " <XslPathValue> " \" "

<XslTemplateAttrName>
  ::= "name = \" " <Identifier> " \" "

<XslNodeListM>
  ::= {<XslNodeListMore>}

<XslNodeListMore>
  ::= "<xsl:apply-templates select =" \" <XslPathValue> " \" />"
      | "<xsl:element name = \" " <Identifier> " \" >"
        <XslNodeAttribute> <XslNodeListL> "</xsl:element>"
      | "<xsl:choose>" <XslNodeWhenList> <XslNodeOtherwise>
        "</xsl:choose>"
      | "<xsl:for-each select = \" " <XslPathValue> " \" >"
        <XslSort> <XslNodeListM> "</xsl:for-each>"
      | "<xsl:if test = \" " <Expression> " \" >" <XslNodeListM>
        "</xsl:if>"
      | "<xsl:value-of select = \" " <XslPathValue> " \" >"
      | "<xsl:text>" <TextValue> "</xsl:text>"

<XslNodeAttribute>
  ::= {"<xsl:attribute name = \" " <Identifier> " \" >"
      <XslNodeListLess> "</xsl:attribute>"}

<XslNodeWhenList>
  ::= <XslNodeWhen> {<XslNodeWhen>}

<XslNodeWhen>
  ::= "<xsl:when test = \" " <Expression> " \" >" <XslNodeListM>
      "</xsl:when>"

<XslNodeOtherwise>
  ::= [<XslNodeListM>]

<XslNodeListLess>
  ::= "<xsl:value-of select = \" " <XslPathValue> " \" />"
      | "<xsl:text>" <TextValue> "</xsl:text>"

<Expression>
  ::= <Identifier> <Relation> <String>

<Relation>
  ::= "="
```

```
<XslPathValue>
  ::= <XslPathValueList> | <XslPathList> "@" <Identifier>

<XslPathList>
  ::= {<Identifier> / <XslPathList>}

<XslPathValueList>
  ::= <Identifier> {/ <Identifier>}

<XslSort>
  ::= "<xsl:sort select = \" " <XslSortAttr>  " \" />"
      {"<xsl:sort select = \" " <XslSortAttr>  " \" />"}

<XslSortAttr>
  ::= <XslSortAttrSelect> <XslSortAttrOrder>

<XslSortAttrSelect>
  ::= "select = " <XslPathValue>

<XslSortAttrOrder>
  ::= "order = " <XslSortOrder>

<XslSortOrder>
  ::= "ascending" | "descending"
```

The XHTML-part of the EBNF-definition:

The context-free syntax of the XHTML documents is the same as the context-free syntax of the XML part. That is why we do not detail it.
`<XhtmlRoot>` = `<XmlRoot>`

3.7 Context-sensitive restrictions of XML-documents

A ”well formed” XML-document is a document that conforms to the XML syntax rules that we described in the previous chapters and is provided by the context-free syntax. Additionally to decide if an XML-document is a ”valid” XML-document we have to check if the list of context-sensitive restrictions are fulfilled by the document. ”Validating” an XML-document means that the XML-documents structure has to be compared with the rules of the Document Type Definition (DTD). In the next chapter we will focus on validating well formed XML-document against a DTD. In order to perform this check we have to create a list of context-sensitive restrictions.

Context-sensitive restrictions of DTD:

- The name in the DTD followed by the `[element-declarations]`, the name of the first element declared, and the name of the root tag in the XML have to be the same.
- In the element declarations every name which we use on the right side has to be declared.

- Every declared element has to be used in some `element-content` declaration.
- If we define attributes of an element we have to assure that the element is already declared.
- Every element and attribute declaration in the DTD has to be unique.

Context-sensitive restrictions of XML:

- We have to check if the tags in our XML-document are nested properly i.e., if they correspond to the structure declared in the DTD.
- Every XML node has to own the required attributes according to the declaration in the DTD. Additionally all his attributes have to own values according to this declaration. The optional attributes are not required, but if they are present their values must also respect the declaration.
- Inserted entity references as attribute values or in parsed character data have to be declared in the DTD.

Furthermore, we consider which restrictions according to the XSL part of an XML-document should be checked.
Context-sensitive restrictions of XSL:

- Every element given in expression of `select="expression"` and every pattern of `match="pattern"` have to be declared in the DTD.
- The XPath in every expression of `select="expression"` and in every pattern of `match="pattern"` have to be correctly defined.
- The attribute `<xsl:template>` has to have at least one of the attribute `match` and `name`.
- The XSL part the `html` and `body` elements have to be present.

4 Validating XML against a Document Type Definition (DTD)

4.1 What is validation about

XML is a meta-markup language that is fully extensible. As long as it is well formed, XML authors can create any XML structure in order to describe their data. An XML author cannot be sure that the structure he created won't be changed by another XML author. There needs to be a way to ensure that the XML structure cannot be changed at random. A DTD provides a roadmap for describing and documenting the structure that makes up an XML-document. A DTD can be used to determine the validity of an XML-document.
An XML-document is considered "well formed" if that document is syntactically correct according to the syntax rules of XML 1.0. However, that does not mean the document is necessarily valid. In order to be considered valid, an XML-document must be validated, or verified, against a DTD or XML schema. A DTD defines the elements required by an XML-document, the elements that are optional, the number of times an element should (could) occur, and the order in which elements should be nested. DTD markup also defines the type of data that may occur in an XML element and the attributes that may be associated with those elements. A document, even if well formed, is not considered valid if it does not follow the rules defined in the DTD.

When an XML-document is validated against a DTD by a validating XML parser, the XML-document is checked to ensure that all required elements are present and that no undeclared elements have been added. The hierarchical structure of elements defined in the DTD must be maintained. The values of all attributes are checked to ensure that they fall within defined guidelines. No undeclared attributes are allowed and no required attributes may be omitted. In short, every last detail of the XML-document from top to bottom is defined and validated by the DTD.

4.2 A simple example

Let us take a look at a simple XML-document using an internal DTD.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Student [
<!ELEMENT Student (#PCDATA)>
]>

<Student>
  Halasz Szilvia
</Student>
```

The internal DTD is contained within the Document Type Declaration, which begins with `<!DOCTYPE` and ends with `]>`. The Document Type Declaration will appear between the XML declaration and the start of the document itself (the document or root element) and identify that section of the XML-document as containing a Document Type Definition.

Following the Document Type Declaration (DOCTYPE), the root element of the XML-document is defined (in this case, Student). The DTD tells us that this document will have a single element, Student, that will contain parsed character data `#PCDATA`. Note that the Document Type Declaration should not be confused with the Document Type Definition. These are two exclusive items. The Document Type Declaration is the area of the XML-document after the XML declaration that begins with `<!DOCTYPE` and ends with `]>`. It actually encompasses the Document Type Definition. The Document Type Definition is contained within an opening bracket (`[`) and a closing bracket (`]`).
The above example shows us a well-formed XML-document. Additionally, because the XML-document contains a single element, Student, which contains only parsed character data, according to the DTD. Therefore, it is also valid XML-document.

4.3 Presenting our approach

In fact, we have to reformulate the above verbally formulated rules for the generator in such a formal way which it can handle. This way the generated editor can check the given restrictions, whether the actually processed document fulfills these restrictions. For this we use different techniques, such as attribution, MSO logic and the auxiliary functions. If these restrictions are fulfilled, then the document is considered valid, like in our running example. In the contrary case, the editor gives an error message to the specifier, that the document which it processed, is not considered valid.
One of the main points of our task is to check proper nestings. Let us look at the next row from our running example:

```
<!ELEMENT Faculty (Facultyname, Facultyaddress, Student+)>
```

In words this means that every element `Faculty` is containing exactly one `Facultyname`, exactly one `Facultyaddress` and one ore more `Student` elements. This could be represented also as following: `Faculty = (Facultyname, Facultyaddress,Student+)`. So if we found the tag Faculty in the XML-document, we have to check if it owns the corresponding substructure.
For example a valid substructure would be:

```
<Faculty>
  <Facultyname>
   . . .
  </Facultyname>
  <Facultyaddress>
   . . .
  </Facultyaddress>
  <Student >
   . . .
  </Student>
  <Student >
   . . .
  </Student>
</Faculty>
```

