Creating a Description Logics Knowledge Base for World-Wide Web Documents

Johan Åberg

Abstract

This thesis concerns structured documents and how to model them using Description Logics. The work is mainly motivated by information retrieval needs from the Internet community but the results can also be used in various document management applications.

In this thesis the term "structured documents" refers to marked-up documents, i.e. documents where the implicit structure has been made explicit using some markup language. The markup language SGML is studied and it is shown how many of the most important aspects of SGML can be modeled using a Description Logic supporting part relationships. The support for part relationships has been vital to model the logical parts of documents.

The main goal of the thesis is to model an amount of SGML functionality sufficient for modeling the HTML document language and HTML document instances and also to implement conversion functions from SGML and SGML instances to their Description Logic representation.

This goal has not quite been accomplished, since some SGML functionality used by the document language HTML could not be modeled using the chosen Description Logic. We show in the thesis though, that HTML documents not using the more complex (from the modeling perspective) markup constructs can be faithfully modeled. On the other hand, the SGML functionality that could be modeled can be used to define and model document languages that are in some aspects more complex than HTML.

Acknowledgements

This thesis has been written at the Laboratory for Intelligent Information Systems (IISLAB) at the Department of Computer and Information Science. IISLAB has provided a very stimulating environment with a lot of arrangements for relaxation. I will never forget the miniature golf competition or our continuous boccia competitions.

I would like to thank all members of IISLAB and in particular my supervisor Patrick Lambrix and the laboratory leader Nahid Shahmehri. Patrick has always taken his time to discuss problems (ranging from Description Logics details to the choice of title for this thesis, or whether to print it in 11 points or 10 points) and he has been constantly encouraging. Nahid has also been constantly encouraging and has given valuable critique on a rehearsal of my presentation of this thesis.

I also want to thank my friends here in Linköping including the members of our former study group and our volleyball team, for taking my mind off my studies.

I would like to include my parents in these acknowledgements and I want to thank them for providing a very stimulating growing environment (probably the best imaginable).

I also express my gratitude towards Tom Jack at Volleyball WorldWide for letting me use one of his web pages as an example in this thesis.

Last but not least I would like to thank Charlotte Carlsson for continuous support and joy.

This thesis was written on a Sun SPARC station 10 and a 17" color monitor. I used the Solaris 2 operating system. The text has been marked up using the LaTeX system.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Background 2

1.3 Purpose 3

1.4 About this document 4

1.4.1 Prerequisites 4

1.4.2 Outline 4

2 Structured Documents and SGML **5**

2.1 Introduction 5

2.2 The structure of SGML 6

2.3 Elements 9

2.4 Attributes 11

2.5 Entities 12

2.6 An example 13

2.7 Additional SGML details 13

2.7.1 Characters 16

2.7.2 Marked sections 16

2.7.3 Included and excluded elements 17

2.7.4 Markup minimization 18

2.7.5 Comments 18

3 Description Logics **21**

3.1 Introduction 21

3.2 FOPC Revisited 22

3.3 Syntax and Semantics 24

3.4 Subsumption 30

3.5 Composite Objects 31

3.6 CLASSIC 35

4 Modeling SGML Documents **39**
4.1 Introduction 39
4.2 Modeling restrictions 41
4.3 Modeling a DTD 43
4.3.1 Element declarations 43
4.3.2 Attribute declarations 50
4.3.3 Entity declarations 55
4.4 Modeling a document instance 55
4.4.1 Element instances 55
4.4.2 Entity references 59
4.5 Examples 59

5 Notes on the Implementation **73**
5.1 Introduction 73
5.2 Modeling DTDs 73
5.2.1 From DTDs to concepts 73
5.2.2 From concepts to DTDs 74
5.3 Modeling document instances 75
5.3.1 From structured documents to individuals 75
5.3.2 From individuals to structured documents 76
5.4 A new user function 76

6 Conclusions and Future Work **79**
6.1 Conclusions 79
6.2 Future work 80

A Modeling HTML **83**
A.1 Introduction 83
A.2 Modeling the HTML DTD 83
A.2.1 The HTML DTD 83
A.2.2 The DL version of the HTML DTD 91
A.3 Modeling HTML documents 112
A.3.1 An HTML document 112
A.3.2 The DL version of the HTML document 118

List of Figures

2.1 A small example of an SGML document instance. 7
2.2 A small example of an SGML document instance using entities and attributes. 8
2.3 An example of an SGML DTD. 14
2.4 An example of an SGML document instance. 15

3.1 Syntax of DL terms for an example language. 24
3.2 The evaluation function for terms in our example language. . 26
3.3 An example is-a hierarchy. 30
3.4 Syntax of DL terms for an example language extended with part constructs. 33
3.5 The evaluation function for terms in our example language extended with part constructs. 34

4.1 Syntax of the most important constructs of CLASSIC($\triangleleft$). . . 44
4.2 How a content of the type: (A , B , C , ...) is modeled in general. 46
4.3 How a content of the type: (A' | B' | C' | ...)+ is modeled in general. 47
4.4 How a content of the type: (A' | B' | C' | ...)* is modeled in general. 47
4.5 How a content of the type: (A' & B' & C' & ...) is modeled in general. Observe that '-CODE X stands for the DL code that restricts the number of fillers of type X. For example: if A'=A+, the '-CODE A = (**atleastp** 1 A-p). 48
4.6 An example of how a content of the type: (A , B , C , ...) is modeled. 48
4.7 An example of how a content of the type: (A' | B' | C' | ...)+ is modeled. 48
4.8 An example of how a content of the type: (A' | B' | C' | ...)* is modeled. 49

4.9 An example of how a content of the type: (A' & B' & C' & ...) is modeled. . . . 49
4.10 An example of how we handle unallowed parts. . . . 51
4.11 An example of how a default value in an attribute declaration is handled. . . . 52
4.12 An example of how an implied default value in an attribute declaration is handled. . . . 53
4.13 An example of how we handle unallowed roles. . . . 54
4.14 An example of how an entity declaration is modeled. . . . 55
4.15 An example of what a typical element instance might look like. 56
4.16 An example of how a string instance is modeled. . . . 56
4.17 An example of how a string instance is modeled. . . . 57
4.18 An example of how a previous instance example is modeled. . 58
4.19 The definition of the close function. . . . 59
4.20 An example of how the close function is called after an individual definition. . . . 59

6.1 The semantics of some new DL constructs intended for handling extra content types. . . . 81
6.2 An example of how a content of the type: (A | B | C | ...)? could be modeled using some extra DL constructs. . . . 81

A.1 What the uppermost part of the HTML document looks like in Netscape. . . . 113

List of Tables

2.1 The meaning of the SGML connectors. 10
2.2 The meaning of the SGML occurrence indicators. 10

4.1 Allowed content types. 47

Chapter 1

Introduction

1.1 Motivation

Recently it has been suggested that the World Wide Web should be seen as a very large database to enable database techniques for searching the Web. In order to accomplish this goal the structure of the Web must be formalized. Thus we need formal methods for describing the structure of HTML documents (the nodes in the World Wide Web). This thesis contributes to this idea by showing how HTML (and parts of the meta language SGML in which HTML is defined) can be formalized in a knowledge base system enabling queries taking advantage of the structure of the documents. Other attempts to solve this problem have been made. In [10] a similar attempt is made using a different formal representation language and in [4] a description of how to query SGML documents using SQL is presented.

This thesis does not only concern the World Wide Web, but also concerns structured documents in general. The results in this thesis can be applied to applications using document databases (SGML databases). Document databases is common in large companies and using the structure of documents gives several advantages in both maintaining and developing documents. The advantages of using the structure of documents is argued for more in e.g. [12].

When maintaining and developing documents it is useful to be able to query the document database. Our presented model of structured documents allows for advanced querying of documents taking advantage of the inherent structure of documents.

1.2 Background

This thesis concerns several different research areas, namely Description Logics, Structured Documents and Information Retrieval. We have dedicated a chapter each to describe the two former areas as they are very important for the purposes of this thesis but we give a very brief description of all three areas now.

Description Logics are logical languages that were designed for reasoning about *classes* of objects and *hierarchies* of classes. Description Logics are gradually extended in different directions to have new functionality and recently Description Logics systems supporting reasoning about part hierarchies of classes were implemented.

Structured Documents is a name for documents that have a known *structure*, i.e. a known ordering of logical parts such as sections and appendices. *SGML* is a standard language used to define different document structures. The well-known language *HTML*, used in the world wide web, is a structured document type. Thus HTML documents are structured documents. The HTML structure is actually defined using SGML.

Information Retrieval is the problem of *specifying* wanted information and of *retrieving* the specified information. The problem of information retrieval is apparent on the Internet. Several so called search engines exist where users can enter words and as a result get pointers to pages containing the entered words. This way of searching is not very efficient though and it often takes a long time to locate the information you want, since you often get too many pointers, many of them irrelevant to the context you had in mind. This is a problem of *precision.* You have good *recall* though, since you get pointers to nearly all documents containing the entered words, at least using some of the search engines which have indexed practically every page on Internet. The problem of precision is based on the fact that it is difficult to specify in a precise manner exactly what kind of information you want.

Several attempts to solve the problem of precision on Internet information retrieval have been made. One of these techniques is learning, or personal information retrieval, where the information retrieval tool *learns* what kind of information its user wants and after a time of usage builds up a profile of the user's interests to be able to judge if a piece of information is interesting for the user and then choose if it will be presented to the user. Another technique to enhance precision of information retrieval from documents is to use the document's structure if that is known. In the case of information retrieval on Internet we know the formal structure of world wide web documents since they are HTML documents and thus we should be able to specify the information we want more precisely. For example, we could state that the part "head" of an HTML document should contain some word and that the "body" part should contain another word. This could clearly result in better precision.

This thesis is part of a larger information retrieval project attempting to use the two methods discussed above to enhance precision. The basic idea is the following: represent Internet documents by Description Logics and store them in a Description Logics system that can reason about the structure of the documents and use Description Logics learning algorithms to employ learning and then place a query engine on top of the Description Logics system to handle queries and answers.

1.3 Purpose

The purpose of this thesis is:

- **Finding a way to represent SGML document types in Description Logics.**

  Using some formal Description Logics framework, or extending existing ones if necessary, a way to represent at least the different kinds of structure elements that are used in the SGML document type specification of HTML should be found.

- **Finding a way to represent document instances of SGML document types in Description Logics.**

  Again using some formal Description Logics framework, or extending existing ones if necessary, a way to represent at least HTML documents should be found.

- **Implementing conversions from document types and instances to Description Logics and vice versa.**

  For practical purposes it is important to have an actual implementation that can convert document types or document instances to Description Logics.

Thus this thesis constitutes the part of the larger project that handles the conversion of HTML documents to Description Logics representations of the documents.

1.4 About this document

When describing subjects, one is of course influenced by the existing written literature on those subjects. In my case I have been greatly influenced by [15] when it comes to Description Logics and by [12] when it comes to SGML and structured documents. I have also taken much advantage from several discussions with my supervisor Patrick Lambrix.

My contributions can be found in chapters 4 and 5 and in Appendix A, while chapters 2 and 3 provide background information for this work.

1.4.1 Prerequisites

The reader of this thesis is assumed to have a general knowledge of computer science and a working knowledge of first order predicate calculus (FOPC). Furthermore, some knowledge of HTML and Internet is assumed. No knowledge of Description Logics or Structured Documents is required.

1.4.2 Outline

- **Chapter 2** gives an introduction to structured documents in general and the SGML parts that are used later in this thesis in particular.
- **Chapter 3** is a tutorial to Description Logics and its different aspects, especially the ones that are used in later chapters. It also introduces the extension of composite objects and describes the Description Logics system CLASSIC.
- **Chapter 4** explains how SGML documents are modeled in Description Logics and gives some examples of usage.
- **Chapter 5** comments the implementation of the various conversions between Description Logics and structured documents.
- **Chapter 6** draws some conclusions from this thesis and gives notes on future work.
- **Appendix A** contains a larger example which shows how the SGML definition of HTML and HTML documents are modeled.

Chapter 2

Structured Documents and SGML

2.1 Introduction

Documents generally have structure. That is, the different parts of a document are ordered in some fashion. Think of this thesis as an example. It has some structure[1], it is divided into chapters, sections and subsections and so on. Even plain text has structure. Words are separated by spaces and sentences are separated by punctuation symbols. It is also true that different documents have different structure. The structure of a newspaper often varies from the structure of a book.

Structure of documents can be defined by *markup*. Markup has been around at least since the time of Gutenberg, when book printing started. Then, and until the introduction of computers in the middle of this century, markup was written on draft versions of documents to indicate how the document should be typeset. For instance, drawing a wavy underline of a word meant that the word should be printed in a bold typeface. Thus markup was done by hand to help typesetters.

Markup had two purposes: to specify the logical parts of a document, and to specify how these parts should be processed (e.g. typeset). The former is referred to as *structural markup* and the latter as *procedural markup*.

Today, one has begun to make a clear distinction between these parts of markup, and mixing them has several disadvantages. Suppose you have a document that is marked up both structurally and procedurally for a specific

[1] Although by the time of writing the structure may not be very clear at all, eventually it obtains a clear structure.

wordprocessor or typesetting program. If you suddenly have to change your wordprocessor/typesetter (maybe because your document has been accepted by a publishing company, and they use different programs than you do), you are in trouble. Unless there exist very advanced conversion programs between your programs and your publisher's, you most likely have to markup all your text once more using their programs. For technical documents that can be very time consuming. For instance, mathematical formulas need very advanced procedural markup.

A solution to this problem has been proposed. Using standardized markup languages for both structural and procedural markup would put an end to all conversion problems. Work on standardizing structural markup is well underway while standardizing procedural markup is a little behind. SGML ([12]) or Standard Generalized Markup Language is a language for defining structural markup.

Generalized markup is the same as structural markup with the exception that generalized markup assumes that the markup language is rigorous, thus meaning that the markup is machine interpretable. SGML has been accepted as an ISO standard [22], and the use of SGML is spreading. The well known markup language HTML ([3]) has been defined by SGML, and several other widely used markup languages defined by SGML exist.

The rest of this chapter gives an introduction to SGML. It is a simplified and incomplete introduction since SGML is very large and complex. We concentrate on what is relevant for this thesis. As the main goal of this thesis is to be able to model HTML documents by Description Logics, we will not discuss SGML concepts that are not used in the HTML DTD. For a complete description of SGML, the reader is referred to the standard book on SGML [12] (written by the creator of SGML Charles F. Goldfarb), or to the ISO standard itself [22]. The presentation below is influenced by [12].

2.2 The structure of SGML

SGML is a language for describing markup languages. Using SGML syntax you can define markup syntax for different documents. Markup separates text into different logical *elements*. In a document type definition you can define different types of elements. You can also define *attributes* for elements and define *entities*. Attributes give additional information about certain elements. For example if we have an element "document" we could define an attribute "author", that could have as value the name of the author of the document. An entity is a bunch of text that can be referenced in a

```
<document>
<head>
<title> A dictionary of gnus </title>
</head>
<body>
<p> This is a paragraph. </p>
<p> This is another paragraph. </p>
</body>
</document>
```

Figure 2.1: A small example of an SGML document instance.

document instance.

An SGML document consists of three parts:

- The SGML declaration for the document.
- The document type definition (DTD).
- A document instance conforming to the above definitions.

The SGML declaration is the document's lexicon. It is a markup declaration that specifies among other things the character set and optional features. We do not go into any detail of SGML declarations.

The document type definition (henceforth referred to as the DTD), defines how the documents belonging to this particular document type should be structured logically. That is, the DTD contains the definitions of the element structure and the attribute definitions and it can also contain entity declarations. Note that an SGML system might also allow a document instance to contain entity references referring to entities other than the ones declared in the DTD. It is entirely up to the system though, if such global entities are allowed.

An instance of an SGML document might look like in Figure 2.1. Logical elements of the text are separated by using start-tags and end-tags. For instance, the whole text is a logical element "document" and is thus bound by the start-tag <document> and the end-tag </document>. Furthermore, the logical element "document" has some structure in the form of parts. In this case it has two parts, namely "head" and "body", and the former part appears before the latter. Both parts "head" and "body" are also structured into parts.

```
<document version=draft>
<head>
<title> The gnu handbook </title>
</head>
<body>
&chapter1;
&chapter2;
&chapter3;
</body>
</document>
```

Figure 2.2: A small example of an SGML document instance using entities and attributes.

An element can have two kinds of basic parts (non-element parts), namely "string" and "EMPTY". A "string" part is a text string, and an "EMPTY" part contains nothing. EMPTY parts are sometimes useful, for instance if we would like to separate sections using a horizontal ruler. It is not logical to let such an element have any content at all, since its only purpose is to let a formatter know about its existence. So, one way to markup a document as containing a horizontal ruler on a specific place would be: <hr></hr>.