In the next steps we will try to define an algorithm to verify if the XML-document hold the specified structure declared in the inline DTD part. To demonstrate the main parts of this algorithm we switch from the internal data type representations to a more intuitive one, and we will discuss the problem on a more abstract level.
Let us go back to our simplified problem from above. By parsing the XML-document, we have to check the following. For every XML tag we check if it is totally declared in XML and DTD part. If yes, we have to check if those are corresponding to each other. When we pass through the XML-document and arrive to the tag Faculty, we check how it is declared in the DTD. We compare this information to the tag-substructure within the XML content, and from this information we can decide if the XML tag satisfies the structures declared in DTD.
Let us see how we formulate the possessed information as being useful for the generator. For transferring the information from the DTD to XML we use attributes in this case the attribute `en_type`, which will be discussed in Chapter 6.3 in detail. We pick this information from the DTD and formulate into a regular expression. In our case this could be written in a similar way as:

```
Facultyname Facultyaddress Student+
```

After this we generate a word, and an alphabet from the tag-substructure and the tag declaration. In our case the word is `Facultyname Facultyaddress Student Student` and the alphabet is {`Facultyname, Facultyaddress, Student`}.
The environment of the generator assures for us a so-called `match` function, which will be presented in detail in Chapter 6.8 together with the functions `makeAlphabet`, `xmlNodeToWord` and `tagStrucToRegExp`. By using this function together with the previously prepared information, we verify if the tag-substructure and the declaration from DTD are compatible. In fact, in this way we can solve the problem by using a well known computer science method, i.e. the regular languages and words recognition. If we get False as results of the function called `match` we warn at the tag `Faculty` the user, that the actual document cannot be considered valid.

The next algorithm description tries to present the above, but in a more accurate way. Note that the below is also not a formal description, and serves only for presenting the ideas. The rules and specifications which serve as input for the generator are of a more concrete and strict level, which we do not present here, but this will be shown in the corresponding chapters.

```
for every xmlNode {
      match (makeAlphabet (en_type xmlNode) xmlNode)
            xmlNodeToWord xmlNode
            (tagStrucToRegExp (en_type xmlNode))
}
```

With this algorithm we are able to recognize if the parsed `xmlNode` has the corresponding substructure according to the declaration in the DTD part. The returned boolean value can be used in the unparsing part to generate an error message that the corresponding context-sensitive restriction is violated. The generation of the error message is presented in Chapter 6.5.1.

Lets discuss a similar but another problem:
Adding attribute declarations to an existing DTD is easy. For each attribute, we need to specify to which element it is declared to, what the attribute name is, what type of values it may have, and whether the attribute is optional or not. This information is specified like following in our running example:

```
<!ATTLIST Student    immatriculation  ID         #REQUIRED
                     interruptstudies (yes|no)   #IMPLIED
                     course           CDATA      #REQUIRED>
```

This means that if we find the tag Student in the XML-document, we have to check if it owns the corresponding attributes. For example a valid substructure would be:

```
<Student immatriculation="a1234"
         interruptstudies="yes"
         course="FMC">
 ...
</Student>
```

To check the values of all attributes to ensure that they fall within defined guidelines and no undeclared attributes are used and no required attributes are omitted we use similar techniques as in the nesting checking of XML elements. We transport the needed information from the DTD part to the XML part as discussed in Chapter 6.3 and use it to validate our document. In the validation process we use the `match` function again and we compare the attributes at the XML tag with his declaration from the DTD to be able to generate error messages if needed.

5 Transforming XML-documents with XSL

The main purpose of our task was to produce an XHTML document from an XML-document with the DTD declaration with help of XSL Transformation using the XPath in it. The XML-document was not designed to do anything, just to carry data. The XSL part is necessary to specify the collection of the elements from the XML part and the transport of them into the XHTML document. The collection is determined by the different XSL elements. The only XSL element which extracts the value of the selected XML node is `<xsl:value-of>`. The other XSL elements return an XHTML tag or an XHTML text or they define which node we are interested in. These nodes are determined by the conditions given in the *expression* or *pattern* in the attributes of XSL elements.
First we classify the XSL elements. We make two groups from them. The first group comprise the simply elements such as "element", "attribute" and "text". We can create the XHTML nodes from these XSL nodes, there is nothing to do with the XML part. Let us call this group "XSL $\rightarrow$ XHTML group". The other group stands for the case when we have to check also the XML part. Let us call this group "XSL $\rightarrow$ XML $\rightarrow$ XHTML group". We can divide this group into three subgroups according to the attributes of the XSL elements:

- *select*: "apply-templates", "for-each", "sort", "value-of", where there is an `expression` which according to we are looking for the properly elements in the XML-document. This `expression` can be an XPath or just an element name. We have the possibility to choose just one element but also all of the elements. With "value-of" we choose just the first XML element which tag name is identical with the `expression` of the attribute of the XSL element. We can also use iteration if we want to list more XML elements. Then it is necessary to apply an iterating XSL element, e.g. the "for-each" element. It contains an element name which is identical with the name of a given XML tag. For every such tag we apply the content of the "for-each" element.

- *test*: "when" in the "choose" element, "if", where there is a `boolean-expression`. This boolean-expression has to be tested. The value of the required `test` attribute contains the expression to be evaluated. There are lots of forms for testing but we focus on just one of them, namely the relation "=" (equal). We test if a given identifier with its value also occurs in the XML part. If yes, then we pick it up, if not, then we do not deal with it. These XSL elements are also iterating elements.

- *match*: "template", where there is a pattern matching. This is also a kind of iterating XSL element. For every XML elements where the pattern matching was fulfilled, we apply the XSL elements that are included by the "template" element.

Now we summarize the steps of the algorithm. We consider the XSL document, go through it and write the value of every attribute "name" or the text in the case of "text" of the elements of the "XSL → XHTML group" directly to the XHTML document. In the implementation there is specified an attribute `en_type_xsl` and an attribute `en_xsltemp`. The type of these attributes are `XmlRoot` and `XslRoot`, respectively. With the attribute `en_type_xsl` we give the whole XML tree to every node of the XSL tree. It means we have all information at every nodes about the XML part. While we are going through the XSL part, and we find an XSL element of the "XSL → XML → XHTML group" then we go through the XML part of the document. With the attribute `en_xsltemp` we give the whole XSL tree to some XSL nodes which are related to the XSL element `<xsl:apply-templates>`. With different functions we are looking for the proper subtree for the `identifier` occurs in the XSL element `<xsl:apply-templates>`. There is a function, namely `makeTransferforTemplates`, detailed in Chapter 6.3, which has these two attributes and an XSL template node as parameters. By using this function we make pattern matching. If there is one or more pattern matching we collect every proper element from the XML part according to the XSL part and these elements will be transferred correctly into the XHTML document.
As result we get a complete and correct XHTML document.

6 Specification of the XML language in DSL

The purpose of the specification is to serve as an input for a generator producing a language-based editor for XML. Since the specification language DSL is used, the specification for XML must be an element of DSL. The editor provides error messages if needed to preserve the restrictions of our XML-documents.

6.1 Structure of the specification

The specification is done in several steps and every step is presented in detail in the next subsections:

- Abstract syntax: specifies the underlying structure of XML-documents including an inline DTD and an optional XSL part by means of a ranked sorted alphabet related to a context-free grammar defining the context-free syntax of XML-documents.
- Attribution: specifies the context-sensitive restrictions of XML-documents by means of the macro attribute grammar formalism combined with an extended monadic second-order logic over trees.
- Logic Description of Node Properties: specifies the context-sensitive restrictions of XML-documents by means of the monadic second-order logic combined with an extended macro attribute grammar.
- Unparsing: specifies the external textual representation of the edited object by means of a transformation of its internal representation as attributed abstract syntax tree into a text consisting of terminal symbols of the context-free grammar of XML-documents augmented by information about the compliance of the context-sensitive syntax of XML-documents.
- Structural Transformations: specifies the possible transformations of the edited object in the structural editing mode of the language-based editor for XML-documents.
- Parsing: specifies the concrete input syntax of XML-documents, i.e., the transformation of inputed text during the textual editing mode of the language-based editor for XML-documents into part of the abstract syntax tree internally representing the edited object.
- Auxiliary Functions: have the purpose of aiding the specification of the last five items.

In addition we try to show the specification steps and aspects on a running example in every chapter, on pieces of the DSL implementation, figures which will show for example the internal representations, and transformations done on the abstract syntax trees.