SGML allows elements to have attributes. For instance, we might have an attribute "version" for the "document" element. Syntactically this is done using the start-tag as follows: <document version=draft>. This is simply a notifier to the SGML parser that the "version" attribute of "document" has the value "draft".

SGML documents are as mentioned ordered logically into an element structure. Physically, the documents can be seen as ordered into *entities*. The document instance above is not divided physically but when handling large documents it is often convenient to handle small parts at a time. A large document could be divided into a main entity and into different chapter entities. Thus when parsing the document the SGML system has the responsibility to handle the different entities. It may, for example, load and parse them separately, and manage their file locations.

An example of a document instance using the entity technique is presented in Figure 2.2. In this example "&chapter1" is an entity reference. That is, it refers to the entity with name "chapter1". It is then up to the SGML system to handle entity references or signal if documents refer to unknown entities. Entities cannot only be used to ease handling of large documents, they can also be used as macros. That is, you can use entities for commonly written sentences. For instance, if you are writing an article about SGML, and find it disturbing to write the whole sentence "Standard Generalized Markup Language" many times, you can declare an entity to stand for this sentence and then in the document instance you can refer to this entity by using short entity references like "&SGML;".

In the following three sections we describe in more detail the three main concepts of SGML (elements, attributes and entities) and present their syntax in a DTD or a document instance.

2.3 Elements

An element definition (declared in a DTD) contains the following:

- The element name.
- The element content.

The element name is simply the name of the element, that is the name that is used in the element's start-tags and end-tags. It usually consists of a sequence of letters, but can also contain other non-whitespace characters.

The content describes the logical parts that the element has. The content is usually described using a form of regular expression (see [14] for an introduction to regular expressions). An example of an element content is:

$$(\mathrm{p} \mid \mathrm{xmp})^*$$

In the above example "p" and "xmp" are element names, the "|" is a *connector* and the "*" is an *occurrence indicator*. The connectors are used in a glueing fashion. That is, they are used to connect elements in different ways. The "|" connector is a logical xor, meaning that either the element "p" or the element "xmp" but not both must be a part of the element with this content. All SGML connectors are listed and explained in Table 2.1. The groups that are referred to in the table can be elements or element contents.

The occurrence indicators indicate how many times an expression can occur. Occurrence indicators can be used directly on element names or on more complex expressions. The "*" indicator means that the expression it contains may occur 0 or more times. All SGML occurrence indicators are listed and explained in Table 2.2. Thus the content example above means

syntax	*meaning*
G1 | G2	Either group G1 or G2 must occur but not both.
G1 , G2	Both group G1 and G2 must occur and G1 directly before G2.
G1 & G2	Both group G1 and G2 must occur but the order is unimportant.

Table 2.1: The meaning of the SGML connectors.

syntax	*meaning*
G*	The group G can occur 0 or more times.
G+	The group G can occur 1 or more times.
G?	The group G can occur 0 or 1 time.

Table 2.2: The meaning of the SGML occurrence indicators.

that the elements “p” and “xmp” can occur, as parts of the element with that content, any number of times and in any order.

Thus element contents consist of elements mixed in a fashion similar to regular expression. As mentioned above, the leafs in the element hierarchy can be either strings or elements with empty contents.

We will give a further example of a more complex element content:

$$(\text{title}\,,\text{abstract?}\,,(\text{string}\,|\,\text{xmp}\,|\,\text{figure})+\,,\text{references?})$$

This means that the defined element must begin with a title, then optionally an abstract and then strings mixed with examples and figures and finally optionally some references.

We do not define the element contents formally since we only use a very restricted version of these contents later on. For more examples, we refer the reader to e.g. [12].

The element name and the content are defined in a DTD using the following syntax:

<!ELEMENT document (head , body)>

The defined element’s name is “document” and it requires a “head” part directly followed by a “body” part.

2.4 Attributes

Attributes are declared in DTDs in a similar way as elements. An attribute declaration contains the following:

- The name of the element that the attribute is declared for.
- The name of the attribute.
- The possible values of the attribute.
- The default value (or other options).

The element name and attribute name are just sequences of letters or other non-whitespace characters. The possible values of an attribute are either strings (declared by: string) or some enumerated set of values like {draft, finished} declared by: (draft | finished).

Default values are used to allow the writer of a document instance to ignore certain attributes. For example if the element "document" has an attribute "version" we can define the default value for that attribute to be "draft". Thus if a writer of a document instance for this particular DTD ignores that attribute then the SGML system automatically sets the attribute to the default value. If the values of an attribute are declared to be strings, then the default value can be any string we choose. If we have given some enumerated set of values we can choose any one of the values and have that value for default value.

A default value can also be one of a few specific keywords. Setting the default value to be "#REQUIRED", means that a document instance writer must supply an attribute value for this attribute, otherwise the document instance is not correct relative the DTD.

The default value can also be the keyword "#IMPLIED". This means that the document instance writer does not have to supply an attribute value, and that the SGML system will automatically choose an allowed value, if none is given.

The default value can be the keyword "#FIXED" followed by some allowed value. This means that the document instance writer must always supply that specific value.

Some examples of attribute declarations that could be part of a DTD are:

<!ATTLIST document version (draft | finished) draft>

<!ATTLIST a href string #REQUIRED>

<!ATTLIST document author string #FIXED "Johan Åberg">

The first attribute declaration declares the attribute "version" for the element "document". The attribute has the allowed values "draft" and "finished". The default value is "draft". The second attribute declaration declares the attribute "href" for the element "a" to take values that are strings. It also states that an attribute value must be supplied with a document instance. Finally, the third attribute declaration declares the attribute "author" for the element "document". The attribute must always be given a value in a document instance and the value must always be the string "Johan Åberg".

2.5 Entities

There are two basic kinds of entities. Apart from the normal kind we have discussed in section 2.2 above there are also *parameter* entities. They are internal entities that are referenced in a DTD. They are often used to enhance the readability of DTDs by splitting up data that is used many times. The declaration syntax for parameter entities is very similar to the syntax for the normal entities and so is the reference syntax.

An entity declaration contains the name of the entity and the string that the entity stands for. We present some examples of entity declarations that could have been part of a DTD:

<!ENTITY sgml "Standard Generalized Markup Language">

<!ENTITY author "Johan Åberg">

These entities are referenced in a document instance using &sgml; or &sgml . That is, if we have a space character directly after the entity reference we can leave out the ending semicolon.

Parameter entities could be declared as follows in a DTD:

<!ENTITY % head_content (title , abstract)>

<!ENTITY % version_val (draft , finished)>

The parameter entities can only be referenced within a DTD and not within a document instance. A typical use of parameter entities in a DTD would be as follows.

<!ELEMENT head %head_content;>

<!ATTLIST document version %version_val draft>

Observe how the ending semicolon can be left out at the end of the parameter entity reference in the attribute declaration above due to the space character that follows the reference.

The processing of a DTD containing internal entities replaces all references to internal entities with the strings that they stand for before parsing the declarations.

2.6 An example

Now that we have discussed the most important concepts of SGML we present a small example of a DTD and a document instance to give an intuitive picture of what SGML documents look like. The example is small and simplifies the use of SGML, but is sufficient for our purposes. It contains many of the important parts that standard documents consist of. The example DTD is presented in Figure 2.3, and an example of a correct instance relative the DTD is presented in Figure 2.4.

The example describes a document type "document" having a "head" part followed by a "body" part. The element "document" has two attributes "author" and "version". The "author" attribute takes values that are strings and has the string "Johan Åberg" as default value. The "version" attribute takes the value "draft" or the value "finished", and has the former as default. Both the "head" and the "body" part have the attribute "numbering". "head" has "roman" as default value and "body" has "arabic" as default.

The logical element "head" has a "title" part followed by an "abstract" part. Both the "title" and "abstract" are strings. The element "body" consists of a mixture of strings, paragraphs, examples and horizontal rulers. Paragraphs and examples are either a string or an emphasized string. The "hr" element has an attribute "height" which specifies the height of the ruler.

The example instance follows the example DTD and has a correct logical structure and the right kind of attributes and attribute values.

2.7 Additional SGML details

In the remainder of this chapter we discuss some additional details of some extra features in SGML that are mentioned further on.

```
<!ENTITY % values “(draft | finished)”>
<!ENTITY sgml ”Standard Generalized Markup Language”>

<!ELEMENT document (head , body)>
<!ATTLIST document author string “Johan Åberg”>
<!ATTLIST document version %values; draft>

<!ELEMENT head (title , abstract)>
<!ATTLIST head numbering (roman | arabic) roman>

<!ELEMENT title (string)>

<!ELEMENT abstract (string)>

<!ELEMENT body (string | p | xmp | hr)*>
<!ATTLIST body numbering (roman | arabic) arabic>

<!ELEMENT hr EMPTY>
<!ATTLIST hr height string “2”>

<!ELEMENT p (string | em)>

<!ELEMENT xmp (string | em)>

<!ELEMENT em (string)>
```

Figure 2.3: An example of an SGML DTD.

```
<document author="Johan Åberg" version="finished">

<head>
<title> Structured Documents and Description Logics
</title>
<abstract> It is shown how Description Logics can be used to model
structured documents in the form of SGML documents.
</abstract>
</head>

<body numbering=arabic>
<p>
This is the first paragraph in this document.
</p>
<p>
This is the second paragraph.
</p>
<hr height="5"></hr>
<p> This is yet <em>another</em> paragraph.</p>
This is just a string mixed among the paragraphs and the examples.
<xmp> This is an example of an entity reference:
Did you know that SGML stands for &sgml;? </xmp>
This is an ending string.
</body>
</document>
```

Figure 2.4: An example of an SGML document instance.

2.7.1 Characters

In the SGML declaration of a document you can define what character set you are going to use, provided that the character set is large enough so that you can write all SGML constructs. This is convenient since there are so many different areas of text writing using very different characters.

In SGML DTDs and instances, it is possible to write characters by referring to their code by a special syntax. This is often convenient if you use large character sets containing unusual characters or if you are using a limited keyboard. It is also useful when you write text in different languages since many languages use some special characters. This way of referring directly to the character code is also a very effective way of solving problems with portability of text between different countries that use different character standards.

The syntax for referring to characters by their code is very similar to entity references. For example, referring to the hyphen character "-", you write -, supposing the code for the hyphen character in the current character set is the number 45. That is, you start a character reference by &# and end it with a semicolon or a space. Thus, the following entity declarations are equivalent:

```
<!ENTITY hyphen "-">

<!ENTITY hyphen &#45;>
```

2.7.2 Marked sections

In an SGML DTD you can mark a section with the keyword "IGNORE" or "INCLUDE". The syntax is explained by the following example:

```
<![ INCLUDE [
<!ENTITY hyphen &#45;>
]]>

<![ IGNORE [
<!ENTITY hyphen &#99;>
]]>
```

When an SGML parser reads this part of the DTD it includes the entity declaration that defines "hyphen" as the character with code 45, and ignores the other entity declaration.

This is often a convenient way for allowing different versions of a document. You can replace the explicit "INCLUDE" and "IGNORE" keywords by entity references. Thus if you want to have the first entity declaration above in document version 1, but the second declaration in version 2 you can define two entities "version1" and "version2" to be either "INCLUDE" or "IGNORE". Then you do not have to change anything explicitly in the DTD when you want to change version of a document. Thus if the entity "version1" is defined to be "INCLUDE" and "version2" is defined to be "IGNORE" the following part of the DTD is equivalent to the part presented above.

```
<![ &version1 [
<!ENTITY hyphen &#45;>
]]>

<![ &version2 [
<!ENTITY hyphen &#99;>
]]>
```

2.7.3 Included and excluded elements

The content of an element declaration defines what the element can have for parts on the "next level" in the element hierarchy. It does not say anything about what elements may occur as parts further down in the hierarchy. Sometimes it can be useful to be able to say that an element can occur as a part of an element on any level down in the element hierarchy.

This can come in handy for example if you want to allow index elements anywhere in a document instance, and this is convenient since that is the usual way of creating an index of a document. When creating the index you collect all index elements within the document and format them at the end. Thus when defining the DTD, you would have to include the index element in every element content, and that would be very tiresome. Instead you could mark the index element as an included element in the definition of the top (top of the element hierarchy) element.

The syntax for included elements is exemplified as follows.

```
<!ELEMENT document (head,body) +(index|comment)>
```

Similarly, sometimes you want to exclude elements explicitly to appear on any level under an element in the element hierarchy. For example, suppose the element "abstract" consists of a number of "paragraph" parts.

Suppose also that the "body" element consists of "paragraph" parts (among others) and that the "paragraph" element has "string" parts and "figure" parts. Figures are of course allowed in a document body, but not in an abstract. Thus, if we insist that an abstract should consist of paragraph parts we can solve this problem by excluding the "figure" element in the definition of the "abstract" element. The syntax for doing this is very similar to the syntax for including elements and is exemplified as follows.

<!ELEMENT abstract (p)+ -(figure)>

In the same way as we could include several elements above we can also exclude several elements at a time by separating them, by the "|" character. We can also both include and exclude elements in an element definition, as in the following example:

<!ELEMENT abstract (p)+ -(figure) +(index)>

2.7.4 Markup minimization

There are two optional parameters in the element declaration called *minimization* parameters. The two parameters are used to specify whether the start-tag or the end-tag respectively can be omitted in document instances. That is, we let the writer of the document instance decide if it is necessary to include a start-tag or end-tag to get the intended element structure.

The two optional parameters can be made compulsory by declaring "OMITTAG YES" in the SGML declaration of the document. The syntax for the two parameters are either "-" or "O". The hyphen means that the tag is compulsory and the "O" that the tag is omissable. The parameters are written directly after the element name in the element declaration. The first stands for the start-tag and the second for the end-tag. The following is an example of how it might look:

<!ELEMENT p - O (string)>

This would thus mean that the start-tag of the element "p" is compulsory while the end-tag may be omitted.

2.7.5 Comments

Comments are allowed in DTDs between declarations and between tags in document instances. A comment declaration is always started by the following string: "<!- - " and ended by "- ->". An example of what a comment might look like follows:

```
<!-- This is an example of a comment. -->
```

Chapter 3

Description Logics

3.1 Introduction

Description Logics (DLs) are a family of logical languages useful for reasoning about classes of objects and hierarchies of classes as well as binary relations between objects and between classes. The objects are called *individuals*, the classes are called *concepts* and the (binary) relations are called *roles*. The languages can be seen as subsets of standard FOPC[1].

Description Logics originate from frame systems and semantic nets in the AI community ([20]). The first knowledge representation system using what we now call Description Logics was KL-ONE ([9]), created by Ronald Brachman. Since then DLs have sometimes been called KL-ONE type languages. They have also been called Concept Languages or Terminological Logics. The latter name comes from the fact that DLs are useful to describe terminologies.

Different DLs differ in several aspects:

- They have different expressivity. For instance, some languages have a construct for negation and some have not.
- Depending on what constructs are allowed in the different languages, they also have different complexity. Some languages are decidable with respect to the subsumption problem and some are not. Some languages have tractable inference algorithms and some have not. See [11] for some complexity results.
- There exist different kinds of implemented systems using different languages and providing different forms of reasoning.

[1]FOPC stands for First Order Predicate Calculus.

Since the origin of KL-ONE, a whole range of knowledge representation systems have emerged, among which KRIS ([1]), LOOM ([17]) and CLASSIC ([6], [7], [8]) are examples. They all differ somewhat in what language they use. We are going to use the CLASSIC system in this thesis so we will present that system more thoroughly later on.

In an attempt to clarify the difference between FOPC and DL we present the basic ideas of FOPC below and then we define Description Logics in a similar manner. The introduction of FOPC in the next section is done in the style of [21].

3.2 FOPC Revisited

The structure of FOPC formulas is as follows. The *basic* building blocks are *terms*. Terms are variables and functions mixed in a well-defined order. For example, a term might be: $f(x, g(y))$, where f and g are function symbols and x and y are variables. The next building blocks are *atomic formulas*. These are predicates applied to terms. An example of an atomic formula is: $Q(f(x, g(y)))$, where Q is a predicate symbol. The next step is to combine atomic formulas in a well-defined order by connectives which results in *formulas*. The connectives are usually the following: $\forall, \exists, \neg, \wedge, \vee, \rightarrow$ *and* $\leftrightarrow$.

A typical formula in FOPC might look like:

$$\forall x \exists y [P(x, x) \wedge Q(f(x, g(y))]$$

where P and Q are predicate symbols applied to terms.

We will describe what the different building blocks mean. Let us begin with terms, as they are the basic building block. In mathematics the value of $f(x)$ depends on what value x has and how f is defined. If f is the successor function on the natural numbers and x has the value 2 then $f(x)$ has the value 3. So the value of a term should depend on an interpretation of the involved variables and function symbols. The interpretation should consist of a domain of values, a mapping from the variables to the domain and for each function symbol a mapping from values to values.