6.2 Abstract Syntax

In order to specify the underlying structure of XML-documents we use a context-free grammar. Taking such a grammar, abstracting from the terminal symbols, and giving the productions unique names, we get the definition of the abstract syntax of XML-documents. The object edited by the language-based editor is internally represented by an abstract syntax tree which is built up representing this abstract syntax. Consider the following production of the context-free grammar in our case:

$$\begin{array}{lll}
1.\ \mathit{DtdXmlXsl} & \rightarrow & \mathit{DtdRoot\ XmlRoot\ XslRoot} \\
 & \rightarrow & \mathit{DtdRoot\ XmlRoot\ XslRoot\ XhtmlRoot} \\
2.\ \mathit{DtdRoot} & \rightarrow & ''<?xmlversion = ''1.0''encoding = UTF - 8? > '' \\
 & & ''<!DOCTYPE''\ \mathit{Identifier}\ ''['' \\
 & & \mathit{EntListOpt\ ElemList\ AttrListOpt}\ '']>''
\end{array}$$

By omitting the terminal symbols and giving the production from 1. the name DTDXMLXSL and XHTML and from 2. DTD, we get a rule of a related regular tree grammar defining the abstract syntax of the above mentioned example:

$$\begin{array}{lll}
1.\ \mathit{DtdXmlXsl} & = & \text{DTDXMLXSL}\ \mathit{DtdRoot\ XmlRoot\ XslRoot} \\
 & | & \text{XHTML}\ \mathit{DtdRoot\ XmlRoot\ XslRoot\ XhtmlRoot} \\
2.\ \mathit{DtdRoot} & = & \text{DTD}\ \mathit{Identifier\ EntListOpt\ ElemList\ AttrListOpt}
\end{array}$$

In the 1. case in the initial state we do not have the *XhtmlRoot* because we create it with the transformation. The *XhtmlRoot* is a new successor for the constructor *XHTML*. For the definition of the abstract syntax of the XML-document, DSL has the construct of a *data block*. A data block consists of its name and a list of data definitions, specifying the rules of a regular tree grammar defining the abstract syntax, encapsulated by the keywords *begindata* and *enddata*.
The data block defining the abstract syntax of the XML-document must contain the definition of a special algebraic type, i.e. the root type. This type has only one constructor and is not allowed in a successor position of any other constructor defined in the data block. The selection of the root type is not done within a data block.

In a data definition the first constructor is called default constructor. This internal representation is done and initialized by starting with the unique constructor of the root type and then recursively instantiating every successor type by its default constructor, so the order of the constructors in a data definition becomes of importance. The specification has to keep in front to avoid neither infinite cycles nor non-algebraic data types occur in this process. The initial abstract tree of the whole XML-document is shown in Figure 1.
The specification of the abstract syntax of DTD, XML and XSL in DSL:

```
begindata DtdXmlXsl

data Root
  = ROOT DtdXmlXsl

data DtdXmlXsl
  = DTDXMLXSL DtdRoot XmlRoot XslRoot
  | XHTML DtdRoot XmlRoot XslRoot XhtmlRoot

data DtdRoot
  = DTD Identifier EntListOpt ElemList AttrListOpt

data XmlRoot
  = XMLROOTNODE Identifier XmlAttrListOpt XmlText XmlNodeList

data XmlAttrListOpt
  = XMLATTRLISTOPTNIL
  | XMLATTRLISTOPT XmlAttrList

data XmlAttrList
  = XMLATTRSINGLE Identifier QuotedString
  | XMLATTRPAIR Identifier QuotedString XmlAttrList

data XmlNodeList
  = XMLNODENIL
  | XMLNODEPAIR XmlNode XmlNodeList

data XmlNode
  = XMLNODEUNSPEC
  | XMLNODECLOSED Identifier XmlAttrList
  | XMLNODEOPEN Identifier XmlAttrListOpt XmlText XmlNodeList

data XmlText
  = XMLTEXTNIL
  | XMLTEXT NString XmlText
  | XMLENTITY Identifier

data EntListOpt
  = ENTLISTOPTNIL
  | ENTLISTOPT EntList

data EntList
  = ENTSINGLE Ent
  | ENTPAIR Ent EntList

data Ent
```

```
  = ENT Identifier IntExt

data ElemList
  = ELEMSINGLE Elem
  | ELEMPAIR Elem ElemList

data Elem
  = ELEM Identifier TagStruc

data XslRoot
  = XSLROOT XslTemplateList

data XslTemplateList
  = XSLTEMPLATESINGLE XslTemplate
  | XSLTEMPLATEPAIR XslTemplate XslTemplateList

data XslTemplate
  = XSLTEMPLATE XslTemplateAttrMatch XslNodeListM

data Identifier
  = IDENTIFIERNULL
  | IDENTIFIER String

data NString
  = NSTRINGNULL
  | NSTRING String
...
enddata
```

Now we will discuss some of the constructors and their purposes.
The editor should be able to translate the edited program into XHTML code for all implementations. The textual outputs of this editor are a XML-document or an XHTML code. The constructor DTDXMLXSL refers to the textual output that will be displayed from the XML-document represented by an abstract syntax tree, and the constructor XHTML refers to the textual output that will be displayed from the XHTML code represented by an abstract syntax tree.
In the data type DtdXmlXsl the constructor DTDXMLXSL refers to the XML-document and the constructor XHTML refers to the XHTML code.
Since the data type `DtdXmlXsl` consists of two constructors, we can not use the data type as the root type although the data type is related to the start symbol in the context-free syntax of the whole XML-document. That does not fulfill the restriction of the root type that should consist of one constructor. As the root type we create a new data type root. The data type only consists of one constructor ROOT. The successor type of the constructor is `DtdXmlXsl`. The definition of the root type as follows:

```
  data Root              = ROOT DtdXmlXsl
```

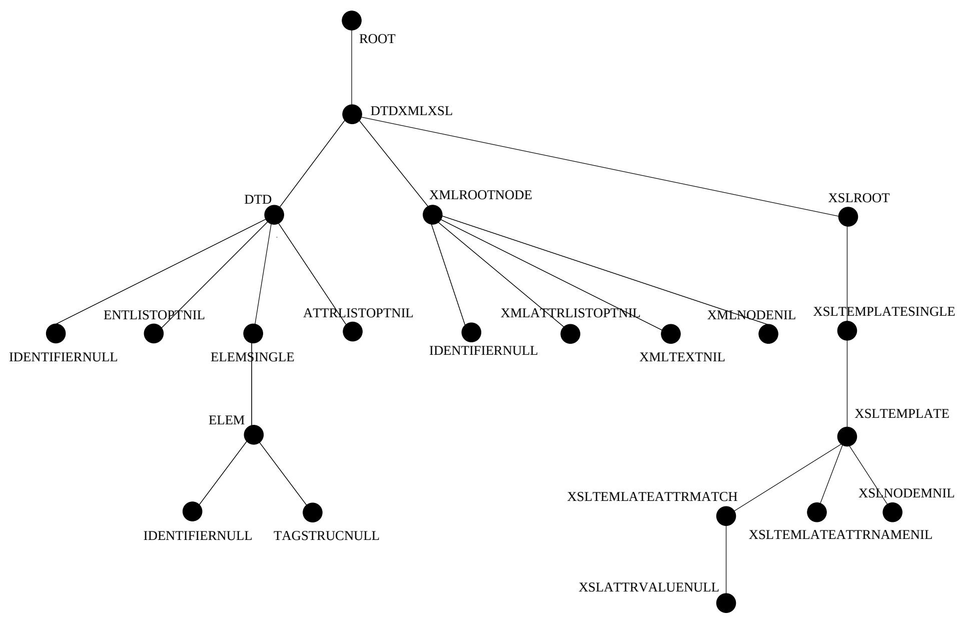


Figure 1: Initial syntax tree

There are some new placeholder constructors which are defined to represent the unspecified part of the document. Their names usually end with suffix NIL or NULL. It makes the constructors as default constructors. The initial abstract syntax tree will be constructed with these constructors.
Some placeholder constructors are needed in the transformation that replaces a subtree by another subtree. The replacing subtree usually contains also some placeholder constructors. As an example let us see a small part of the abstract syntax of DTD:

```
data EntListOpt       = ENTLISTOPTNIL
                      | ENTLISTOPT EntList
data EntList          = ENTSINGLE Ent
                      | ENTPAIR Ent EntList
data Ent              = ENT Identifier QuotedString
```

The list of entities is optional, that is why we have the placeholder constructor `ENTLISTOPTNIL`. We can replace this by the subtree `ENTLISTOPT`. In this case, we need one more placeholder constructor for the data type `EntList`. `EntList` is a list of entities that are composed of the pair `Identifier` and `QuotedString`, where both `Identifier` and `QuotedString` are strings.