We continue with atomic formulas. A predicate is a statement for objects which is either true or false. For example, $x = y$ should evaluate to true given an interpretation where $=$ is the equality relation over the natural numbers and both x and y have the value 4. Thus atomic formulas are evaluated to truth values.

FOPC formulas are also evaluated to truth values. The atomic formulas are evaluated first and the resulting truth values are then combined according to the definitions of the different connectives.

An interpretation of a FOPC formula is thus defined as containing a domain of values, a mapping from variables to values in the domain, a mapping from values to values for each function and a mapping from values to truth values for all predicates.

If a FOPC formula evaluates to true given an interpretation we say that the interpretation is a *model* of the formula. We can also define the *model set* of a formula, to be the set of all models. Unless we restrict the cardinality of domains to be finite we usually get infinite model sets. If a formula has at least one model we say that the formula is *satisfiable*. If a formula is always true no matter what interpretation, we say that the formula is *valid* or a *tautology*. Similarly, if a formula lacks models, we say that it is *unsatisfiable* or a *contradiction*.

Now we can define different relations between formulas, such as *logical equivalence*. Two logical formulas A and B are logically equivalent iff the model set of A is equal to the model set of B, written $A \simeq B$. We can also define *logical implication* as follows. A formula A logically implies a formula B iff A's model set is a subset of B's model set, that is, whenever A is true, B is also true, written $A \models B$.

Now we can define rules for these relations. Remember the well known rule Modus Ponens: $A, A \rightarrow B \models B$. That is, if we know A and we know that A implies B then we also know B. If this rule is always correct then we say that the rule is *sound*. A set of such rules for a given relation is called an *inference system*. An inference system is sound iff all its members are sound. The system is *complete* iff whenever $A \models B$ we can get the conclusion B from A using the existing rules.

Complete inference systems *do* exist for FOPC, for instance the system Natural Deduction.

FOPC is unfortunately known to be undecidable, that is, it is impossible to have a mechanical procedure which correctly classifies a FOPC formula as satisfiable or unsatisfiable in a finite number of steps. See e.g. [5] for a proof[2].

Now that we have reviewed the most important concepts of FOPC we are ready to introduce Description Logics. The presentation and some of the formal definitions below are influenced by [15] and [21].

[2]Actually, this proof assumes that Church's thesis holds, since the proof works by showing that if FOPC were decidable then the Halting problem would also be decidable.

<concept-term> ::=	$\top$ |
	<concept-name> |
	(**and** <concept-term>+) |
	(**all** <role-term> <concept-term>) |
	(**atleast** <positive-integer> <role-term>) |
	(**atmost** <non-negative-integer> <role-term>) |
	(**fills** <role-term> <individual-name>+)
<individual-term> ::=	<concept-term>
<role-term> ::=	<role-name>
<concept-name> ::=	<symbol>
<individual-name> ::=	<symbol>
<role-name> ::=	<symbol>

Figure 3.1: Syntax of DL terms for an example language.

3.3 Syntax and Semantics

Description Logics have the following building blocks:

- Terms
- Axioms

Terms are the *basic* building blocks for DLs just like terms are for FOPC. Unlike FOPC there are three different kinds of terms: *concept terms*, *individual terms* and *role terms*. The formal syntax for the different terms of an example language is defined as in Figure 3.1.[3]

We will give an example of what a typical concept term (or individual term, as they are identical in our example language) in this example language might look like:

(**and** man (**atleast** 1 child_of))

Here 'man' is a concept name and 'child_of' is a role name. This might be a description of the concept 'father'. This term intuitively stands for all objects who are fathers.

[3] Observe that this language is actually a part of the CLASSIC language that we are going to use later on.

In this example language role terms are just names although some other DL languages allow some constructs for forming more complex role terms.

We have now given an introduction to Description Logics terms, so let us continue with axioms. One usually distinguishes between terminological axioms and assertional axioms. Terminological axioms associate a concept name with a concept term. Assertional axioms associate individual names with individual terms.

A set of terminological axioms is usually called a *T-Box* and a set of assertional axioms is usually called an *A-Box.* These are formally defined below.

An example of a terminological axiom might be:

$$\text{father} \doteq (\textbf{and}\ \text{man}\ (\textbf{atleast}\ 1\ \text{child\_of}))$$

We now describe the semantics of DL terms and axioms. DL terms and axioms are evaluated in much the same way as FOPC formulas. First of all we need to know what the world we are trying to model looks like, that is, what objects are there and what are the relations between them i.e. we need to know the set of all the objects and how all concepts and roles are defined for those objects and what individual correspond to what object. This context is called an *interpretation* just as for FOPC. This evaluation mechanism is formally defined below.

So, in the example concept term above, if the objects belonging to the concept 'man' are: *john*, *martin* and *peter*, i.e. the concept 'man' is the set $\{john, martin, peter\}$, and the role child_of is defined as: $\{< john, martin >, < martin, peter >\}$ (meaning that *martin* is a child of *john* and that *peter* is a child of *martin*), then the value of the term describing the concept 'father' would be the set $\{john, martin\}$. Thus the value of the term is the set of all objects that are commonly known as fathers.

The semantics of a term is the set of objects that satisfies the term. The semantics of an axiom on the other hand is a truth value just as for FOPC formulas. That is, the value of the example axiom above is true iff the set of objects belonging to the concept-name 'father' is the same as the set of objects that belong to the term composition on the right hand side of the axiom.

Now, before we define the semantics of terms and axioms formally we define *vocabularies* and *interpretations.*

Definition 1 [Vocabularies] *A Description Logic vocabulary* $\mathcal{V}$ *is a tuple* $< I, R, C >$ *where* I *is a set of individual names,* R *is a set of role names*

$val[A, \mathcal{I}] = \varepsilon[A]$
$val[i, \mathcal{I}] = \varepsilon[i]$
$val[r, \mathcal{I}] = \varepsilon[r]$
$val[\top, \mathcal{I}] = \Delta$
$val[(\textbf{and}\ \alpha\ \beta), \mathcal{I}] = val[\alpha, \mathcal{I}] \cap val[\beta, \mathcal{I}]$
$val[(\textbf{all}\ r\ \alpha), \mathcal{I}] = \{x \in \Delta \mid \forall y \in \Delta[< x, y > \in val[r, \mathcal{I}] \rightarrow y \in val[\alpha, \mathcal{I}]]\}$
$val[(\textbf{atleast}\ m\ r), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid < x, y > \in val[r, \mathcal{I}]\} \geq m\}$
$val[(\textbf{atmost}\ m\ r), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid < x, y > \in val[r, \mathcal{I}]\} \leq m\}$
$val[(\textbf{fills}\ r\ i_1\ ...\ i_m), \mathcal{I}] =$
$\{x \in \Delta \mid < x, val[i_1, \mathcal{I}] > \in val[r, \mathcal{I}] \wedge ... \wedge < x, val[i_m, \mathcal{I}] > \in val[r, \mathcal{I}]\}$

Figure 3.2: The evaluation function for terms in our example language.

and C is a set of concept names.

Definition 2 [Interpretations for vocabularies] *An interpretation $\mathcal{I}$ for a Description Logic vocabulary $\mathcal{V}$ is a tuple, $< \Delta, \varepsilon >$, where Δ is a set and ε is an interpretation function such that $\varepsilon[A] \subseteq \Delta$ where A is a concept name in $\mathcal{V}$, $\varepsilon[r] \subseteq \Delta \times \Delta$ where r is role name in $\mathcal{V}$, $\varepsilon[i] \in \Delta$ when i is an individual name in $\mathcal{V}$ and ε respects the unique names assumption for individuals in $\mathcal{V}$.*[4]

The set of all interpretations for a given vocabulary $\mathcal{V}$ is written $\mathcal{I}_{\mathcal{V}}$.

Thus an interpretation gives the meaning of names, which are the basic parts of terms.

We present an evaluation function for terms in our example language in Figure 3.2. The evaluation function is defined for a vocabulary $\mathcal{V}$ given an interpretation $\mathcal{I} =< \Delta, \varepsilon >\in \mathcal{I}_{\mathcal{V}}$. Observe that A is a concept name, r is a role name, i is an individual name and α and β are general concept terms.

Observe that defining a separate evaluation function for terms (the *val* function) is not a standard in Description Logics. One usually just extends the interpretation function (the ε function) to be defined over terms as well as over the objects in the vocabulary. If we extend the interpretation function that way we make the interpretation of the term constructs part of our normal interpretation of objects. However, we really want to have the interpretation of term constructs fixed, not reinterpreted every time we interpret a term, which can easily be believed if one does not separate

[4] The unique names assumption for individuals states that $\varepsilon[i_1] \neq \varepsilon[i_2]$, whenever $i_1 \neq i_2$.

interpretations of term constructs and objects. Thus our goal of introducing a separate evaluation function for terms is to make it more explicit that we interpret the term constructs once and for all for each possible evaluation function and that using this evaluation function, an interpreter of terms just has to interpret the objects in the vocabulary. The notion of a separate evaluation function has been used previously in [21].

The notions of T-Box and A-Box and knowledge base are now formally defined.

Definition 3 **[T-Box]** *A T-Box $\mathcal{T}$ for a vocabulary $\mathcal{V} =< I, R, C >$ is a set of* terminological axioms *of the following kinds:*

$$A \doteq \alpha \text{ and } A \dot{\leq} \alpha$$

where $A \in C$ and α is a general concept term for $\mathcal{V}$ (i.e. using only names in $\mathcal{V}$) and each A occurs only once on the left-hand side of an axiom and $\mathcal{T}$ is acyclic.

Definition 4 **[A-Box]** *An A-Box $\mathcal{A}$ for a vocabulary $\mathcal{V} =< I, R, C >$ is a set of* assertional axioms *of the following kind:*

$$i :: \alpha$$

where $i \in I$ and α is a general individual term for $\mathcal{V}$ (i.e. using only names in $\mathcal{V}$).

Definition 5 **[Knowledge base]** *A Knowledge base $\mathcal{K}$ for a vocabulary $\mathcal{V}$ is a tuple $< \mathcal{T}, \mathcal{A} >$ where $\mathcal{T}$ is a T-Box for $\mathcal{V}$ and $\mathcal{A}$ is an A-Box for $\mathcal{V}$.*

Now we have syntax for building knowledge bases, but we still need semantics. We already have semantics for terms but now it is time for the axioms and knowledge bases. We define the evaluation function *val* for knowledge bases as follows[5].

Definition 6 **[Evaluation of Knowledge bases]** *Given a knowledge base $\mathcal{K} =< \mathcal{T}, \mathcal{A} >$ for a vocabulary $\mathcal{V} =< I, R, C >$ the evaluation function val is defined as*

$$val[\mathcal{K}, \mathcal{I}] = min(val[\mathcal{T}, \mathcal{I}], val[\mathcal{A}, \mathcal{I}])$$

where $\mathcal{I} \in \mathcal{I}_{\mathcal{V}}$.

[5]Below T and F stand for the truth values true and false, and we define the relation $<$ over the truth values as $F < T$. The function min is defined as the minimization function over the $<$ relation, thus for example, $min(T, F) = F$ but $min(T, T) = T$

val is then defined for an A-Box (and similarly for a T-Box) $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ *for a vocabulary* $\mathcal{V}$ *as follows*

$$val[\mathcal{A}, \mathcal{I}] = min(val[a_1, \mathcal{I}], val[a_2, \mathcal{I}], ..., val[a_n, \mathcal{I}])$$

where $\mathcal{I} \in \mathcal{I}_{\mathcal{V}}$.

val is defined for assertional axioms $a = (i :: \alpha)$ *for a vocabulary* $\mathcal{V}$ *and* $\mathcal{I} \in \mathcal{I}_{\mathcal{V}}$ *as*

$$val[a, \mathcal{I}] = \begin{cases} T & \textit{if } val[i, \mathcal{I}] \in val[\alpha, \mathcal{I}] \\ F & \textit{otherwise} \end{cases}$$

val is defined for terminological axioms $t = (A \doteq \alpha)$ *and* $t = (A \dot{\leq} \alpha)$ *for a vocabulary* $\mathcal{V}$ *and* $\mathcal{I} \in \mathcal{I}_{\mathcal{V}}$ *as*

$$val[(A \doteq \alpha), \mathcal{I}] = \begin{cases} T & \textit{if } val[A, \mathcal{I}] = val[\alpha, \mathcal{I}] \\ F & \textit{otherwise} \end{cases}$$

$$val[(A \dot{\leq} \alpha), \mathcal{I}] = \begin{cases} T & \textit{if } val[A, \mathcal{I}] \subseteq val[\alpha, \mathcal{I}] \\ F & \textit{otherwise} \end{cases}$$

A concept defined in a T-Box as $A \doteq \alpha$ is called a *defined* concept. This means that we have given necessary *and* sufficient conditions in α for an individual to belong to A. On the other hand, a concept defined as $A \dot{\leq} \alpha$ is said to be *primitive*. This means that we have given necessary conditions but *not* sufficient conditions for an individual to belong to the concept A. Observe how this is taken care of above in the semantics by the function *val*. When we define logical relations in the next section we see that we are able to automatically classify individuals into defined concepts but not into primitive concepts. The only way to make an individual a member of a primitive concept is to explicitly tell the knowledge base that the individual is a member of the concept.

For example it is reasonable to make the concept father a defined concept as we did above (repeated here):

$$\text{father} \doteq (\textbf{and}\ \text{man}\ (\textbf{atleast}\ 1\ \text{child\_of}))$$

since these restrictions (being a man and having at least 1 child) seem both necessary and sufficient.

Describing a car is not so easy. It can be seen as having some necessary conditions like having four tyres and having an engine and so on but these conditions are obviously not sufficient (think of go-carts). It seems impossible to enumerate all necessary conditions, so one would have to define car as a primitive concept and perhaps enumerate some of the necessary conditions, like:

car $\dot{\leq}$ (**and** (**atleast** 1 engine-role) (**atmost** 1 engine-role) (**all** engine-role engine) (**atleast** 4 tyre-role) (**atmost** 4 tyre-role) (**all** tyre-role tyre)).

On the other hand one could define the concept of red cars as a defined concept as the conjunction:

red-car $\doteq$ (**and** car (**all** color red)).

Now, before we can proceed to define some logical relations we need the notions of model and satisfiability.

Definition 7 [**Model sets of concept terms**] *The model set of a concept term α under the vocabulary $\mathcal{V}$, is defined to be the set:*

$$Mod(\alpha) = \{\mathcal{I} \in \mathcal{I}_{\mathcal{V}} \mid val[\alpha, \mathcal{I}] \neq \emptyset\}.$$

Definition 8 [**Model sets of axioms**] *The model set of an axiom a for T-Box or A-Box under the vocabulary $\mathcal{V}$, is defined to be the set:*

$$Mod(a) = \{\mathcal{I} \in \mathcal{I}_{\mathcal{V}} \mid val[a, \mathcal{I}] = T\}.$$

Definition 9 [**Model sets of A-Boxes and T-Boxes**] *The model set of a T-Box $\mathcal{T}$ (and similarly for A-Boxes) is defined to be the set:*

$$Mod(\mathcal{T}) = Mod(\{t_1, t_2, ...\}) = Mod(t_1) \cap Mod(t_2) \cap ... = \bigcap_{i \geq 1} Mod(t_i).$$

Definition 10 [**Model sets of Knowledge bases**] *The model set of a Knowledge base $\mathcal{K} =< \mathcal{T}, \mathcal{A} >$ is defined to be the set:*

$$Mod(\mathcal{K}) = Mod(\mathcal{T}) \cap Mod(\mathcal{A}).$$

Definition 11 [**Satisfiability**] *A Knowledge base $\mathcal{K}$ (and similarly for axioms, T/A-boxes and concept terms) is said to be satisfiable iff it has at least one model.*

Observe that so far we have been able to do all our formal definitions independent of what term language we use. This means that we can go ahead and change term languages and their semantics as we like without having to change anything in the underlying formal framework.

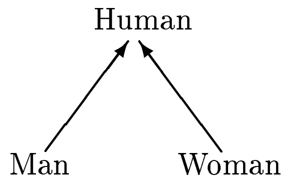

Figure 3.3: An example is-a hierarchy.

3.4 Subsumption

The most important relation for Description Logics is the *subsumption* relation. Subsumption is a relation between concepts and is used to model the is-a hierarchy. See [20] for a discussion of the is-a relation. The subsumption relation is formally defined for concept terms as follows.

Definition 12 [The subsumption relation] *The subsumption relation, written $\Rightarrow$, is defined for concept terms over a vocabulary $\mathcal{V}$ as follows:*

$$\alpha \Rightarrow \beta \textit{ iff } val[\alpha,\mathcal{I}] \subseteq val[\beta,\mathcal{I}]$$

for every $\mathcal{I} \in \mathcal{I}_{\mathcal{V}}$.