6.3 Attribution

Here we have to specify the context-sensitive syntax of XML by using a macro attribute grammar formalism combined with an extended monadic second-order logic over trees. The

attribution is defined in a Mag Block. With this attribution we can transfer information over the abstract syntax tree that is defined in the abstract syntax. The transfer of the information which can be found in the tree and can be used at another place in the tree can be implemented by means of inherited and synthesized attributes.
In our task, the attribution is applied together with logic description of node properties in checking of context-sensitive restrictions.
Many attributes to the abstract syntax tree are related with the context sensitive restrictions for an XML-document. In Chapter 3.7 we see the context-sensitive restrictions. To check if these restrictions are fulfilled by an XML-document, they should refer to other parts of the document.
First of all we have to identify the context sensitive restrictions that should be fulfilled in a part of the XML-document. Then we have to determine the needed information and in which part the information is available or can be generated. Then we have to determine according to the abstract-syntax tree how the information can be transferred. We will show the attribution for some context-sensitive restrictions. In attributions, we only discuss the main attributes. We omit some supporting attributes such that we can concern on the main idea of the attributions. We summarize the properties of several structural attributes. We describe where the given attribute occurs in the XML-documents, in which direction it transports the information, the type of the represented information and the aim of the attribute.

Structural attributes:

Name of the attribute: *name*

- **occurrence:**
  in the whole XML-document on identifier `a`
- **direction of information transport:**
  synthesized
- **kind of represented information:**
  It returns a String for the given identifier.
  String
- **aim:**
  For the identifier it returns the name.

Name of the attribute: *first*

- **occurrence:**
  in the DTD part on elements
- **direction of information transport:**
  synthesized
- **kind of represented information:**
  It returns a boolean value for the given element.
  Bool

- **aim:**
  It tests if the element is the first declared (root) element in the DTD.

Name of the attribute: *decl_type*

- **occurrence:**
  in the DTD part on elements or on list of elements
- **direction of information transport:**
  synthesized
- **kind of represented information:**
  It returns if the given element is declared and the structure of the declaration for the given element.
  Identifier → (Bool,TagStruc)
- **aim:**
  It sums and transports the information about the given element if it is declared. It provides the information for the attribute *en_type*.

Name of the attribute: *en_type*

- **occurrence:**
  in the whole XML-document
- **direction of information transport:**
  inherited
- **kind of represented information:**
  It returns if the given element is declared and the structure of the declaration for the given element.
  Identifier → (Bool,TagStruc)
- **aim:**
  It carries the same information as the attribute *decl_type*. It is used by declaration tests in the XML and XSL parts; and by the nesting check in the XML part.

Name of the attribute: *decl_type_attr*

- **occurrence:**
  in the DTD part on attributes or on list of attributes
- **direction of information transport:**
  synthesized
- **kind of represented information:**
  It returns the if for the given element whether there exists an attribute declaration and how the declaration looks like.
  Identifier → (Bool,AttrElems)

- **aim:**
  It sums and transports the information about a given element it has an attribute declaration and if it is the case how it looks like. It provides the information for the attribute *en_type_attr*.

Name of the attribute: *en_type_attr*

- **occurrence:**
  in the whole XML-document
- **direction of information transport:**
  inherited
- **kind of represented information:**
  It returns the if for the given element whether there exists an attribute declaration and how the declaration looks like.
  Identifier → (Bool,AttrElems)
- **aim:**
  It is defined by the attribute *decl_type_attr*. It is used by the testing if a given XML node owns the required attributes.

Name of the attribute: *decl_type_ent*

- **occurrence:**
  in the DTD part on entity or on list of entities
- **direction of information transport:**
  synthesized
- **kind of represented information:**
  It returns a boolean expression if the given entity is declared.
  Identifier → Bool
- **aim:**
  It transports and sums the information if a given entity is declared in the DTD part. It gives the information for the attribute *en_type_ent*.

Name of the attribute: *en_type_ent*

- **occurrence:**
  in the whole XML-document
- **direction of information transport:**
  inherited
- **kind of represented information:**
  It returns a boolean expression if the given entity is declared.
  Identifier → Bool

- **aim:**
  It is defined by the attribute *decl_type_ent*. It is used by the testing if a given XML node owns the required entities.

Name of the attribute: *en_type_xsl*

- **occurrence:**
  in XSL part
- **direction of information transport:**
  inherited
- **kind of represented information:**
  It returns the XML tree.
  XmlRoot
- **aim:**
  It gives the whole XML tree to every XSL node. It is used by the attribute *make_xhtml*. With the help of auxiliary functions it collects all the information from the XML part according to the XSL part to make the XHTML code.

Name of the attribute: *make_xhtml*

- **occurrence:**
  in XSL part
- **direction of information transport:**
  synthesized
- **kind of represented information:**
  It returns the XHTML tree.
  XhtmlRoot
- **aim:**
  With this attribution we construct the information about the created XHTML nodes.

6.4 Logic Description of Node Properties

Here we are able to specify the context-sensitive syntax of XML-documents by means of a set of MSO-formulas. Moreover, it is possible to combine the specification formalisms of macro attribute grammars and monadic second-order logic to use the advantages of both paradigms. With MSO-formulas we can define so-called node properties with respect to the considered abstract syntax tree. By means of these node properties the context-sensitive syntax of XML can be specified in a logical and therefore more global way. These properties of concrete nodes in the abstract syntax tree can be used to produce error messages depending on the violation of the context-sensitive syntax of XML in the edited object. This is possible because of the combination of the mso formulas with macro attributes, which

generate the error massages and present them to the user of the language-based editor after each program modification.
Now we summarize the properties of the logical attributes.

Logical attributes:

Name of the attribute: *?elem_declared*

- **occurrence:**
  in DTD part on attributes
- **aim:**
  It tests if the name of a given attribute is declared or not.

Name of the attribute: *?def_elem_id*

- **occurrence:**
  in DTD part on attributes
- **aim:**
  It tests if the corresponding element is declared. It is used by the attribute ?*elem_declared.*

Name of the attribute: *?elem_used*

- **occurrence:**
  in DTD part on elements
- **aim:**
  It checks if a given element is used in some element declaration. It uses the the attribute ?*first_one* and ?*used_elem_id* attributes.

Name of the attribute: *?first_one*

- **occurrence:**
  in DTD part on elements
- **aim:**
  It tests if the given element is the first element. It is used by the attribute ?*elem_used.*

Name of the attribute: *?used_elem_id*

- **occurrence:**
  in DTD part on elements
- **aim:**
  It tests if the given element is situated in the structure of some element declaration. It is used by the attribute ?*elem_used.*

Name of the attribute: *?elem_equal_doc*

- **occurrence:**
  in DTD part
- **aim:**
  It tests if the root tag of the XML-document is first declared in the DTD. It uses the the attribute ?*in_root* and the attribute ?*in_doc* attributions.

Name of the attribute: *?in_root*

- **occurrence:**
  in XML part in the rootnode
- **aim:**
  It tests if the given element is the root element. It is used by the attribute ?*elem_equal_root* and the attribute ?*elem_equal_doc*.

Name of the attribute: *?in_doc*

- **occurrence:**
  in XML part in the rootnode
- **aim:**
  It tests if the given element is situated in the DTD declaration part. It is used by the attribute ?*elem_equal_doc*.

Name of the attribute: *?elem_equal_root*

- **occurrence:**
  in DTD part
- **aim:**
  It tests if the root tag of the XML-document is first declared in the DTD. It uses the the attribute ?*in_root* and the attribute ?$first\_one$ attributions.

Name of the attribute: *?typeAltList*

- **occurrence:**
  in DTD part in the attributes on datatypes
- **aim:**
  It tests if the value of the datatype of the attribute is valid declared. It uses the the attribute ?*in_typeAltList* attribution.

Name of the attribute: *?in_typeAltList*

- **occurrence:**
  in DTD part in the alternative of datatype of the attributes
- **aim:**
  It tests if a given element is in the alternative of datatypes. It is used by the attribute ?*typeAltList*.

Name of the attribute: *?elem_unique_declared*

- **occurrence:**
  in DTD part at elements
- **aim:**
  It tests if a given element is unique declared.

Name of the attribute: *?in_declaration*

- **occurrence:**
  in DTD part at elements-identifiers
- **aim:**
  It tests if a given identifier is situated in the element declaration part. It is used by the attribute ?*elem_unique_declared*.

Name of the attribute: *?ent_unique_declared*

- **occurrence:**
  in DTD part on elements
- **aim:**
  It tests if a given entity is unique declared.

Name of the attribute: *?in_ent_declaration*

- **occurrence:**
  in DTD part on element-identifiers
- **aim:**
  It tests if a given identifier is declared as an entity. It is used by the attribute ?*ent_unique_declared*.