$\alpha \Rightarrow \beta$ is pronounced as 'α is subsumed by β', or 'β subsumes α'.

The symbol $\Rightarrow$ could intuitively be seen as a normal FOPC implication, although implication is defined for *formulas* in FOPC and here we use it for DL *terms*. The intuition is the same: $A \Rightarrow B$ means if A 'holds' then B also 'holds', whether A and B are FOPC formulas or concept terms.

The is-a relation can be used to reason about descriptions. For example, we know that man and woman are humans. However, a man is not a woman or the other way around. We could describe this information by forming a hierarchy or tree. In Figure 3.3 we show an example is-a hierarchy. In the figure, the arrows represent is-a relations.

We define the elements in the figure as concepts:

$$\text{human} \;\dot{\leq}\; \top$$

$$\text{man} \doteq (\textbf{and}\ \text{human}\ (\textbf{fills}\ \text{sex male}))$$

$$\text{woman} \doteq (\textbf{and}\ \text{human}\ (\textbf{fills}\ \text{sex female}))$$

According to the definition of both man and woman we see that all objects that are members of those concepts must also be members of the concept human. Thus the concepts man and woman are subsumed by the concept human.

Just as we could define rules for the logical implication relation for FOPC (remember Modus Ponens) we can define subsumption rules for Description Logics. One rule could be: if we know that $C_1 \Rightarrow C_2$ then we know that (**all** r C_1) $\Rightarrow$ (**all** r C_2). This is obviously true given the semantics of the all-construct (restricting the role range, restricts the concept members).

As we mentioned before different Description Logics systems have different languages, but they all use the subsumption relation. The algorithms they use for maintaining the subsumption hierarchy may differ though (see e.g. [1], [7] or [17]).

3.5 Composite Objects

We have now presented the subsumption relation and we are thus equipped for handling the is-a hierarchy. Sometimes though, this is not enough for modeling some domains. When we want to model classes and subclasses the is-a relation is fine, but if we want to model composite objects and reason about them we need a part-of relation. We could model this using just a role 'part' for connecting concepts or individuals as parts or wholes, but this is sometimes not enough for modeling composite objects.

For example, the definition of the concept car was a little bit unintuitive. We used two different roles 'engine-role' and 'tyre-role' and restricted the fillers for these roles to be of the correct type (belong to the correct concept) and we also restricted the number of fillers for each type. Both the tyres and the engine are parts of a car. If we wanted to describe this using a role 'part' we would find ourselves in trouble, because then we could not describe the fact that a car has exactly 4 tyre parts. We could not do this in any other way than we have already done, that is making a distinction between the different kind of parts using different roles, but using roles in this way is rather restrictive. We cannot impose constraints to hold between different kind of parts. Neither could we distinguish between part roles and ordinary roles.

Another approach to the problem of composite objects would be to give parts a special status just like roles, concepts or individuals. This method is proposed and described in more detail in [15]. It is also the method we are going to use later on. So, if we let part names be special objects we could define special constructs for restricting parts, and then we could model objects having different restrictions on its different kind of parts. Then we might describe our concept car in the following way (the syntax will be discussed more later):

car $\doteq$ (**and** vehicle (**atleastp** 1 engine-part) (**atmostp** 1 engine-part)
(**atleastp** 4 tyre-part) (**atmostp** 4 tyre-part))

Notice that unlike before we have defined the concept car as a defined concept and assumed the existence of a concept vehicle (defined or primitive). The new constructs **atleastp** and **atmostp** are very similar to **atleast** and **atmost** but the former constructs restrict part names instead of roles.

One thing we still cannot model is constraints between different types of parts. For instance we might want to impose the (silly) constraint that the engine should be heavier than any of the tyres, or that all the tyres should be of a specific make. New constructs have been introduced to handle this and we can introduce the following description:

car $\doteq$ (**and** vehicle (**atleastp** 1 engine-part) (**atmostp** 1 engine-part)
(**atleastp** 4 tyre-part) (**atmostp** 4 tyre-part)
(**allp** tyre-part michelin)
(**pp-constraint** heavier engine-part tyre-part))

The new construct **allp** is similar to **all** but instead of restricting role fillers it restrict part fillers of a specific part name. The construct **pp-constraint** imposes restrictions between the different part names. It requires that all part fillers for the two part names should be related by a role (the role 'heavier' in the above example).

The new **pp-constraint** construct could also have been introduced for roles but such a constraint construct does not make sense for roles in general.

We present the syntax of our example language extended with part constructs in Figure 3.4. Observe that we have introduced the construct **part-fills**, which is the part equivalent to **fills**. That is, the construct gives you a way to specify concepts which require specific fillers for specific part names.

Before we give the semantics we have to redefine the notions of vocabularies and interpretations for vocabularies. We have to do this since we have not only changed the syntax of our language, we have also given part names a special object status. The new definitions, which are very similar to the old ones, are given below.

Definition 13 **[Vocabularies redefined]** *A Description Logic vocabulary* $\mathcal{V}$ *is a tuple* $< I, R, C, P >$ *where* I *is a set of individual names,* R *is a set of role names,* C *is a set of concept names and* P *is a set of part names.*

<concept-term> ::=	⊤ |
	<concept-name> |
	(**and** <concept-term>+) |
	(**all** <role-term> <concept-term>) |
	(**atleast** <positive-integer> <role-term>) |
	(**atmost** <non-negative-integer> <role-term>) |
	(**fills** <role-term> <individual-name>+) |
	(**allp** <part-name-name> <concept-term>) |
	(**atleastp** <positive-integer> <part-name-name>) |
	(**atmostp** <non-negative-integer>
	<part-name-name>) |
	(**part-fills** <part-name-name>
	<individual-name>+) |
	(**pp-constraint** <role-name> <part-name-name>
	<part-name-name>)
<individual-term> ::=	<concept-term>
<role-term> ::=	<role-name>
<concept-name> ::=	<symbol>
<individual-name> ::=	<symbol>
<role-name> ::=	<symbol>
<part-name-name> ::=	<symbol>

Figure 3.4: Syntax of DL terms for an example language extended with part constructs.

$val[A,\mathcal{I}] = \varepsilon[A]$
$val[i,\mathcal{I}] = \varepsilon[i]$
$val[r,\mathcal{I}] = \varepsilon[r]$
$val[n,\mathcal{I}] = \varepsilon[n]$
$val[\top,\mathcal{I}] = \Delta$
$val[(\textbf{and}\ \alpha\ \beta), \mathcal{I}] = val[\alpha,\mathcal{I}] \cap val[\beta,\mathcal{I}]$
$val[(\textbf{all}\ r\ \alpha), \mathcal{I}] = \{x \in \Delta \mid \forall y \in \Delta[< x,y > \in val[r,\mathcal{I}] \rightarrow y \in val[\alpha,\mathcal{I}]]\}$
$val[(\textbf{atleast}\ m\ r), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid < x,y > \in val[r,\mathcal{I}]\} \geq m\}$
$val[(\textbf{atmost}\ m\ r), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid < x,y > \in val[r,\mathcal{I}]\} \leq m\}$
$val[(\textbf{fills}\ r\ i_1\ ...\ i_m), \mathcal{I}] =$
$\{x \in \Delta \mid < x, val[i_1,\mathcal{I}] > \in val[r,\mathcal{I}] \wedge ... \wedge < x, val[i_m,\mathcal{I}] > \in val[r,\mathcal{I}]\}$
$val[(\textbf{allp}\ n\ \alpha), \mathcal{I}] = \{x \in \Delta \mid \forall y \in \Delta[y \lhd_n x \rightarrow y \in val[\alpha,\mathcal{I}]]\}$
$val[(\textbf{atleastp}\ m\ n), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid y \lhd_n x\} \geq m\}$
$val[(\textbf{atmostp}\ m\ n), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid y \lhd_n x\} \leq m\}$
$val[(\textbf{part-fills}\ n\ i_1\ ...\ i_m), \mathcal{I}] = \{x \in \Delta \mid val[i_1,\mathcal{I}] \lhd_n x \wedge ... \wedge val[i_m,\mathcal{I}] \lhd_n x\}$
$val[(\textbf{pp-constraint}\ r\ n_1\ n_2), \mathcal{I}] =$
$\{x \in \Delta \mid \forall y_1, y_2 \in \Delta[(y_1 \lhd_{n_1} x \wedge y_2 \lhd_{n_2} x) \rightarrow < y_1, y_2 > \in val[r,\mathcal{I}]]\}$

Figure 3.5: The evaluation function for terms in our example language extended with part constructs.

Definition 14 [Interpretations for vocabularies redefined] *An interpretation $\mathcal{I}$ for a Description Logic vocabulary $\mathcal{V}$ is a tuple, $< \Delta, \varepsilon >$, where Δ is a set and ε is an interpretation function such that $\varepsilon[A] \subseteq \Delta$ where A is a concept name in $\mathcal{V}$, $\varepsilon[r] \subseteq \Delta \times \Delta$ where r is role name in $\mathcal{V}$, $\varepsilon[i] \in \Delta$ when i is an individual name in $\mathcal{V}$, $\varepsilon[n] \subseteq \Delta \times \Delta$ when n is a part name name in $\mathcal{V}$ and ε respects the unique names assumption for individuals in $\mathcal{V}$.*[6]

The evaluation function for terms in our example language extended with part constructs is presented in Figure 3.5. The evaluation function is as usual defined for a vocabulary $\mathcal{V}$ given an interpretation $\mathcal{I} =< \Delta, \varepsilon > \in \mathcal{I}_\mathcal{V}$. Observe that A is a concept name, r is a role name, i is an individual name, α and β are general concept terms and n is a part name. We write $y \lhd_n x$ as a short hand for $< y, x > \in val[n,\mathcal{I}]$. We pronounce $y \lhd_n x$ as y is an n-part of x or y is a direct part of x with name n.

Now that we have increased the syntax of our example language we have

[6]The unique names assumption for individuals states that $\varepsilon[i_1] \neq \varepsilon[i_2]$, whenever $i_1 \neq i_2$.

to check that we still can handle the subsumption relation. We probably have to add inference rules for this relation. One subsumption rule we might have to add would be: if we know that $C_1 \Rightarrow C_2$ we know that (**allp** n C_1) $\Rightarrow$ (**allp** n C_2). This obviously holds since restricting the range of a part name also restricts the concept members.

3.6 CLASSIC

As mentioned above CLASSIC is an implemented Description Logics system. It has about the same syntax as the one we presented first in this chapter. Only two extra constructs are used in this thesis.

The first is the **one-of** construct. It allows you to define a set of individuals. For example one could define the concept soft drink as:

soft-drink $\doteq$ (**one-of** coca-cola pepsi-cola sprite fanta)

The other construct we use is quite different from the former constructs. The construct **test-c** allows you to use procedural tests, written in the host language, to check if individuals satisfy the test. Thus the procedural test must return truth values. As an example of this we could define the concept driver as:

driver $\doteq$ (**and** person (**all** age (**test-c** driver-age-test)))

Then we could define the following function in Lisp as that is the implementation language of the CLASSIC version that we use:

```
(defun driver-age-test (age)
   (if (numberp age) (≥ age 18)))
```

Observe that the minimum age for having a drivers license in Sweden is 18.

Apart from the usual concept terms CLASSIC also has some built-in concepts like strings and lists.

Lately CLASSIC has also been updated with the part constructs that were presented in the previous section. All of these will be used later on (see [15] for a description of the new constructs and functionality). Note that the CLASSIC version containing the updates is not an official version and will henceforth be referred to as CLASSIC($\lhd$).

As for roles, CLASSIC allows you to define a role as an attribute, that is the role is functional. It has the implicit restriction that for each individual the role can have at most one filler. This is used for convenience later on.

CLASSIC also allows one to *close* a role for an individual. This can come in handy since CLASSIC has an open world assumption (as all DLs). That is, what CLASSIC does not know explicitly is unknown. For example, assume we have a concept that requires that a certain role has at most one filler. We also have an individual that would be a member of this particular concept if only it would satisfy the role filler requirement. Assume that it has no known fillers for the given role. Unless the role is declared as an attribute or closed for the current individual CLASSIC will not recognize the individual as a member of the concept because given the open world assumption the individual could have fillers that CLASSIC does not know about.

In the new implementation containing part constructs one also has the ability to close part names for individuals. This is important for the same reason as above, and is used later on.

CLASSIC allows for *rules.* Rules can be used for adding properties to individuals belonging to a given concept, that are not definitional but hold anyway. An example of this might be a rule that gives default fillers for a specific role if the individual lacks fillers for this role. Having at least one filler might not be a requirement for membership of the concept in question but membership might imply at least a default filler for the role. We use rules for this purpose later on.

CLASSIC performs different inferences. For concepts it does classification. That is, when a concept is defined it is put in its place according to the subsumption (is-a) hierarchy. Concepts are not allowed to change (be redefined). Thus classification is done once and for all at definition time.

For individuals recognition and propagation is performed. Recognition is the task of checking if an individual is a member of a certain concept or not. Propagation means that individuals are given all the information that is known from the knowledge base. That is, for example if an individual is recognized as a member of some concept, then knowledge about that concept (from its place in the is-a hierarchy) can be used to get further knowledge about the individual. This knowledge is then added to the individual. The new implementation of CLASSIC (CLASSIC($\triangleleft$)) containing part constructs also places individuals in their correct place in the part-of hierarchy.

CLASSIC also triggers rules that have a concept as antecedent as soon as an individual is defined (or redefined) and the individual is found to be a member of the antecedent concept.

The syntax that CLASSIC uses is slightly different from the one we have used here. The CLASSIC version that we use is built on top of Common Lisp so all communication with CLASSIC is through Lisp functions. Defining a concept is done like this:

(cl-define-concept 'man '(and human (fills sex male)))

The mentioned concepts, roles and individuals have to be defined before this concept is defined. This applies in general for CLASSIC. Everything has to be defined in the correct order. The definition of an individual looks like this:

(cl-create-ind 'charles '(and man (fills name "charles")))

The definition of a role that is an attribute is done as:

(cl-define-primitive-role 'name :attribute t)

Leaving out the attribute keyword means that you define an ordinary role (not functional).

Part names are defined in much the same way, although you cannot declare a part name to be functional.

(cl-define-primitive-part-name 'tyre-part)

There are different kinds of rules in CLASSIC but the syntax for defining them is quite similar, so we only give an example of one of them now.

(cl-add-rule 'document-rule @document '(fills page "13") :filter '(test-c cl-no-fillers? page))

We explain this syntax further. First the rule's name is "document-rule". The "@document" is a reference to the actual object with the name "document" and is the antecedent concept. The concept term following the ":filter" keyword is an additional test for the current individual that happened to be classified as a member of the document concept. In this case it is a call to an external test function that checks if the individual has no fillers for the page role. (This procedural test has to be defined before the rule is defined.) The consequence for the individual that triggered the rule and satisfied the procedural test is that it now has a filler for the page role, namely the string "13".

CLASSIC allows the user to state several queries about the knowledge base, by supplying user Lisp functions. One can check things about the is-a hierarchy or the part-of hierarchy or whether an individual is a member of some concept or similar questions. For a complete reference of the CLASSIC implementation that we use (without part handling) see [19], and for the part-of handling see [15].

Chapter 4

Modeling SGML Documents

4.1 Introduction

In the two previous chapters we have presented Description Logics and SGML separately. In this chapter we show that we can use Description Logics to represent SGML documents. That is, we can handle both DTDs and document instances.

It is the logical structure of documents that we want to model. The logical structure is a part hierarchy of elements. Thus it is not surprising that the part-of reasoning in Description Logics will come in handy. The basic ideas behind the modeling are the following:

- Element declarations can be seen as declarations of classes or types of logical parts of documents. Concepts in Description Logics are useful for modeling classes. Thus it seems natural to model element declarations using concepts.

- A logical part of a document instance can be seen as a member of the element classes that it satisfies, i.e. the classes that have a name matching the tag name and a content model that matches the part's logical parts. It therefore seems obvious that logical parts of document instances can be modeled by Description Logics individuals.

- SGML attributes are modeled by Description Logics attributes (functional roles).

- Default values for attributes are modeled by Description Logics rules that trigger if an individual is classified as the antecedent concept of the rule and fills the attribute for the individual with the default value unless the individual already has a filler for that attribute.

- Entities can be seen as parts of document instances that have been taken out and been replaced by references. Thus it seems natural to model entities in the same way as document parts, that is, using Description Logics individuals.

The purpose of modeling SGML documents by Description Logics, as mentioned earlier, was twofold. First we wanted to use Description Logics system's built in inference mechanisms to do the checking whether document instances conform to their DTDs. Second we wanted to store HTML documents in such a way that we could take advantage of the document's internal structure in an information retrieval application. The gain of this would be that we could use Description Logics techniques for machine learning and thus have very advanced information retrieval functionality.

This thesis is concerned with the correctness checking of SGML documents and thus the conversion of both DTDs and document instances to Description Logics. We will not be concerned with the document instances or DTDs after we have converted them to Description logics any more than doing the correctness checking, which is done automatically (the result is just reported on screen in the Description Logics system). We have implemented programs though that convert Description Logics versions of DTDs and document instances back to their earlier form.

We have chosen the Description Logics system CLASSIC($\triangleleft$) as our base system for several reasons. Most importantly it has been extended with part-of functionality which offers a nice way to deal with structured documents that have an internal logical part hierarchy. The CLASSIC($\triangleleft$) code is also available to the thesis writer. Furthermore the unextended CLASSIC system is currently the most efficient Description Logics system (see e.g. [2] or [13]).

The modeling of SGML documents is handled largely as follows. A program for handling DTDs has been written. It takes a file containing a DTD as input. It then scans the file and converts it part for part to Description Logics constructs and outputs a file containing these constructs. The output file can then be loaded directly into CLASSIC($\triangleleft$). A program for handling document instances has also been written. It also scans an input file containing a document instance and converts it part for part to Description Logics equivalents and finally outputs a file that can be loaded directly into CLASSIC($\triangleleft$). The correctness checking is then performed if you first load the Description Logics version of the DTD into CLASSIC($\triangleleft$)

and then load the Description Logics version of the instance. If no error messages appear and the Description Logics individual standing for the top element (in the element hierarchy) of the instance has been classified as belonging to the concept standing for the top element class of the DTD, the instance is correct relative the DTD.

In the first section below we discuss the major restrictions that we have made on the SGML syntax. After that we present in detail in two separate sections, how DTDs and instances are modeled. Then we proceed with some examples of DTDs and document instances and show how they are modeled.

4.2 Modeling restrictions

First of all, we do not model SGML functionality that is not used by HTML. This includes unique identifier attributes and content reference attributes. We now proceed to explain the further restrictions that we have imposed.

In the SGML declaration of HTML the keyword "OMITTAG" is set to "YES". This is also used in the HTML DTD, where some elements can have omitted start-tags or end-tags. We do not allow any omitting of start-tags or end-tags in the SGML documents that we model. If we had allowed this, the parsing of a document instance would have been dependent on the DTD. For example, to know that a logical element A ends because another element B starts, we must know the element A's content. Thus we would have to access the CLASSIC($\lhd$) Description Logics system and check element definitions every time a new element was started in the document instance. This would be inefficient. Another solution would be to consult the text file containing the DTD and check the element contents, or even better, maintain all element declarations directly in the conversion program. However, this would mean a large overhead for the system and we would loose the correctness checking functionality of the DL system.

For the same reason as above we require that even elements with empty contents have an end-tag. This is actually not allowed according to the SGML standard, but it really makes no difference, and many parsers (e.g. Netscape) allow end-tags of elements with empty contents.

We do not allow marked sections. This would also require access to the CLASSIC($\lhd$) system, to check entities. As marked sections are mainly used for convenience in large documents, not handling them does not influence the expressiveness of the languages we model.

Restrictions have been imposed on the type of regular expressions that are allowed. We have identified four important content types that are sufficient for modeling the content types used in the HTML DTD (unless we include the "deprecated" mode, which is only used very seldomly and only for backward compatibility reasons). This can be seen by consulting the HTML DTD in e.g. [3].

We do not allow recursive element hierarchies. That is, we do not allow an element to occur as a part of itself (on any level in the part hierarchy), since this would go against the CLASSIC($\lhd$) assumption of acyclic T-Boxes. Recursive element definitions do occur in the HTML DTD, but are mainly used in the highlighting elements and in forms and are thus not so important. We discuss this in more detail in the Appendix where we show how a very large part of HTML can be modeled.

Character references are not supported in the sense that we do not transform references to the corresponding character on screen. We simply ignore character references and interpret them as text. Our purpose is not to format and display documents and interpreting character references is really formatting.

Included and excluded elements are not allowed in element declarations. The method of including and excluding elements is used very sparsely in the HTML DTD, and it generally does not affect the correctness checking of HTML documents. The main idea of included and excluded elements is to give the ability to reuse elements in different places in the element hierarchy where their meaning is approximately the same, but they differ in some of the parts that are allowed or not allowed. Thus if we were to handle this in Description Logics the meaning of a concept would be dependent on its place in the concept hierarchy and that is not reasonable. We would then have to use different constructs for an element depending on its place in the hierarchy. However, this could just as well be handled by the person defining the DTD, since included and excluded elements give no further functionality, they are just syntactic sugar. Therefore we do not allow included and excluded elements.

We restrict the structure of DTDs. If an element A is referenced from the contents of some other element B then the declaration of B must be before the declaration of A. This is just a restriction imposed to make the scanning and parsing of DTDs easier and has no influence on the expressiveness of the language. It is also a structure that is often used in common DTDs, since it is a natural way of structuring a DTD and increases the readability.

The restrictions are recapitulated below:

- Functionality not used by HTML is not modeled.
- No tag minimization parameters are allowed in element declarations.

- All logical elements of a document instance (even elements with empty contents) must be separated by both start-tags and end-tags.
- No marked sections are allowed in DTDs.
- The kind of regular expressions that are allowed as element contents has been restricted.
- Recursive element hierarchies are not allowed.
- Character references are not processed.
- No included or excluded elements are allowed in element declarations.
- The structure of DTDs is restricted.

4.3 Modeling a DTD

This section contains a description and an explanation of how we model DTDs. Shortly, the logical hierarchy of element classes that are due to the element declarations and their contents are modeled by a concept hierarchy. The hierarchies are actually isomorphic. The concepts that build up the concept hierarchy all have restrictions that objects must satisfy to be members of the concepts. These restrictions are for example that the concept requires a filler of a specific kind for a specific attribute role or that it requires certain kind of parts and a certain order of the parts. These restrictions correspond to the attributes for the element and the contents of the element.

At this point it might be convenient to show what Description Logics constructs are implemented in the CLASSIC($\lhd$) system we will use. Figure 4.1 shows the syntax of the most important constructs in CLASSIC($\lhd$).

4.3.1 Element declarations

An example of what an element declaration might look like is as follows.

```
<!ELEMENT document (head,body)>
```

We thus have two things to model: the name and the content. The name is modeled by letting the concept have the same name as the element. Sometimes this is not enough though. For example, if we have two element declarations that have the same content and the same kind of attributes.

<concept-term> ::= $\top$ |
<concept-name> |
(**and** <concept-term>+) |
(**one-of** <individual-name>+) |
(**test-c** <fn> <arg>*) |
(**all** <role-term> <concept-term>) |
(**atleast** <positive-integer> <role-term>) |
(**atmost** <non-negative-integer> <role-term>) |
(**fills** <role-term> <individual-name>+) |
(**allp** <part-name-name> <concept-term>) |
(**atleastp** <positive-integer> <part-name-name>) |
(**atmostp** <non-negative-integer>
<part-name-name>) |
(**part-fills** <part-name-name>
<individual-name>+) |
(**pp-constraint** <role-name> <part-name-name>
<part-name-name>)

<individual-term> ::= <concept-term>
<role-term> ::= <role-name>

<concept-name> ::= <symbol>
<individual-name> ::= <symbol>
<role-name> ::= <symbol>
<part-name-name> ::= <symbol>

<fn> ::= a Common Lisp function that takes an individual, plus the extra arguments and returns T, ? or NIL.
<arg> ::= an expression passed to the test functions.

Figure 4.1: Syntax of the most important constructs of CLASSIC($\triangleleft$).

This might seem strange but is sometimes convenient. These two elements would be modeled by identical concepts with just different names. Since these two concepts would then be equivalent according to the model theory of Description Logics, CLASSIC($\lhd$) would not be able to distinguish between them. We want to assume that different names mean different elements, even if the content is the same. We solve this problem by adding a role "name" to each concept that has a filler of the type string and has as value the string of the name of the concept. For example in the case above we would have a restriction for the concept "document" that says: (**fills** name "document").

The content of an element is much more difficult to model. The content can be seen as a restriction of the logical parts that elements in instances are allowed to have. Here CLASSIC($\lhd$)'s open world assumption causes a difficulty. It is not enough just to find a way of modeling regular expressions and somehow connect that to parts. We also need a way to say that we cannot have any other parts than the ones we have mentioned. This problem also concerns attributes. SGML does not allow an element to have values for attributes that are not mentioned in the element declaration.

Modeling regular expressions

We need different part names for each element in a content. We use the convention that the part name takes the name of the element plus the suffix "-p". So given the content: (head , body), we introduce two part names "head-p" and "body-p". Further on we need to restrict the part fillers to be members of the corresponding concept. That is, for example, all part fillers of the part name "head-p" should belong to the concept "head". Thus, the concept with the content (head , body) should contain the restrictions: (**allp** head-p head) and (**allp** body-p body).

Modeling simple number restrictions on parts is no problem. We can use the **atmostp** and **atleastp** constructs. For example, modeling the content (p+) would use the following restrictions on the corresponding concept (**allp** p-p p) and (**atleastp** 1 p-p).

Modeling simple order restrictions on parts is easy as well. We use a special role "before" and the **pp-constraint** construct. Thus the content (head , body) would be modeled by the following restrictions: (**allp** head-p head), (**allp** body-p body), (**pp-constraint** before head-p body-p), (**atleastp** 1 head-p), (**atmostp** 1 head-p), (**atleastp** 1 body-p) and (**atmostp** 1 body-p).

Elements with empty contents get no restrictions at all except for the

(A , B , C , ...)

(**and** (**allp** A-p A) (**allp** B-p B)
  (**allp** C-p C) ...
  (**atleastp** 1 A-p) (**atmostp** 1 A-p)
  (**atleastp** 1 B-p) (**atmostp** 1 B-p)
  (**atleastp** 1 C-p) (**atmostp** 1 C-p) ...
  (**pp-constraint** before A-p B-p)
  (**pp-constraint** before B-p C-p) ...)

Figure 4.2: How a content of the type: (A , B , C , ...) is modeled in general.

"name" role mentioned above.

When the keyword "string" appears in the content of an element it is given special treatment. The corresponding part name is "string-p" in the usual manner. Instead of restricting the part fillers for "string-p" to be members of a concept "string", we call the concept "text". We do this because "string" is a built-in primitive concept in CLASSIC($\lhd$) whose members have no properties except their name. Using this built in concept "string" is not appropriate as "string" individuals are treated in a special manner. For example, they are not printed with the rest of the individuals when one uses the CLASSIC($\lhd$) function for printing all individuals. Thus we introduce a special concept that has the "string" individual as a filler for some role. Then the string appears explicitly as a role filler when printing the individual. We call this special concept "text" and define it to have only one restriction on its members, namely: (**atleast** 1 textstring).

Now we have enough power to model the content types in Table 4.1. These types are actually enough to model the content types that occur in the HTML DTD (if we stay without the optional "deprecated" mode, which basically only adds the ability of inserting plaintext after the body of an HTML document), as can be seen by inspecting all the element declarations in for example [3]. In Figures 4.2, 4.3, 4.4 and 4.5 we show how the different content types are modeled in general and in Figures 4.6, 4.7, 4.8 and 4.9 we give examples of how some real usage of the different content types in element declarations are modeled. Observe that the second and third content types are actually equivalent in expressivity. Both types are included just for convenience when modeling already existing DTDs.

content type	*restrictions*
(A , B , C , ...)	A, B, C and so on are elements or "string", and no element or "string" occurs twice.
(A' | B' | C' | ...)+	A, B, C and so on are elements or "string", and no element or "string" occurs twice. All ' should be one of *, +, ? or nothing, and atleast one ' should be a ? or a *.
(A' | B' | C' | ...)*	A, B, C and so on should be elements or "string, and no element or "string" occurs twice. All ' should be one of *, +, ? or nothing.
(A' & B' & C' & ...)	A, B, C and so on should be elements or "string, and no element or "string" occurs twice. All ' should be one of ? or nothing.

Table 4.1: Allowed content types.

(A' | B' | C' | ...)+

(**and** (**allp** A-p A)
    (**allp** B-p B) (**allp** C-p C) ...)

Figure 4.3: How a content of the type: (A' | B' | C' | ...)+ is modeled in general.

(A' | B' | C' | ...)*

(**and** (**allp** A-p A)
    (**allp** B-p B) (**allp** C-p C) ...)

Figure 4.4: How a content of the type: (A' | B' | C' | ...)* is modeled in general.

(A' & B' & C' & ...)

(**and** (**allp** A-p A) (**allp** B-p B)
  (**allp** C-p C) ...
  '-CODE A
  '-CODE B
  '-CODE C ...)

Figure 4.5: How a content of the type: (A' & B' & C' & ...) is modeled in general. Observe that '-CODE X stands for the DL code that restricts the number of fillers of type X. For example: if A'=A+, the '-CODE A = (**atleastp** 1 A-p).

<!ELEMENT document (head , body , appendix)>

(cl-define-concept 'document '(**and** (**fills** name "document")
  (**allp** head-p head) (**allp** body-p body)
  (**allp** appendix-p appendix) (**atleastp** 1 head-p)
  (**atmostp** 1 head-p) (**atleastp** 1 body-p)
  (**atmostp** 1 body-p) (**atleastp** 1 appendix-p)
  (**atmostp** 1 appendix-p) (**pp-constraint** before head-p body-p)
  (**pp-constraint** before body-p appendix-p)))

Figure 4.6: An example of how a content of the type: (A , B , C , ...) is modeled.

<!ELEMENT body (p | xmp* | fig?)+>

(cl-define-concept 'body '(**and** (**fills** name "body") (**allp** p-p p)
  (**allp** xmp-p xmp) (**allp** fig-p fig)))

Figure 4.7: An example of how a content of the type: (A' | B' | C' | ...)+ is modeled.

<!ELEMENT p (string | hr?)*>

(cl-define-concept 'p '(**and** (**fills** name "p") (**allp** string-p text)
    (**allp** hr-p hr)))

Figure 4.8: An example of how a content of the type: (A' | B' | C' | ...)* is modeled.

<!ELEMENT body (chapters & annexes & appendix?)>

(cl-define-concept 'body '(**and** (**fills** name "body")
    (**allp** chapters-p chapters) (**allp** annexes-p annexes)
    (**allp** appendix-p appendix) (**atleastp** 1 chapters-p)
    (**atmostp** 1 chapters-p) (**atleastp** 1 annexes-p)
    (**atmostp** 1 annexes-p) (**atmostp** 1 appendix-p)))

Figure 4.9: An example of how a content of the type: (A' & B' & C' & ...) is modeled.

Handling the open world assumption

Let us return to the problem that we mentioned above about CLASSIC($\lhd$)'s open world assumption. We need a way to state a requirement that individuals belonging to a concept must not have part fillers or role fillers for parts or roles other than the ones mentioned in the definition of the concept. We solve this problem by introducing two rules for each concept. One rule for parts and one rule for roles. The basic idea is that the rules are set to trigger for individuals that are classified as members of the concept and the individuals are then filtered through a **test-c** function that checks if the individual has restrictions on parts (or roles) other than the ones mentioned in the definition of the antecedent concept. The result of having an individual filtered through that test is a warning message on screen stating that the individual has unallowed parts (or roles). For each concept we thus have to define the test functions as well.

We describe more precisely how the rule and the corresponding test function for controlling unallowed parts work. We postpone descriptions of how the rules and test functions work for roles until we have introduced our modeling of roles in the next section, and for now it suffices to say that they work in a very similar way.

We have mentioned earlier how rules work, but we give a short repetition here. A rule triggers when an individual is classified as belonging to the rule's antecedent concept. The triggering individual is then passed through an optional filter which tests additional aspects of the individual. Finally, if the individual passes the filter some action is taken. This action should eventually return some additional properties that this particular individual should have and which will be merged with the other properties of the individual. In our case the resulting action is a call to a lisp function that, as a side effect, outputs a warning message on screen and returns the CLASSIC top concept "CLASSIC-THING", which does not change the individual at all when merged with the other properties according to the semantics of the **and** construct.

The filter uses (as mentioned above) a **test-c** function. This function is a Lisp function and uses the CLASSIC($\lhd$) user function "cl-part-name-restrs" to get the list of part names that are restricted for the individual. It then uses the user function "cl-at-mostp" to remove each part from the list that has a restriction of at most 0 part fillers. It then checks if there are individuals on the list that were not mentioned in the concept definition. If there were unallowed individuals, T is returned, otherwise NIL is returned.

In Figure 4.10 we present an example of an element declaration and the warning function, the test function and the rule that are generated from the element declaration.

Observe that "@c{document}" is a way of referring to the concept "document" where "c" stands for concept. "@pn{HEAD-P}" is a reference to the part name "head-p" and "pn" stands for part name. Similarly we can reference individuals and roles by changing the "c" or "pn" to an "i" or "r". The rule "document-part-rule" defined above triggers for individuals that belong to the concept "document". Such an individual is passed through the test function "document-unallowed-parts?" which checks if the individuals has restrictions on parts other than "HEAD-p" and "BODY-p". If it has restrictions on unallowed parts T is returned, otherwise NIL is returned. If T is returned the function "generate-part-warning" is called which generates a warning output on screen saying that the individual has unallowed parts.