Name of the attribute: *?attrlist_unique_declared*

- **occurrence:**
  in DTD part on elements
- **aim:**
  It tests if a given attribute is unique declared.

Name of the attribute: *?in_attr_declaration*

- **occurrence:**
  in DTD part on element-identifiers
- **aim:**
  It tests if for a given identifier there is an attribute declaration. It is used by the attribute ?*attr_unique_declared*.

Let us see some example for logical attribtes.
Example 1: DTD - element declarations:
In the DTD, XML elements are declared with an element declaration. An element declaration has the following syntax:

```
<!ELEMENT element-name element-content>
```

All `element-names` occurring on the left-hand side with exception of the root element (in this case `Faculty`) has to be used as some child element, as shown in the next example, if not the editor has to deliver an error message to the specifier that this context-sensitive restriction of the XML-document is violated. We combine the specification formalisms of macro attribute grammars and monadic second-order logic and use it in the unparsing to generate the error message if needed.

```
<!ELEMENT Faculty (Facultyname, Facultyaddress)>
<!ELEMENT Facultyname    (#PCDATA)>
<!ELEMENT Facultyaddress (#PCDATA)>
```

The attribution:

```
beginmag CSR_Attr

  syn name :: Identifier -> String

  t@(IDENTIFIERNULL) in
    name t = ""
  t@(IDENTIFIER t1) in
    name t = t1

endmag
```

MSO formulas:

```
beginmso CSR_Logic

  formula ?elem_used    :: Identifier -> Bool
  formula ?used_elem_id :: Identifier -> Bool
  formula ?first_one    :: Identifier -> Bool

  ?elem_used x
         = Or (?first_one x)
             (Or (Label IDENTIFIERNULL x)
                 (ExNode y:: Identifier
                     (And (?used_elem_id y)
                         (Mag (equal_name (name y) (name x))))))
```

```
   ?first_one x
          = Parent x ELEM Node y
              (Mag (first y))

   ?used_elem_id x
          = Parent x COS_NAME Node y
              (True)

 endmso
```

Unparsing:

```
beginmag Unparse_Mag

  ...
  up (ELEM t1 t2) =
       "<!ELEMENT   " ++ (up t1) ++
           (check " { element not used } " (?elem_used t1)) ++
       "      " ++ (up t2) ++ ">"
  ...

endmag
```

Auxiliary functions:

```
equal_name :: String -> String -> Bool
equal_name string1 string2 =  string1 == string2

check :: String -> Bool -> String
check message condition
           = if condition then "" else message
```

Example 2: DTD - attribute declarations:
An attribute declaration has the following syntax:

```
<!ATTLIST element-name
   attribute-name1 attribute-type1 default-value1>
   attribute-name2 attribute-type2 default-value2 ...>
```

If we define attributes of an element (`element-name`) we have to assure that the element is already declared like in the next example.

```
<!ELEMENT Student (Name, Semester)>
...
<!ATTLIST Student immatriculation ID #REQUIRED
         ... >
```

For this, we combine the specification formalisms of macro attribute grammars and monadic second-order logic, and we use those in the unparsing part to deliver an error message to the specifier if needed.

MSO formulas:

```
beginmso CSR_Logic

  formula ?elem_declared :: Identifier -> Bool
  formula ?def_elem_id   :: Identifier -> Bool

  ?elem_declared x
        = Or (Label IDENTIFIERNULL x)
             (ExNode y:: Identifier
                  (And (?def_elem_id y)
                       (Mag (equal_name (name y) (name x)))))

  ?def_elem_id x
        = ExNode y :: Elem
            (Edge y 1 x)

 endmso
```

Unparsing:

```
beginmag Unparse_Mag

  ...
  up (ATTR t1 t2) =
       "<!ATTLIST " ++ (up t1) ++
           (check " { Element not declared } "
             (?elem_declared t1))
       ++ "\t\n" ++ (up t2) ++ "\b" ++ ">"
  ...

endmag
```

6.5 Unparsing

Here we define how the edited object should be textually represented to the user. Because the edited object is internally represented by an abstract syntax tree during the editing process we need to transform this in an external representation, and we have to deal with error-messages concerning violations of context-sensitive syntax of XML-documents. The Figure 4 shows the initial syntax trees external representation. The needed transformation is specified as a recursive function over the abstract syntax tree, i.e. the algebraic types defined in the data block specifying the abstract syntax. This function has the result type String and is called unparse function. The unparse function can be seen as special synthesized attribute. Because of this it is specified in a mag block in the same way as a synthesized attribute respecting some more restrictions.The unparsing also translates the abstract syntax tree into another one for implementation of XHTML. We discuss both textual representations in the following subsections.

6.5.1 Displaying XML-documents

Here we translate an abstract syntax tree into an XML-document. The synthesized attribute is a plain text consisting of terminal symbols of the context-free syntax of XML-documents and error messages according to the context-sensitive restrictions of XML-documents. The plain text consists of the error messages if some context-sensitive restrictions are not fulfilled. The error messages will appear whenever the user of this editor edits an XML-document.
The following specification of our unparse function transforms the initial syntax tree (Figure 1) in the external representation (Figure 4), and is a part of the unparse function definition for XML-document with an inline DTD.

```
beginmag DtdXmlXsl_Unparse_Mag

unparse up :: a -> String where a from
     {DtdRoot, XmlRoot, Identifier, EntListOpt, ElemList,
      AttrListOpt, XmlAttrListOpt, XmlText, XmlNodeList, ...}

up (DTDXMLXSL t1 t2 t3)
     = "<?xml version = \"1.0\" encoding = UTF-8?>" ++ "\n"
       ++ (up t1) ++ "\n\n" ++ (up t2) ++ "\n\n" ++ (up t3)
up (XHTML t1 t2 t3 t4)
     = (uph t4)
t@(XMLROOTNODE t1 t2 t3 t4) in
up t = "<" ++ (up t1) ++ (check " {Root TAG not declared} "
                              (?elem_equal_root t1))
       ++ (check ("{Nesting error}")
                   (match (makeAlphabet (en_type t t1) t4)
                          (xmlRootNodeToWord t)
                          (tagStrucToRegExp (en_type t t1)))
       ...
       ++ (up t2) ++ ">" ++ (up t3) ++ "\t\n" ++ (up t4)
       ++ "\b\n" ++ "</" ++ (up t1) ++ ">"
up (XMLROOTNODEHIDE t1 t2 t3 t4)
     = "XML"
up (XMLATTRLISTOPTNIL)
     = "_attr_"
up (XMLNODENIL)
     = ""
up (XMLTEXTNIL)
     = "_text_"
up (DTD t1 t2 t3 t4)
     = "<!DOCTYPE  " ++ (up t1)
       ++ (check " {Declaration Error} "
             (?elem_equal_doc t1))
       ++ "   [" ++ "\t\n" ++ (up t2) ++ "\n" ++ (up t3)
       ++ "\t\n" ++ (up t4) ++ "\b\b\n" ++ "]>"
```

```
up (DTDHIDE t1 t2 t3 t4)
     = "DTD"
up (ENTLISTOPTNIL)
     = ""
up (ELEMSINGLE t1)
     = (up t1)
up (ELEM t1 t2)
     = "<!ELEMENT   " ++ (up t1)
       ++ (check " { element not used } " (?elem_used t1))
       ++ "      " ++ (up t2) ++ ">"
up (ATTRLISTOPTNIL)
     = ""
up TAGSTRUCNULL
     = "_content-type_"
up IDENTIFIERNULL
     = "_identifier_"
...
endmag
```

Note, how the error messages are generated, for example the message `"{Nesting error}"` is generated at the `XMLROOTNODE` which we discussed in Chapter 4.3.

We notice, that there is a so called `HIDE` constructor for the DTD, XML and XSL part, respectively. In the example above we present the `XMLROOTNODEHIDE` and `DTDHIDE` constructors. The main purpose of these constructors is to hide some part of the XML-document. Some advantages of hiding parts of the documents are the following:

- We can concentrate on the actually edited part of the document.
- It raises the performance that the hidden part of the document is not considered. From this part there is no function call or there are no attribution applications performed for this hidden part, but one can get information from this part.
- We do not have to scroll the screen.