4.3.2 Attribute declarations

A typical attribute declaration might look like this:

<!ATTLIST document version (draft | finished) draft>

```
<!ELEMENT document (head, body)>

(defun generate-part-warning (ind)
  (format t "~&Error: The individual ~a has unallowed roles!" ind)
  'CLASSIC-THING)

(defun document-unallowed-parts? (ind)
  (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil)
      (if (not (equal (cl-at-mostp ind el) 0))
        (if (eq (member (format nil "~a" el)
              '("@pn{HEAD-P}" "@pn{BODY-P}") :test #'equal) nil)
          (return t))))))

(cl-add-description-rule 'document-part-rule @c{document}
  'generate-part-warning :filter '(test-c document-unallowed-parts?))
```

Figure 4.10: An example of how we handle unallowed parts.

The attribute's name is "version" and the corresponding element is "document" and the possible values are "draft" and "finished" and the default value is "draft". An attribute corresponds in Description Logics to a role with the same name declared as an attribute role. An attribute declaration for some element is modeled by restricting the role fillers of the role corresponding to the attribute for the concept corresponding to the element.

Allowed values

To handle the allowed values of an attribute we restrict the possible fillers for the role for the concept using the **one-of** construct. In the example above we would include the restriction: (**all** version (**one-of** draft finished)) in the definition of the concept "document". On the other hand if we had an attribute declaration whose possible values were declared as string we would model this by the restriction: (**all** version string).

Default values

Before we describe how general default values are managed we describe how the #REQUIRED and the #FIXED keywords are handled as defaults. The

```
<!ATTLIST document version (draft | finished) draft>

(defun cl-no-fillers? (ind role-name)
  (let ((role (cl-named-role role-name)))
    (not (cl-fillers ind role))))

(cl-add-rule 'document-rule1
  @c{document}
  '(and document (fills version draft))
  :filter '(test-c cl-no-fillers? version))
```

Figure 4.11: An example of how a default value in an attribute declaration is handled.

keyword #REQUIRED is handled by including a number restriction on the role. For example, if we have the following attribute declaration:

<!ATTLIST document version (draft | finished) #REQUIRED>

we impose the restriction: (**atleast** 1 version) on the concept "document".

The keyword #FIXED is always followed by a value from the allowed values. For instance, if we have the following attribute declaration:

<!ATTLIST document version (draft | finished) #FIXED draft>

we impose the restriction: (**fills** version draft) on the concept "document".

Default values are as mentioned handled by rules. The antecedent concept is the concept corresponding to the element that the attribute is defined for. The filter is a test function "cl-no-fillers?" that checks whether an individual has no fillers for the role corresponding to the attribute. The result that occurs if an individual passes the filter is that it is filled by the default value for the role corresponding to the attribute. In Figure 4.11 we present an example of how a default rule and test function might look like for an attribute declaration containing a default value.

The test function "cl-no-fillers?" needs only be defined once and for all for each DTD. The keyword #IMPLIED is the only thing that remains to be modeled. The keyword means that the system should choose the default value on its own. A natural way of modeling this is in the same way as

```
<!ATTLIST document version string #IMPLIED>

(defun cl-no-fillers? (ind role-name)
   (let ((role (cl-named-role role-name)))
      (not (cl-fillers ind role))))

(cl-add-rule 'document-rule1
   @c{document}
   '(and document (fills version "arne-anka"))
   :filter '(test-c cl-no-fillers? version))
```

Figure 4.12: An example of how an implied default value in an attribute declaration is handled.

for ordinary default values and just selecting the default value according to some algorithm, like always select the first enumerated allowed value or a standard string if the allowed value declaration is string. We present an example of this in Figure 4.12.

Handling the open world assumption

As was the case for parts when modeling element contents we need a way to specify that a concept corresponding to some element is not allowed to have restrictions on roles other than the ones corresponding to the attributes that are defined for the element. We handled this for parts by introducing a rule that filtered out individuals that had unallowed parts by giving an error message. We use the same technique for attributes and demonstrate this by an example presented in Figure 4.13.

Observe that the "name" role must always be allowed. That is no problem since it is declared as an attribute and is always given a filler for each individual, thus an individual cannot be given extra fillers for that role without having CLASSIC reporting an error. The "before" role is also allowed at all times. When compiling a document instance to the Description Logics equivalent we do not know before hand if the order of the element instances is important. Therefore the use of the "before" role is to describe the order of element instances and we have to allow the use of the "before" role at all times.

```
<!ELEMENT document (head , body)>

<!ATTLIST document version (draft | finished) draft>

<!ATTLIST document author string #REQUIRED>

(defun generate-role-warning (ind)
  (format t "~&Error: The individual ~a has unallowed roles!" ind)
  'CLASSIC-THING)

(defun document-unallowed-roles? (ind)
  (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil)
      (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{NAME}" "@r{BEFORE}"
            "@r{VERSION}" "@r{AUTHOR}")
            :test #'equal) nil)
        (return t))))))

(cl-add-description-rule 'document-role-rule @c{document}
  'generate-role-warning :filter '(test-c document-unallowed-roles?))
```

Figure 4.13: An example of how we handle unallowed roles.

<!ENTITY sgml "Standard Generalized Markup Language">

```
(cl-create-ind 'sgml '(and
   (fills textstring
      "Standard Generalized Markup Language")
   (fills name "document-entity")))
```

Figure 4.14: An example of how an entity declaration is modeled.

4.3.3 Entity declarations

An example of an entity declaration might be as follows:

<!ENTITY sgml "Standard Generalized Markup Language">

Entities are modeled by individuals that belong to the concept "text" and thus have a filler for the role "textstring". We also require a filler for the role "name". The value of an entity is always a string which is modeled as a filler for the "textstring" role. The reason for having a name filler for an entity is that we want to separate entities that are declared in different DTDs. The naming convention that we use is that we give a name corresponding to the top element of a DTD. For example, if the top element is "document", then we give all entities declared in that DTD the name "document-entity". We present an example of how an entity declaration is modeled in Figure 4.14.

4.4 Modeling a document instance

This section shows how document instances are modeled. Briefly, the logical elements of a document instance are separated by their start-tags and end-tags. Each logical element is converted to a Description Logics individual with restrictions according to its logical parts and its attributes.

4.4.1 Element instances

Modeling an element instance requires modeling of the element name and the contents and possibly attribute values. We present an example of an element instance in Figure 4.15.

One can distinguish between three kinds of element instances:

- Element instances that are strings, without any start-tags or end-tags.

```
.
.
.
<body numbering=arabic>
<p>
This is the first paragraph in this document.
</p>
<hr height="5"></hr>
<xmp> This is an example of an entity reference:
Did you know that SGML stands for &sgml;? </xmp>
This is an ending string.
</body>
.
.
.
```

Figure 4.15: An example of what a typical element instance might look like.

```
(cl-create-ind 'string1 '(and
   (fills textstring "This is an ending string")))
```

Figure 4.16: An example of how a string instance is modeled.

- Element instances that have element instances as content.
- Element instances with empty content.

Common for all three cases is that the instances are modeled by individuals. Instances that are strings are given an individual name "stringX", where X is some unique number used to distinguish different individuals modeling element instances with the same name. A string individual is restricted to be a member of the "text" concept and thus to have a filler for the role "textstring". It can also have a filler for the "before" role to describe its place in the logical hierarchy. For example, the element instance "This is an ending string" in the example presented earlier would be modeled as in Figure 4.16.

Observe that no filler for the "before" role is needed since no element instance follows this one.

Element instances with contents are modeled using the **part-fills** con-

```
(cl-create-ind 'p1 '(and
    (fills name "p")
    (part-fills string-p textp2)
    (fills before hr3)))
```

Figure 4.17: An example of how a string instance is modeled.

struct. A content is also an element instance and thus modeled by an individual as well. For example, the "p" part in the earlier example instance could be modeled as displayed in Figure 4.17. In this case the reference to the individual "textp2" refers to the string individual modeling the string instance: "This is the first paragraph in this document.". "hr3" refers to the individual representing the "hr" element instance that follows the "p" instance in the example.

An element instance can have more than one logical part. This is modeled by having several **part-fills** restrictions. An element instance with empty contents is modeled in the obvious way without any **part-fills** restrictions. An element instance can also have attribute values. This is modeled by the **fills** construct.

CLASSIC($\lhd$) requires that individuals are defined before they are referred to (except for string individuals). Thus we have to define the children of a node in the logical element instance hierarchy before the node itself.

In Figure 4.18 we present how the example instance from Figure 4.15 could be represented in our model.

Handling the open world assumption

CLASSIC($\lhd$)'s open world assumption has the following consequence for the modeling of document instances. To be able to check that number restrictions on parts hold for an individual we have to close all part names that are restricted in the definition of the concept that the individual is a candidate for being member of. If we did not close a part name that had one filler for some individual we could not deduce that it had exactly one filler for that part name. After an individual is defined a call to the function "close-concept-restricted-part-names-for-ind" with the individual as argument is done. The function uses the CLASSIC($\lhd$) user function "cl-part-name-restrs" to get a list of the part names that are restricted in the definition of the concept that has the same name as the individual. It then uses the function "cl-ind-close-part-name" to close each part name on the

```
(cl-create-ind 'string1 '(and
    (fills textstring "This is an ending string")))

(cl-create-ind 'textp2 '(and
    (fills textstring "This is an example of an entity reference:
        Did you know that SGML stands for &sgml;?")))

(cl-create-ind 'xmp3 '(and
    (fills name "xmp")
    (part-fills string-p textp2)
    (fills before string1)))

(cl-create-ind 'hr4 '(and (fills name "hr")
    (fills before xmp3)
    (fills height "5")))

(cl-create-ind 'textp5 '(and
    (fills textstring "This is the first paragraph in this document.")))

(cl-create-ind 'p6 '(and
    (fills name "p")
    (part-fills string-p textp5)
    (fills before hr4)))

(cl-create-ind 'body7 '(and
    (fills name "body")
    (fills numbering arabic)
    (part-fills p-p p6)
    (part-fills hr-p hr4)
    (part-fills xmp-p xmp3)
    (part-fills string-p string1)))
```

Figure 4.18: An example of how a previous instance example is modeled.

```
(defun close-concept-restricted-part-names-for-ind (part-list ind)
   (dolist (el part-list nil)
      (cl-ind-close-part-name ind el)))
```

Figure 4.19: The definition of the close function.

```
(cl-create-ind 'p6 '(and
   (fills name "p")
   (part-fills string-p textp5)
   (fills before hr4)))

(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{p}) @i{p6})
```

Figure 4.20: An example of how the close function is called after an individual definition.

list. The close function is usually defined in a DTD. We present the close function in Figure 4.19. In Figure 4.20 we give an example of an individual definition and the corresponding call to the close function.

4.4.2 Entity references

Entity references in a document instance are left without any action taken. We leave the work for the conversion routine that converts the Description Logics representation of a document instance back to a document instance. The conversion program exchanges all entity references that refer to entities that are defined in the Description Logics system with the corresponding string. Entity references that refer to unknown entities are left as they are.

4.5 Examples

In this section we present a larger example of a DTD with corresponding document instance and how they both are modeled. The following declarations define an example DTD:

```
<!ENTITY % values "(draft | finished)">
<!ENTITY sgml "Standard Generalized Markup Language">
```

```
<!ELEMENT document (head , body)>
<!ATTLIST document author string "Johan Åberg">
<!ATTLIST document version %values; draft>

<!ELEMENT head (title , abstract)>
<!ATTLIST head numbering (roman | arabic) roman>

<!ELEMENT title (string)>

<!ELEMENT abstract (string)>

<!ELEMENT body (string | p | xmp | hr)*>
<!ATTLIST body numbering (roman | arabic) arabic>

<!ELEMENT hr EMPTY>
<!ATTLIST hr height string "2">

<!ELEMENT p (string | em)*>

<!ELEMENT xmp (string | em)*>

<!ELEMENT em (string)>
```