6.5.2 Displaying XHTML code

This translation is from an abstract syntax tree into an XHTML code. We need only one synthesized attribute to translate an abstract syntax tree. The name of the synthesized attribute is *uph*. The definition of the attribute as follows:

```
unparse uph :: a -> String
```

The synthesized attribute `uph` is also a plain text with the difference from the synthesized attribute `up` that it does not integrate error messages. In the previous section we considered the translation of the `XMLROOTNODE` with the synthesized attribute `up`. The translation of the `XMLROOTNODE` with the synthesized attribute `uph` is the following:

```
uph (XHTMLROOTNODE t1 t2 t3 t4) = (uph t4)
```

As another example we take a look at following rule:

```
up (XHTML t1 t2 t3 t4) = (uph t4)
```

The synthesized attribute `'up'` calls the other synthesized attribute `'uph'` for the last parameter, i.e. it generates the XHTML part of the document. We do not deal with the DTD, XML and XSL part of the document we just want to represent the XHTML code as result of the XML-document. The information that is represented are collected during the transformation.

6.6 Structural Transformations

In this step of the specification we have to define the structural transformations which are based on the abstract syntax of the language XML. The definitions of the transform function enables thereby structural editing in the generated language-based editor for XML. In this mode of editing the abstract syntax tree of the edited object is directly transformed. This is done by replacing a selected subtree by a new subtree which is computed from the replaced one like showed in Figure 2. The transform function specifies this computation. During the editing process of the language-based editor there is always a subtree of the abstract syntax tree selected by the user. The root symbol of the currently selected subtree determines the set of rule definitions of the transform function applicable at this time. The equality of the type of the node argument and the result type of the transform function assure that the subtree is only replaced by a tree of the same type.
Example:

```
t@(XMLNODEUNSPEC) in
  trans t InsClosed
    = XMLNODECLOSED
        IDENTIFIERNULL
        (XMLATTRSINGLE IDENTIFIERNULL STRINGVALUENULL)
  trans t InsOpen
    = XMLNODEOPEN
        IDENTIFIERNULL
        XMLATTRLISTOPTNIL
        XMLTEXTNIL
        XMLNODENIL
```

The transition function defined above will be active if the user is actually editing the abstract syntax tree at the subtree root `XMLNODEUNSPEC`. At this position he has the choice to transform the unspecified XML tag into a closed or an opened tag. The editor deals with buttons. Pressing one of the two bottoms the subtree with the root `XMLNODEUNSPEC` will be replaced with one of the right hand sides. Depending on the choice of the user, the transformation of the internal representation in the syntax tree is showed in Figure 3.
For a subtree representing a list, the transformations are *insert before*, *insert after*, *cut first element*, and *cut rest of list* that refer to insertion and deletion in a list. As examples are transformations for a list of constructors. The names of the transformations are `InsAfter`, `InsBefore`, `CutFirst`, and `CutRest`. Now let us see an example for an XSL transformation:

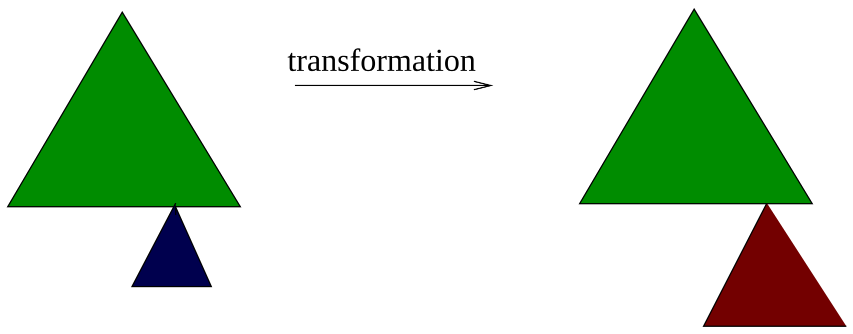


Figure 2: Transformation of one subtree into another subtree (the blue colored subtree will be transformed into the red colored one)

```
t@(XSLNODEWHENSINGLE t1) in
  trans t InsBefore = XSLNODEWHENPAIR
                          (XSLNODEWHEN EXPRESSIONNULL
                                       XSLNODEMNIL)
                          t
  trans t InsAfter = XSLNODEWHENPAIR
                         t1
                         (XSLNODEWHENSINGLE
                           (XSLNODEWHEN EXPRESSIONNULL
                                        XSLNODEMNIL))

t@(XSLNODEWHENPAIR t1 t2) in
  trans t InsBefore = XSLNODEWHENPAIR
                         (XSLNODEWHEN EXPRESSIONNULL
                                      XSLNODEMNIL)
                         t
  trans t CutFirst = t2
  trans t CutRest = XSLNODEWHENSINGLE t1
```

The transition function defined above will be active if the user is actually editing the abstract syntax tree at the subtree root XSLNODELISTLESSUNSPEC. At this position the user can choose one of the possibilities. After pressing one of the buttons, the subtree with the root XSLNODELISTLESSUNSPEC will be replaced with one of the right hand sides.

Other transformations are also changing the label of root of subtree. Examples of the transformations are transformations for a subtree with type DtdXmlXsl. The possible labels for the root of the subtree are DTDXMLXSL and XHTML. The meaning of the labels is explained in Chapter 6.2. We define transformations `Trans_to_XML` and `Trans_to_XHTML` to change the label with DTDXMLXSL and XHTML, respectively.

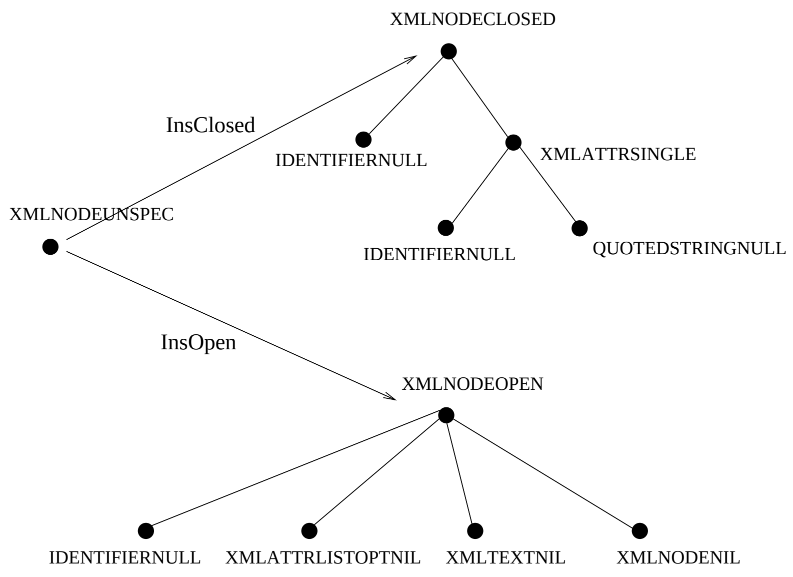


Figure 3: Transformation of an unspecified XML node into a specific one

```
t@(DTDXMLXSL t1 t2 t3) in
  trans t Trans_to_XHTML = XHTML t1 t2 t3 (make_xhtml t3)
t@(XHTML t1 t2 t3 t4) in
  trans t Trans_to_XML = DTDXMLXSL t1 t2 t3
```

With the transformation we create a new subtree with the XHTML elements. With the attribute `en_type_xsl` we transfer the whole XML tree to every node of the XSL tree. We use auxiliary functions, detailed in Chapter 6.8 to create the XHTML nodes for every XSL node. Then with the attribution `make_xhtml` we construct the information about the created XHTML nodes.

6.7 Parsing

In this part, we can specify textual inputs in the language-based editor. The textual inputs can be some part of the language. To do this, we need the parsing. In the editor the parsing will transform a textual input into a part of the abstract syntax tree of the edited object in the language-based editor. To make textual input possible in the editor, three blocks are defined in the specification for three different purposes. These three blocks are described in the next three subsections.