This DTD is modeled by the following sequence of CLASSIC(⊲) definitions:

```
(defun close-concept-restricted-part-names-for-ind (part-list ind)
   (dolist (el part-list nil)
      (cl-ind-close-part-name ind el)))

(defun cl-no-fillers? (ind role-name)
   (let ((role (cl-named-role role-name)))
      (not (cl-fillers ind role))))

(defun generate-part-warning (ind)
   (format t "~&Error: The individual ~a has unallowed parts!" ind)
   'CLASSIC-THING)
```

```
(defun generate-role-warning (ind)
   (format t "~&Error: The individual ~a has unallowed roles!" ind)
   'CLASSIC-THING)

(cl-define-primitive-role 'height :attribute t)
(cl-define-primitive-role 'numbering :attribute t)
(cl-define-primitive-role 'version :attribute t)
(cl-define-primitive-role 'author :attribute t)
(cl-define-primitive-role 'textstring :attribute t)
(cl-define-primitive-role 'before :attribute t)
(cl-define-primitive-role 'name :attribute t)

(cl-define-primitive-part-name 'head-p)
(cl-define-primitive-part-name 'body-p)
(cl-define-primitive-part-name 'title-p)
(cl-define-primitive-part-name 'abstract-p)
(cl-define-primitive-part-name 'p-p)
(cl-define-primitive-part-name 'xmp-p)
(cl-define-primitive-part-name 'hr-p)
(cl-define-primitive-part-name 'em-p)
(cl-define-primitive-part-name 'string-p)

(cl-define-concept 'text '(and (atleast 1 textstring)))

(cl-create-ind 'sgml '(and
   (fills textstring "Standard Generalized Markup Language")
   (fills name "document-entity")))

(cl-define-concept 'em '(and (allp string-p text)
   (atleastp 1 string-p) (atmostp 1 string-p) (fills name "em")))

(defun em-unallowed-roles? (ind)
   (let ((role-list (cl-role-restrs ind)))
      (dolist (el role-list nil)
         (if (not (equal (cl-at-most ind el) 0))
            (if (eq (member (format nil "~a" el)
               '("@r{NAME}" "@r{BEFORE}")
            :test #'equal) nil) (return t))))))
```

```
(cl-add-description-rule 'em-role-rule @c{em}
    'generate-role-warning
    :filter '(test-c em-unallowed-roles?))

(defun em-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@pn{STRING-P}") :test #'equal) nil)
                (return t))))))

(cl-add-description-rule 'em-part-rule @c{em}
    'generate-part-warning
    :filter '(test-c em-unallowed-parts?))

(cl-define-concept 'xmp '(and (allp string-p text) (allp em-p em)
    (pp-constraint before string-p em-p) (fills name "xmp")))

(defun xmp-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil)
            (if (not (equal (cl-at-most ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@r{NAME}" "@r{BEFORE}") :test #'equal) nil)
                (return t))))))

(cl-add-description-rule 'xmp-role-rule @c{xmp}
    'generate-role-warning
    :filter '(test-c xmp-unallowed-roles?))

(defun xmp-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@pn{STRING-P}" "@pn{EM-P}") :test #'equal) nil)
                    (return t))))))

(cl-add-description-rule 'xmp-part-rule @c{xmp}
```

```
    'generate-part-warning
    :filter '(test-c xmp-unallowed-parts?))

(cl-define-concept 'p '(and (allp string-p text)
    (atleastp 1 string-p) (atmostp 1 string-p)
    (allp em-p em) (atleastp 1 em-p) (atmostp 1 em-p)
    (pp-constraint before string-p em-p) (fills name "p")))

(defun p-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil)
            (if (not (equal (cl-at-most ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@r{NAME}" "@r{BEFORE}") :test #'equal) nil)
                (return t))))))

(cl-add-description-rule 'p-role-rule @c{p}
    'generate-role-warning
    :filter '(test-c p-unallowed-roles?))

(defun p-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@pn{STRING-P}" "@pn{EM-P}") :test #'equal) nil)
                (return t))))))

(cl-add-description-rule 'p-part-rule @c{p}
    'generate-part-warning
    :filter '(test-c p-unallowed-parts?))

(cl-define-concept 'hr '(and (fills name "hr")
    (all height string)))

(cl-add-rule 'hr-rule1 @c{hr} '(and hr
    (fills height "2"))
    :filter '(test-c cl-no-fillers? height))

(cl-define-concept 'body '(and (allp p-p p)
```

```
    (allp xmp-p xmp) (allp hr-p hr) (allp string-p text)
    (all numbering (one-of roman arabic)) (fills name “body”)))

(cl-add-rule 'body-rule2 @c{body} '(and body
    (fills numbering arabic))
    :filter '(test-c cl-no-fillers? numbering))

(defun body-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil)
            (if (not (equal (cl-at-most ind el) 0))
                (if (eq (member (format nil “~a” el)
                    '(“@r{NAME}” “@r{BEFORE}” “@r{NUMBERING}”)
                    :test #'equal) nil) (return t))))))

(cl-add-description-rule 'body-role-rule @c{body}
    'generate-role-warning
    :filter '(test-c body-unallowed-roles?))

(defun body-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
                (if (eq (member (format nil “~a” el)
                    '(“@pn{P-P}” “@pn{XMP-P}”
                    “@pn{HR-P}” “@pn{STRING-P}”)
                    :test #'equal) nil) (return t))))))

(cl-add-description-rule 'body-part-rule @c{body}
    'generate-part-warning
    :filter '(test-c body-unallowed-parts?))

(cl-define-concept 'abstract '(and (allp string-p text)
    (atleastp 1 string-p) (atmostp 1 string-p)
    (fills name “abstract”)))

(defun abstract-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil)
            (if (not (equal (cl-at-most ind el) 0))
```

```
            (if (eq (member (format nil "~a" el)
                '("@r{NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))

(cl-add-description-rule 'abstract-role-rule @c{abstract}
    'generate-role-warning
    :filter '(test-c abstract-unallowed-roles?))

(defun abstract-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@pn{STRING-P}") :test #'equal) nil)
                (return t))))))

(cl-add-description-rule 'abstract-part-rule @c{abstract}
    'generate-part-warning
    :filter '(test-c abstract-unallowed-parts?))

(cl-define-concept 'title '(and (allp string-p text)
    (atleastp 1 string-p) (atmostp 1 string-p)
    (fills name "title")))

(defun title-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil)
            (if (not (equal (cl-at-most ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@r{NAME}" "@r{BEFORE}") :test #'equal) nil)
                (return t))))))

(cl-add-description-rule 'title-role-rule @c{title}
    'generate-role-warning
    :filter '(test-c title-unallowed-roles?))

(defun title-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
```

```
        (if (eq (member (format nil "~a" el)
          '("@pn{STRING-P}") :test #'equal) nil)
        (return t))))))

(cl-add-description-rule 'title-part-rule @c{title}
  'generate-part-warning
  :filter '(test-c title-unallowed-parts?))

(cl-define-concept 'head '(and (allp title-p title)
  (atleastp 1 title-p) (atmostp 1 title-p)
  (allp abstract-p abstract) (atleastp 1 abstract-p)
  (atmostp 1 abstract-p)
  (pp-constraint before title-p abstract-p)
  (all numbering (one-of roman arabic)) (fills name "head")))

(cl-add-rule 'head-rule3 @c{head} '(and head
  (fills numbering roman))
  :filter '(test-c cl-no-fillers? numbering))

(defun head-unallowed-roles? (ind)
  (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil)
      (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
          '("@r{NAME}" "@r{BEFORE}" "@r{NUMBERING}")
            :test #'equal) nil) (return t))))))

(cl-add-description-rule 'head-role-rule @c{head}
  'generate-role-warning
  :filter '(test-c head-unallowed-roles?))

(defun head-unallowed-parts? (ind)
(let ((part-list (cl-part-name-restrs ind)))
  (dolist (el part-list nil)
    (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
        '("@pn{TITLE-P}" "@pn{ABSTRACT-P}")
          :test #'equal) nil) (return t))))))

(cl-add-description-rule 'head-part-rule @c{head}
```

```
    'generate-part-warning
    :filter '(test-c head-unallowed-parts?))

(cl-define-concept 'document '(and (allp head-p head)
    (atleastp 1 head-p) (atmostp 1 head-p)
    (allp body-p body) (atleastp 1 body-p) (atmostp 1 body-p)
    (pp-constraint before head-p body-p)
    (all version (one-of draft finished))
    (all author string) (fills name "document")))

(cl-add-rule 'document-rule4 @c{document} '(and document
    (fills version draft)) :
    filter '(test-c cl-no-fillers? version))

(cl-add-rule 'document-rule5 @c{document} '(and document
    (fills author "Johan Åberg"))
    :filter '(test-c cl-no-fillers? author))

(defun document-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil)
            (if (not (equal (cl-at-most ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@r{NAME}" "@r{BEFORE}" "@r{VERSION}"
                    "@r{AUTHOR}")
                      :test #'equal) nil) (return t))))))

(cl-add-description-rule 'document-role-rule @c{document}
    'generate-role-warning
    :filter '(test-c document-unallowed-roles?))

(defun document-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil)
            (if (not (equal (cl-at-mostp ind el) 0))
                (if (eq (member (format nil "~a" el)
                    '("@pn{HEAD-P}" "@pn{BODY-P}") :test #'equal) nil)
                (return t))))))
```

```
(cl-add-description-rule 'document-part-rule @c{document}
    'generate-part-warning
    :filter '(test-c document-unallowed-parts?))
```

We now present an example instance that satisfies the restrictions of the previous DTD:

```
<document author="Johan Åberg" version="finished">

<head>
<title> Structured Documents and Description Logics
</title>
<abstract> It is shown how Description Logics can be used to model
structured documents in the form of SGML documents.
</abstract>
</head>

<body numbering=arabic>
<p>
This is the first paragraph in this document.
</p>
<p>
This is the second paragraph.
</p>
<hr height="5"></hr>
<p> This is yet <em>another</em> paragraph.</p>
This is just a string mixed among the paragraphs and the examples.
<xmp> This is an example of an entity reference:
Did you know that SGML stands for &sgml;? </xmp>
This is an ending string.
</body>
</document>
```

The following is the CLASSIC($\triangleleft$) representation of the instance above:

```
(cl-create-ind 'string15 '(and
```

```
    (fills textstring “This is an ending string. ”)))

(cl-create-ind ’textxmp14 ’(and
    (fills textstring “This is an example of an entity reference:
        Did you know that SGML stands for &sgml;? ”)))

(cl-create-ind ’xmp14 ’(and (partfills string-p textxmp14)
    (fills before string15) (fills name “xmp”)))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{xmp} ) @i{xmp14})

(cl-create-ind ’string13 ’(and
    (fills textstring “This is just a string mixed among the
        paragraphs and the examples. ”) (fills before xmp14)))

(cl-create-ind ’string12 ’(and (fills textstring “paragraph.”)))

(cl-create-ind ’textem11 ’(and (fills textstring “another”)))

(cl-create-ind ’em11 ’(and (partfills string-p textem11)
    (fills before string12) (fills name “em”)))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{em}) @i{em11})

(cl-create-ind ’string10 ’(and
    (fills textstring “This is yet ”) (fills before em11)))

(cl-create-ind ’p9 ’(and (partfills string-p string12)
    (partfills em-p em11) (partfills string-p string10)
    (fills before string13) (fills name “p”)))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p9})

(cl-create-ind ’hr8 ’(and (fills height “5”)
    (fills before p9) (fills name “hr”)))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{hr}) @i{hr8})

(cl-create-ind ’textp7 ’(and
    (fills textstring “This is the second paragraph.”)))
```

```
(cl-create-ind 'p7 '(and (partfills string-p textp7)
   (fills before hr8) (fills name "p")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{p}) @i{p7})

(cl-create-ind 'textp6 '(and
   (fills textstring "This is the first paragraph in this document.")))

(cl-create-ind 'p6 '(and (partfills string-p textp6)
   (fills before p7) (fills name "p")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{p}) @i{p6})

(cl-create-ind 'body5 '(and (partfills string-p string15)
   (partfills xmp-p xmp14) (partfills string-p string13)
   (partfills p-p p9) (partfills hr-p hr8) (partfills p-p p7)
   (partfills p-p p6) (fills numbering arabic) (fills name "body")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{body}) @i{body5})

(cl-create-ind 'textabstract4 '(and
   (fills textstring "It is showed how Description Logics
      can be used to modelstructured documents in the
      form of SGML documents.")))

(cl-create-ind 'abstract4 '(and (partfills string-p textabstract4)
   (fills name "abstract")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{abstract}) @i{abstract4})

(cl-create-ind 'texttitle3 '(and
   (fills textstring "Structured Documents and Description Logics ")))

(cl-create-ind 'title3 '(and (partfills string-p texttitle3)
   (fills before abstract4) (fills name "title")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{title}) @i{title3})

(cl-create-ind 'head2 '(and (partfills abstract-p abstract4)
```

```
    (partfills title-p title3) (fills before body5) (fills name "head")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{head}) @i{head2})

(cl-create-ind 'document1 '(and (partfills body-p body5)
    (partfills head-p head2) (fills author "Johan Åberg")
    (fills version "finished") (fills name "document")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{document}) @i{document1})
```

Chapter 5

Notes on the Implementation

5.1 Introduction

In this chapter we give some notes on the implementation of the different conversion functions for modeling DTDs and document instances by Description Logics. All code was written in Common Lisp using the Allegro 4.2 Lisp system.

5.2 Modeling DTDs

5.2.1 From DTDs to concepts

In the file "convert-dtd.lisp" the functions for converting a DTD to its Description Logics representation are located. The main function "convert-dtd" takes two filenames as input. The first filename should be the name of the file that contains the DTD to be converted and the second filename should be the name of the file on which the Description Logics representation of the DTD will be printed. The DTD is supposed to be a syntactically correct DTD although the functions do some error checking.

Comments are allowed between declarations according to the syntax presented in the chapter on structured documents. Comments are ignored in the conversion. The order of element declarations is important as pointed out in the section on restrictions in the modeling chapter. Entity declarations as well as attribute declarations can be positioned in any order.

The "convert-dtd" function works as follows. It first scans the file containing the DTD and stores element declarations, parameter entity declarations, entity declarations and attribute declarations in global lists. It then proceeds to convert the scanned data to Description Logics representations. First the predefined functions "close-concept-restricted-part-names-for-ind", "cl-no-fillers?", "generate-part-warning" and "generate-role-warning" are written to the output file. Then declarations of roles for each attribute used in the DTD are written. Each role is declared as an attribute role. Then the special roles "textstring", "before" and "/name" are declared (also as attributes). (The role "name" got the name "/name" instead since for instance HTML uses an attribute with the name "name".) Thereafter part names are defined according to the scanned contents in element declarations. Then the special concept "text" is defined.

The conversion function then continues to write the Description Logics representation of the scanned entities. Then, **test-c** functions are defined to test role restrictions (modeling attributes) of concepts (modeling elements). For example, if an element has an attribute "height" which takes string values defined for it, then the test (**all** height string) is performed by a **test-c** call in the element definition and not directly which is the obvious way to model that restriction. Apart from that exception, element declarations are modeled by concept definitions and the rules for hanDescription Logicsing default values and checking for unallowed parts are written after each concept definition.

An example of what a converted DTD might look like is presented in the Appendix where a (somewhat restricted) version of the HTML DTD is converted.

After a successful conversion, the resulting output file can be loaded into CLASSIC($\lhd$) with the command "(load "filename")". Each declaration in the file is then read and evaluated into CLASSIC($\lhd$).

5.2.2 From concepts to DTDs

The file "convert-concepts-to-dtd.lisp" contains the functions that given a concept (should be the "top" concept of the concepts modeling a DTD) and a filename (should be the filename where the resulting DTD should be located) converts the Description Logics representation of a DTD back to a DTD.

The main routine, called "convert-concepts-to-dtd", first converts all individuals representing entities for the DTD to entity declarations. Then, for the input concept and for each concept below the input concept in the part hierarchy the element declarations and corresponding attribute declarations are written so that the element order satisfies the order restrictions defined in the modeling chapter.

To use the conversion function, the file "convert-concepts-to-dtd.lisp" has to be loaded into CLASSIC($\triangleleft$). Then if the concepts are defined, the function can be called, resulting in a DTD corresponding to the Description Logics representation.

5.3 Modeling document instances

5.3.1 From structured documents to individuals

The file "convert-instance.lisp" contains the functions that hanDescription Logicse the conversion of document instances to Description Logics representations of document instances. The main routine is called "convert-instance" and takes three filenames as input. The first name should point to the file containing the document instance, the second to the file that should contain the resulting Description Logics definitions and the third should point to a file containing a number that will be used to postfix all individuals created. For example, the first time a Description Logics representation of a document instance for some DTD is to be created, the file could contain "0". The individuals will then be postfixed with "1" followed by "2" and so on. The last number to postfix an individual will be written back to the file, for use the next time a document for the DTD should be converted. The reason for this is that we want to be able to store several different documents for some DTD at the same time in CLASSIC($\triangleleft$). Thus we have to be able to give individuals different names.

The conversion functions require a syntactically correct document instance although some error checking is done. Note that, for parsing reasons, comments are only allowed directly between two tags. They are not allowed to be mixed with text.

A call to the "convert-instance" function with the proper filenames, results in a tree (represented by a list) containing the document instance. This tree is then traversed resulting in the writing of individual definitions on the output file. The individual definitions are written as described in the modeling chapter.

The resulting file can then be read into CLASSIC($\triangleleft$). This should be done after the Description Logics representation of the corresponding DTD has been read into CLASSIC($\triangleleft$). If the document instance were a correct instance of the DTD after reading the file into CLASSIC($\triangleleft$) all individuals should belong to the corresponding concepts and no error messages should have been printed on screen.

Observe how different document instances can be loaded into CLASSIC($\triangleleft$)

and then, using CLASSIC($\lhd$)'s built in query facilities, queried taking advantage of the document's structure. Note that to be able to hanDescription Logicse the Description Logics representation of text, special functions have to be created to process the Lisp strings and interact with CLASSIC($\lhd$), but this could be done without much trouble. This work though, is part of further extensions to fit this thesis work into the larger information retrieval project mentioned in the introduction chapter.

For an example of how real document instances are converted consult the Appendix which shows how two real HTML documents are converted.

5.3.2 From individuals to structured documents

The file "convert-inds-to-instance.lisp"contains all functions for converting a Description Logics representation of a document instance back to the document instance.

The main function is called "convert-inds-to-instance" and takes a Description Logics individual (should be the "top" individual among the individuals describing the document instance) and a filename (the filename should point to the file where the document instance should be written) and converts each individual in the part hierarchy starting with the given individual to the matching start-tags and end-tags and strings. This is done in a way that preserves the order of the document instance.

All references to entities in the strings contained in the individuals are checked against the ones defined in the DTD and replaced by the value of the entity should it exist. If it does not exist the entity reference is left without any action taken.

The postfix numbers that the individuals have are discarded and only their real names are used in the resulting document instance. Observe that because of this no numbers are allowed in element names. We have made an exception though with the HTML elements H1 to H6 which we allow.

The conversion can be performed first after the "convert-inds-to-instance.lisp" file is loaded into CLASSIC($\lhd$). Supposing that the file containing the Description Logics representation of the document instance has been read into CLASSIC($\lhd$), the conversion can now be performed.

5.4 A new user function

A new user function was implemented to enable the printing of an individual definition on file. The function called "cl-print-ind-to-file" was added to CLASSIC($\lhd$)'s "user.lisp" file.

The function takes an individual and a filename as input. The CLASSIC($\triangleleft$) definition of the individual is then written to the file with the given name. This function was requested for purposes of further extensions of this part of the larger information retrieval project mentioned in the introduction chapter.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The main goal of this thesis was to find a way to model enough functionality of SGML to be able to model the HTML definition as well as HTML document instances. A part of the goal was also to implement conversion functions from document language definitions and document instances to their Description Logics representations and back again.

The modeling goal has not been completely satisfied as not all functionality needed for HTML has been modeled. The main restriction was to not allow recursive element declarations. Allowing this would require the use of a more expressive Description Logics system. Actually the acyclicity assumption of T-Boxes is very common in the Description Logics community since cyclic concept definitions for example do not always have a clear meaning while some cyclic definitions do. The problems of allowing cyclic T-Boxes have been analysed in [18].

Apart from the restriction of acyclic T-Boxes the restrictions on the modeling are quite small and do generally not affect the modeling of HTML. Furthermore, the implementations of the conversion functions satisfy the initial goal of an implementation of the modeling.

The SGML functionality that is modeled also allows one to model languages that are different in nature from HTML. You could model much larger and more complex languages than HTML despite the fact that the content types that can be modeled were chosen to fit HTML.

The efficiency of the modeling of structured documents using Description Logics is promising. We have tested to have several large HTML documents represented in CLASSIC($\lhd$) simultaneously without noticing any slowdowns. The question as to how the system will behave when the use is scaled up is still open though.

The conversion of document definitions and document instances takes just seconds running uncompiled Lisp code on our system (a Sun SPARC station 10). The efficiency of loading the representations into CLASSIC($\lhd$) depends entirely on whether the CLASSIC($\lhd$) code is compiled or not. Loading a large document instance (consisting of about 180 individual definitions) takes about 20 seconds if the system runs using compiled code. If the code is not compiled it takes close to 20 minutes to load the document representation.