6.7.1 Lexical Analysis

This block defines the scanner by the list of tokens specified by means of regular expressions. There are two type of tokens:

- non-value-capturing tokens just matching a text - are used as keywords, as shown below, it just matches the string "`#PCDATA`".

  ```
  token PcdataT = "#PCDATA"
  ```

- value-capturing tokens which matches a text and stores a value of the specified type, as shown below, it matches the regular expression in the right hand side.

  ```
  token IdentifierT String   = [a-zA-Z] [0-9a-zA-Z_]*
  ```

The order of the token definitions are important for the process of scanning. Some token definitions of the specification of the editor for XML-documents are listed below.

```
beginlex Text_Input
     ...
     token PlusT                = "+"
     token PclosedT             = ")"
     token AtT                  = "@"
     token AssignT              = "="
     token AnyT                 = "ANY"
     token PcdataT              = "#PCDATA"
     token NmtokenT             = "NMTOKEN"
     token IdentifierT String   = [a-zA-Z] [0-9a-zA-Z_]*
     token QuotedStringT String = '"' [0-9a-zA-Z_\&;:,.]+ '"'
     ...
endlex
```

6.7.2 Concrete Input Syntax

This section describes the specification for parsing the textual input with the help of the tokens defined in the lex block. This specification is in DSL construct called parse block. The parser is defined for the context-free grammar by the list of parse definitions. A parse definition represents a set of productions of this context-free grammar. For example, a parse definition of type `XmlAttrList`:

```
beginparse Text_Input
    ...
    parse P_XmlAttrList :: XmlAttrList
        | P_Identifier
          AssignT
          P_QuotedString    = XMLATTRSINGLE $1 $3
        | P_Identifier
          AssignT
          P_QuotedString
          P_XmlAttrList     = XMLATTRPAIR $1 $3 $4
    ...
endparse
```

A parse definition begins with a keyword `parse` followed by a non-terminal symbol i.e. `P_XmlAttrList` here, which has to be specified as context-free production. `XmlAttrList` is an algebraic data type for the non-terminal symbols `XMLATTRSINGLE` and `XMLATTRPAIR`, which can store a part of input syntax tree. In the above example, in the right hand sides of the definition consists the semantic definition build up by constructors `XMLATTRSINGLE` and `XMLATTRPAIR` of the type `XmlAttrList` together with the keywords `$1` , `$3` and `$4`, which are the number of symbols between `|` and `=`.

6.7.3 Input Determination

In this section, input determination of the parser is specified, i.e. which constructs of the abstract syntax should be textually editable and in which way this should be done. This is done in an input block. It contains a a list of input determinations. The input determination starts with the keyword `input`. It is followed by an algebraic data type of the abstract syntax, followed by the keyword `as`, and followed by the name of parse definition. For example:

```
begininput Text_Input
     ...
     input Identifier    as    P_Identifier
     input NString       as    P_NString
     input QuotedString  as    P_QuotedString
     input XmlAttrList   as    P_XmlAttrList
     input DataType      as    P_DataType
     input DefValue      as    P_DefValue
     ...
endinput
```

6.8 Auxiliary Functions

In this section we present some auxiliary functions that are usable in mag blocks. The auxiliary functions can be defined as normal Haskell functions.
In this specification some auxiliary functions are called in the unparsing, the attribution, and the logic description parts. In this chapter we will discuss some special auxiliary functions which are often called, or have some special role.
One of the functions is `check`. The definition of the function looks like this:

```
check :: String -> Bool -> String
```

The function check returns the first argument if the second argument is false. Otherwise it returns the empty string. The function is useful to generate error messages for a context-sensitive restriction. The error messages and the context-sensitive restriction are the first and the second argument.
The environment of the generator assures for us a match function, which looks like this:

```
match   :: Alphabet -> [Symbol] -> RegExp -> Bool
```

It takes an alphabet, a symbol list and a regular expression and returns a boolean value. So we are able to check if a symbol list(word) over an alphabet is according to the regular expression. This function is used by the validation of XML-document against the DTD. We also need some other auxiliary functions to sum this information, which is needed as argument by the match function.
The next functions are collecting the information from the DTD and XML part of the document.

```
makeAlphabet      :: TagStruc -> XmlNodeList -> [Symbol]

xmlNodeListToWord :: XmlNodeList -> [Symbol]

tagStrucToRegExp  :: TagStruc -> RegExp
```

The role of the match function is discussed also in the Chapter 4.

We consider another function, namely *makeTransferforTemplates*. The definition of this function is the following:

```
makeTransferforTemplates
   :: XslTemplate -> XmlRoot -> XslRoot -> XhtmlNodeList
```

This function takes the first XSL template node, the XML tree, the XSL tree and it gives a list of XHTML nodes. Then we call an another function from it with the following type definition:

```
goList :: XslNodeListM -> XmlRoot -> XslRoot
          -> (XhtmlText, XhtmlNodeList)
```

So, we take the list of the XSL elements and the XML tree to create a pair for the XHTML part. We need the two types `XhtmlText` and `XhtmlNodeList` because of the difference of the XSL elements. Some of them return an XHTML tag without text, and some of them give just an XHTML text back. For every XSL element in this list we call a function:

```
getFirstElem
   :: (XslNodeListMore,XmlRoot) -> (XhtmlText, XhtmlNodeList)
```

It means, we choose the proper XML nodes from the XML tree according to the XSL node, and it returns a pair of XHTML text and list of XHTML nodes. If there is no match then it gives an empty XHTML text and an empty XHTML nodelist. In the function *getFirstElem* we are using several function calls with the same type definition, such as:

```
callElement
  :: XslNodeListMore -> XmlRoot -> XslRoot
     -> (XhtmlText, XhtmlNodeList)

callValueOf
  :: XslNodeListMore -> XmlRoot -> XslRoot
     -> (XhtmlText, XhtmlNodeList)
```

```
callForEach
  :: XslNodeListMore -> XmlRoot -> XslRoot
     -> (XhtmlText, XhtmlNodeList)

...
```

With these functions we distinguish the handle of the XSL elements. As mentioned earlier, there are some cases where we need to create an XHTML tag, for example the `<xsl:element>` element. And there are some cases where we need the text from the XML part, for example the `<xsl:value-of>` element. If we go through on the list of XSL elements, we get a list of XHTML nodes. From this we create the complete XHTML code with help of the following function:

```
makeRootfromNodeList :: XhtmlNodeList -> XhtmlRoot
```

7 How To Use The Editor

The result of this project is the language-based editor for XML-documents with an inline DTD. We will discuss in this section how the editor can be used to help the user to write valid XML-documents.
The specification of XML-documents is saved in the file DtdXmlXsl.dsl, and attached to this document on a CD. This file consists of the specification in DSL that are explained in the previous chapters.
First of all, the editor can be generated from our specification file. We need the generator system designed to generate the editor from a specification file in DSL and run the generator system. The generator system asks for the specification file as input. The result of the process is an executable file Start and its supporting files. We can start the editor from the file Start.

Figure 4 shows a window of the editor. We discuss further the parts of the window.
Menus consist of the following items:

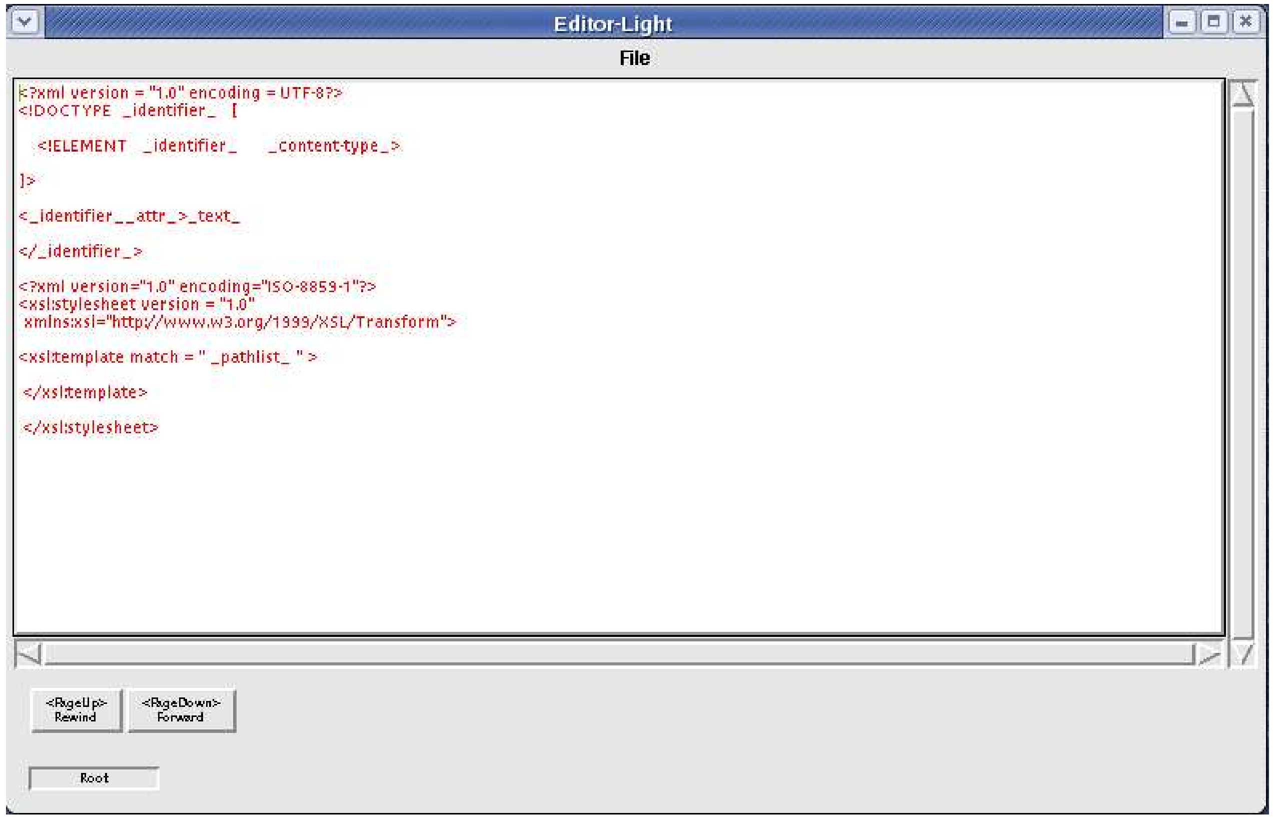
1. New: to create a new file consisting of an XML-document
2. Open: to open a file consisting of an XML-document
3. Save: to save the current XML-document
4. Exit: to exit from the editor

In the editing area we can see the edited XML-document. The current-edited part of the program is shown by red color. We can edit the current-edited part by an applicable transformation or textual input. The applicable transformations are showed in Transformation Button the name of the transformation and a keyboard key. We can choose one transformation by mouse device or the keyboard key. For textual input, we can apply if it is defined as textual input in parsing part of the specification. The input is then given by the keyboard. To delete the whole current editing part we can use the insert key.

The editor displays empty names, for example *_quotedstring_* is to indicate the string which is between quotes. Some of the unspecified objects, which can be entered through textual input mode are listed below.

identifier	: name of the elements
string	: gives the place for the text
content-type	: defines the structure of a given element
xmlnode	: an XML node (i.e. closed and open XML node)
xsl-elements	: the XSL elements (i.e. template, element, value-of, if, for-each,...)
expression	: an XSL expression in some XSL elements (e.g. if, when)

We can change the current-edited part by the so-called change buttons. The change buttons consists of two buttons, i.e. `Rewind` and `Forward`. Regarding to the abstract syntax tree representing the XML-document, they change the current-edited subtree, and are always available. Buttons of different functionality are shown below with few examples.



```
<?xml version = "1.0" encoding = UTF-8?>
<!DOCTYPE  _identifier_  [

  <!ELEMENT   _identifier_     _content-type_>

]>

<_identifier__attr_>_text_

</_identifier_>

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version = "1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match = " _pathlist_ " >

 </xsl:template>

 </xsl:stylesheet>
```

Figure 4: Window of the editor

Ins_X	: inserts an X structure
Ins_X_Before	: inserts an X structure before an existing one
Ins_X_After	: inserts an X structure after an existing one
Cut	: erases the selected object and displays default constructor of selected type
Cut_first	: cuts the first object of selected structure
Cut_rest	: cuts the rest objects from the selected object in a structure
Open	: changes the *closed* XML tag into *open*
Close	: changes the *open* XML tag into *closed*
Flip	: changes the order of the rows in the XML-document
Hide	: hides the assigned part of the document
Show	: appears the hidden part of the document (i.e. some two closed and/or open XML tag)

To change the appearance of the documents, we should change the current edited subtree to a subtree with data type $DtdXmlXsl$. The names of the transformations as follows:

Trans_to_XHTML	: shows the XHTML code
Trans_to_XML	: shows the XML-document

The transformation is written in Chapter 6.6.

8 Conclusion

8.1 Afterword

This project builds a specification for XML-documents with an inline DTD and an XSL part defining a transformation of the XML part into XHTML. This specification will be used by the generator system as input. The specification language used in the specification is DSL. The language-based editor is able to edit such a document that is structurally correct according to the context-free syntax. The editor is also able to give appropriate error messages according to the context-sensitive restrictions.

The main tasks of this project was the validation of the XML part against the DTD part and the transformation of the XML-document into XHTML according to the XSL part. The problem faced by resolving the task was the development of the abstract syntax, the development and implementation of algorithms to validate and transform informations according to the context-sensitive restrictions.

In the implementation we were concentrating on the most important elements of the XML-documents. In the XSL part we did not consider every command, every attributes of the elements and every subelements. XSL is a too wide language that is why we concentrated on the most important elements of it.

It was interesting to implement the different XSL commands. Considering the *open* tag in the XHTML code we can get a text or a subnode. We differ two cases of the XSL elements. At first, the text for the XHTML documents we obtain from the XML part according to the XSL elements. Secondly, the name and the attribute of the XHTML tags we get directly from the XSL elements.
We transport the whole XML part into XSL to collect the elements from the XML part according to the elements of the XSL part. We implemented it with different functions to go through on the XML part.

The XSL part is solved such a way that one can expand it with further XSL elements and subelements or inside the elements with further conditions for collecting the information from the XML part.

8.2 Made experiences on DSL

The design of the abstract syntax is one of the first steps of the specification which has to be done. The decisions made in this step are very important and takes effect later on in the whole implementation progress. Because of this fact this step has to be well planned to come before the problem of redesign of the abstract syntax. To test a given specification all the blocks and specification steps have to be done, so it is possible to test only complete specifications. It means that the wrong, i.e. not well planned decisions can be recognized in latter state of the implementation and the correction of these wrong conceptional decisions takes lot energy of the specifier.

The specification done in DSL is used by the generator system to generate the editor for the specified language L. In this way the editor is able to give error messages if needed if some restriction of the language L is violated. The error messages are in the same color and font as the other unparsed information, i.e. the elements of the language L. Using different colors or fonts could be used to give a better advice for the user of the editor to distinguish the language elements from the error messages. It would be also nice to integrate in the specification formalism DSL the option to define textual representation properties of algebraic data types. For example if we could specify that every statement used in an if or while statement in a specified language L is textually represented in a special color, and all declarations assigning a value to a variable in another one.
It would be also practical, if we could save the source code not just as a "dsl" document but also as other, for example in our case we could create an "xsl" and an "xml" document with an inline "dtd". As result we could get an "html" file.
If we edit a document and it is more than one page, we have to scroll the screen. It would be more use-friendly if it shows the current place where the cursor is.

8.3 Attachment

We attach a CD appendix to the documentation. This appendix encloses some running example in eo and also in xml, xsl and html format. The eo format is a format which represents the actual syntax tree. We attach also the documentation in pdf (DtdXmlXsl.pdf) format, and the specification source code in dsl (DtdXmlXsl.dsl).

References

[1] E. Bormann. Entwurf und Implementierung eines Systems zur Erzeugung syntaxgesteuerter Editoren. Master's Thesis, Dresden University of Technology, 2000.

[2] E. Bormann. Automatic Generation of Language-Based Editors using DSL, 2004. Manuscript

[3] T. Bray, J. Paoli, C.M. Spearberg-McQueen, E.Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) Version 1.1. Available on: http://www.w3.org/TR/xml11, W3C Proposed Recommendation 05 November 2003.

[4] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles. Extensible Stylesheet Language (XSL) Version 1.0. Available on: http://www.w3.org/TR/xsl, W3C Recommendation 15 October 2001

[5] J. Axelsson, B. Epperson, M. Ishikawa, S. McCarron, A. Navarro, S. Pemberton. XHTML TM Version 2.0. Available on: http://www.w3.org/TR/xhtml2, W3C Working Draft 22 July 2004

[6] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers - Principles, Techniques and Tools. *Addison-Wasley, 1986.*

[7] Cheryl M. Hughes. The Web Wizard's Guide to XML. *Pearson Education, 2003*

[8] Neil Bradley. The XSL companion, Second edition. *Addison-Wasley, 2002*

[9] Joseph Fasel, Paul Hudak, and John Peterson. A gentle introduction to Haskell 98. Available on: http://www.haskell.org/tutorial, 1999

[10] Zolán Fülöp and Heiko Vogler. *Syntax-directed semantics - Formal models based on tree transducers.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg, first edition, 1998

[11] A. Kühnemann and Heiko Vogler. Synthesized and inherited functions - a new computational model for syntax-directed semantics. *Acta Inform.*, 31:431-477, 1994.

[12] C. Lescher. Entwurf und Implementierung einer Eingabesprache für ein System zur Erzeugung syntaxgesteuerter Editoren, 1999. Minor thesis, Dresden University of Technology.