6.2 Future work

As mentioned in the introduction to this thesis, the work presented here is a part of a larger information retrieval project for the Internet. This thesis covers the modeling of HTML documents.

Work remains to be done concerning machine learning for the specific Description Logic language we use for representing HTML documents. Work on this has already begun and results are reported in [16].

We also need to implement functions for handling queries on the documents that are represented in CLASSIC($\lhd$). This includes functions for processing the Lisp strings that are fillers for the "textstring" role used to represent strings in documents.

Another thing that might be needed is some kind of more usable user interface between a user of this information retrieval system and CLASSIC($\lhd$).

Furthermore, a conversion function for converting general HTML document instances to document instances that only use the functionality that we model is needed.

Research on extending the parts of SGML functionality that can be modeled should be done. For example, investigations on how to allow T-Box cycles and what consequences that would bring should be performed.

One could also do more work on extending the content types that can be modeled. For example, we could model the content type (A | B | C | ...)? by introducing an **xor** construct and different semantics of the **atleastp** and **atmostp** constructs. We could extend the evaluation function as presented in Figure 6.1 (α and β are general concept terms, m is a natural number, n is a part name and $\uplus$ stands for the disjoint union of sets).

$val[(\mathbf{xor}\ \alpha\ \beta), \mathcal{I}] = val[\alpha,\mathcal{I}] \uplus val[\beta,\mathcal{I}]$
$val[(\mathbf{atleastp}\ m\ \alpha), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid \exists n : y \lhd_n x \wedge y \in val[\alpha,\mathcal{I}]\} \geq m\}$
$val[(\mathbf{atmostp}\ m\ \alpha), \mathcal{I}] = \{x \in \Delta \mid \sharp\{y \in \Delta \mid \exists n : y \lhd_n x \wedge y \in val[\alpha,\mathcal{I}]\} \leq m\}$

Figure 6.1: The semantics of some new DL constructs intended for handling extra content types.

(A | B | C | ...)?

(**and** (**atmostp** 1 (**xor** A B C ...))
  (**allp** A-p A) (**allp** B-p B)
  (**allp** C-p C) ...)

Figure 6.2: An example of how a content of the type: (A | B | C | ...)? could be modeled using some extra DL constructs.

Using these new constructs we could model the above mentioned content type as in Figure 6.2.

Appendix A

Modeling HTML

A.1 Introduction

This Appendix shows how we can model HTML with Description Logics. In two separate sections below we show first how the HTML DTD (somewhat restricted) can be modeled and then we present an example document and its Description Logics representation.

A.2 Modeling the HTML DTD

A.2.1 The HTML DTD

Below we present a slightly modified version of the HTML DTD presented in [3]. The main changes are that no "highlighting", "forms" or "deprecated" modes are supported and "recommended" mode is required. The reasons for these restrictions are that recursive element declarations are common in these modes. Furthermore, four element declarations have been changed, also due to problems with recursiveness.

A's content has changed from:
(string | A | IMG | BR)*
to
(string | IMG | BR)*.

DD's content has changed from
(string | A | IMG | BR | P | UL | OL | DIR | MENU | DL | PRE |
BLOCKQUOTE)*
to

(string | A | IMG | BR | P | PRE)*.

LI's content has changed from:
(string | A | IMG | BR | P | UL | OL | DIR | MENU | DL | PRE |
BLOCKQUOTE)*
to
(string | A | IMG | BR | P | DL | PRE)*.

BLOCKQUOTE's content has changed from:
(H1-H6 | string | A | IMG | BR | P | UL | OL | DIR | MENU | DL | PRE |
BLOCKQUOTE | HR | ADDRESS)*
to
(H1-H6 | P | UL | OL | DIR | MENU | PRE | HR | ADDRESS | IMG)*.

Continuously, we have stated that the content types that we model are enough for modeling all content types used in the HTML DTD. This is actually not quite true since we have one single element declaration that uses an unallowed content type. This is the element declaration of the element DL. It has the content: (DT | DD)+. Modeling this type would require the introduction of an **xor** construct as discussed before. We have handled this in a different way. The content has been changed to (DT | DD)*, which is reasonable since the content of both the DT and the DD element is restricted by the occurrence indicator *. Thus, the only difference with our version of the content is that in our version the DL element can have an empty content while the closest to an empty content with the original version would be a DT or a DD part with empty contents. Thus the difference is marginal.

We have also removed all included and excluded elements from the element declarations. The element A was excluded in the declaration of A. Since we have already removed this recursive property of A above (both directly under A in the part hierarchy but also from any level under A), we can say that we, in a way, model this element exclusion. Also the elements P, UL, OL, DIR, MENU, DL, PRE and BLOCKQUOTE are excluded from the DIR and MENU elements. However, since we have already restricted the element declaration of LI we almost model this exclusion restriction. Furthermore, the elements META and LINK are included in the declaration of the HEAD element and are then excluded again from the TITLE element (which is a part of HEAD). We did not consider these restrictions important enough for changing the concerning element declarations.

All declarations of shorttag minimization has been removed, and we require that all elements have both start tags and end tags in HTML document instances.

All attributes declared as #FIXED have been removed because they are very seldomly used in real HTML document instances, and most browsers (for instance Netscape) do not have the #FIXED requirement on them.

```
<!ENTITY % HTML.Version “-//IETF//DTD HTML 2.0//EN”>

<!-- ======= Character mnemonic entities =======-->

<!ENTITY amp “&#38;”>
<!ENTITY gt “&#62;”>
<!ENTITY lt “&#60;”>
<!ENTITY quot “&#34;”>


<!-- ======= Document Structure =======-->

<!ENTITY % html.content “HEAD,BODY”>

<!ELEMENT HTML (%html.content;)>

<!ATTLIST HTML VERSION string “//IETF//DTD HTML 2.0//EN”>

<!-- ========= Imported Names =========-->

<!ENTITY % Content-Type “string”>

<!ENTITY % HTTP-Method “GET | POST”>

<!ENTITY % URI “string”>


<!-- ========= DTD “Macros” =========-->

<!ENTITY % heading “H1 | H2 | H3 | H4 | H5 | H6”>

<!ENTITY % list “UL | OL | DIR | MENU”>
```

```
<!-- ======= Document Head =======-->

<!ENTITY % head.content “TITLE & ISINDEX? & BASE?”>

<!ELEMENT HEAD (%head.content;)>


<!ELEMENT NEXTID EMPTY>
<!ATTLIST NEXTID N string #REQUIRED>

<!-- <NEXTID> Next ID to use for link name -->
<!-- <NEXTID N=...> Next ID to use for link name -->


<!ELEMENT META EMPTY>

<!ATTLIST META HTTP-EQUIV string #IMPLIED>
<!ATTLIST META NAME string #IMPLIED>
<!ATTLIST META CONTENT string #REQUIRED>

<!-- <META> Generic Metainformation -->
<!-- <META HTTP-EQUIV=...> HTTP response header name -->
<!-- <META NAME=...> Metainformation name -->
<!-- <META CONTENT=“...”> Associated information -->


<!ELEMENT BASE EMPTY>
<!ATTLIST BASE HREF %URI; #REQUIRED>

<!-- <BASE> Base context document -->
<!-- <BASE HREF=“...”> Address for this document -->


<!ELEMENT ISINDEX EMPTY>

<!-- <ISINDEX> Document is a searchable index -->


<!ELEMENT LINK EMPTY>
```

```
<!ATTLIST LINK HREF %URI; #REQUIRED>
<!ATTLIST LINK REL string #IMPLIED>
<!ATTLIST LINK REV string #IMPLIED>
<!ATTLIST LINK URN string #IMPLIED>
<!ATTLIST LINK TITLE string #IMPLIED>
<!ATTLIST LINK METHODS string #IMPLIED>

<!-- <LINK> Link from this document -->
<!-- <LINK HREF="..."> Address of link destination -->
<!-- <LINK URN="..."> Lasting name of destination -->
<!-- <LINK REL=...> Relationship to destination -->
<!-- <LINK REV=...> Relationship of destination to this -->
<!-- <LINK TITLE="..."> Title of destination (advisory) -->
<!-- <LINK METHODS="..."> Operations allowed (advisory) -->


<!ELEMENT TITLE (string)>

<!-- <TITLE> Document title -->

<!-- <HEAD> Document head -->


<!-- ========== Document Body ==========-->

<!ENTITY % text "string | A | IMG | BR">

<!ENTITY % block.forms "BLOCKQUOTE">

<!ENTITY % preformatted "PRE">

<!ENTITY % block "P | %list; | %preformatted; | %block.forms;">

<!ENTITY % body.content
   "(%heading; | %block; | HR | ADDRESS | IMG)*">

<!ELEMENT BODY %body.content;>

<!-- <BODY> Document body -->
```

```
<!ELEMENT BLOCKQUOTE
   (%heading; | P | %list; | %preformatted; | HR | ADDRESS | IMG)*>

<!-- <BLOCKQUOTE> Quoted passage -->

<!-- ========== Lists =========-->
<!ELEMENT OL (LI+)>
<!ELEMENT UL (LI+)>


<!ATTLIST OL COMPACT (COMPACT) #IMPLIED>


<!ATTLIST UL COMPACT (COMPACT) #IMPLIED>

<!-- <UL> Unordered list -->
<!-- <UL COMPACT> Compact list style -->
<!-- <OL> Ordered, or numbered list -->
<!-- <OL COMPACT> Compact list style -->


<!ELEMENT DIR (LI+)>
<!ELEMENT MENU (LI+)>


<!ATTLIST DIR COMPACT (COMPACT) #IMPLIED>

<!ATTLIST MENU COMPACT (COMPACT) #IMPLIED>

<!-- <DIR> Directory list -->
<!-- <DIR COMPACT> Compact list style -->
<!-- <MENU> Menu list -->
<!-- <MENU COMPACT> Compact list style -->

<!ELEMENT LI (%text; | P | DL | PRE)*>

<!-- <LI> List item -->

<!ELEMENT DL (DT | DD)*>
```

```
<!ATTLIST DL COMPACT (COMPACT) #IMPLIED>

<!ELEMENT DT (%text;)*>

<!ELEMENT DD (%text;|P|PRE)*>

<!-- <DL> Definition list, or glossary -->
<!-- <DL COMPACT> Compact style list -->
<!-- <DT> Term in definition list -->
<!-- <DD> Definition of term -->


<!-- ========== Text Flows ==========-->

<!ELEMENT ADDRESS (%text;|P)*>

<!ENTITY % pre.content “string|A|HR|BR”>

<!ELEMENT PRE (%pre.content;)*>

<!ATTLIST PRE WIDTH string #IMPLIED>

<!-- <PRE> Preformatted text -->
<!-- <PRE WIDTH=...> Maximum characters per line -->


<!-- ========== Paragraphs=========-->

<!ELEMENT P (%text;)*>

<!-- <P> Paragraph -->

<!-- ========= Headings, Titles, Sections ========-->

<!ELEMENT H1 (%text;)*>
<!ELEMENT H2 (%text;)*>
<!ELEMENT H3 (%text;)*>
<!ELEMENT H4 (%text;)*>
<!ELEMENT H5 (%text;)*>
```

```
<!ELEMENT H6 (%text;)*>

<!-- <H1> Heading, level 1 -->
<!-- <H2> Heading, level 2 -->
<!-- <H3> Heading, level 3 -->
<!-- <H4> Heading, level 4 -->
<!-- <H5> Heading, level 5 -->
<!-- <H6> Heading, level 6 -->



<!ENTITY % A.content "(string | IMG | BR)*">

<!ELEMENT A %A.content;>
<!ATTLIST A HREF %URI; #IMPLIED>
<!ATTLIST A NAME string #IMPLIED>
<!ATTLIST A REL string #IMPLIED>
<!ATTLIST A REV string #IMPLIED>
<!ATTLIST A URN string #IMPLIED>
<!ATTLIST A TITLE string #IMPLIED>
<!ATTLIST A METHODS string #IMPLIED>

<!-- <A> Anchor; source/destination of link -->
<!-- <A NAME="..."> Name of this anchor -->
<!-- <A HREF="..."> Address of link destination -->
<!-- <A URN="..."> Permanent address of destination -->
<!-- <A REL=...> Relationship to destination -->
<!-- <A REV=...> Relationship of destination to this -->
<!-- <A TITLE="..."> Title of destination (advisory) -->
<!-- <A METHODS="..."> Operations on destination (advisory) -->


<!-- ========== Images ==========-->

<!ELEMENT IMG EMPTY>

<!ATTLIST IMG SRC %URI; #REQUIRED>
<!ATTLIST IMG ALT string #IMPLIED>
<!ATTLIST IMG ALIGN (top | middle | bottom) #IMPLIED>
<!ATTLIST IMG ISMAP (ISMAP) #IMPLIED>
```

```
<!-- <IMG> Image; icon, glyph or illustration -->
<!-- <IMG SRC="..."> Address of image object -->
<!-- <IMG ALT="..."> Textual alternative -->
<!-- <IMG ALIGN=...> Position relative to text -->
<!-- <IMG ISMAP> Each pixel can be a link -->

<!-- ========== Text Markup ==========-->

<!ELEMENT BR EMPTY>

<!-- <BR> Line break -->

<!ELEMENT HR EMPTY>

<!-- <HR> Horizontal rule -->

<!-- <HTML> HTML Document -->
```

A.2.2 The DL version of the HTML DTD

Here we present the file that resulted from executing our program "convert-dtd" on the HTML DTD presented above. It is a quite large file, but loading it into the compiled CLASSIC(◁) goes quite quickly (just a few seconds).

```
(defun close-concept-restricted-part-names-for-ind (part-list ind)
  (dolist (el part-list nil) (cl-ind-close-part-name ind el)))

(defun cl-no-fillers? (ind role-name)
  (let ((role (cl-named-role role-name)))
    (not (cl-fillers ind role))))

(defun generate-part-warning (ind)
  (format t "~&Error: The individual ~a has
    unallowed parts!" ind) 'CLASSIC-THING)

(defun generate-role-warning (ind)
```

```
    (format t "~&Error: The individual ~a has
       unallowed roles!" ind) 'CLASSIC-THING)

(cl-define-primitive-role 'ISMAP :attribute t)
(cl-define-primitive-role 'ALIGN :attribute t)
(cl-define-primitive-role 'ALT :attribute t)
(cl-define-primitive-role 'SRC :attribute t)
(cl-define-primitive-role 'WIDTH :attribute t)
(cl-define-primitive-role 'COMPACT :attribute t)
(cl-define-primitive-role 'METHODS :attribute t)
(cl-define-primitive-role 'TITLE :attribute t)
(cl-define-primitive-role 'URN :attribute t)
(cl-define-primitive-role 'REV :attribute t)
(cl-define-primitive-role 'REL :attribute t)
(cl-define-primitive-role 'HREF :attribute t)
(cl-define-primitive-role 'CONTENT :attribute t)
(cl-define-primitive-role 'NAME :attribute t)
(cl-define-primitive-role 'HTTP-EQUIV :attribute t)
(cl-define-primitive-role 'N :attribute t)
(cl-define-primitive-role 'VERSION :attribute t)
(cl-define-primitive-role 'textstring :attribute t)
(cl-define-primitive-role 'before :attribute t)
(cl-define-primitive-role '/name :attribute t)

(cl-define-primitive-part-name 'HEAD-p)
(cl-define-primitive-part-name 'BODY-p)
(cl-define-primitive-part-name 'TITLE-p)
(cl-define-primitive-part-name 'ISINDEX-p)
(cl-define-primitive-part-name 'BASE-p)
(cl-define-primitive-part-name 'BLOCKQUOTE-p)
(cl-define-primitive-part-name 'H1-p)
(cl-define-primitive-part-name 'H2-p)
(cl-define-primitive-part-name 'H3-p)
(cl-define-primitive-part-name 'H4-p)
(cl-define-primitive-part-name 'H5-p)
(cl-define-primitive-part-name 'H6-p)
(cl-define-primitive-part-name 'UL-p)
(cl-define-primitive-part-name 'OL-p)
(cl-define-primitive-part-name 'DIR-p)
(cl-define-primitive-part-name 'MENU-p)
```

```
(cl-define-primitive-part-name 'ADDRESS-p)
(cl-define-primitive-part-name 'LI-p)
(cl-define-primitive-part-name 'DL-p)
(cl-define-primitive-part-name 'DT-p)
(cl-define-primitive-part-name 'DD-p)
(cl-define-primitive-part-name 'PRE-p)
(cl-define-primitive-part-name 'P-p)
(cl-define-primitive-part-name 'HR-p)
(cl-define-primitive-part-name 'A-p)
(cl-define-primitive-part-name 'string-p)
(cl-define-primitive-part-name 'IMG-p)
(cl-define-primitive-part-name 'BR-p)

(cl-define-concept 'text '(and (atleast 1 textstring)))

(cl-create-ind 'quot '(and text (fills textstring "&#34;")
   (fills /name "HTML-entity")))

(cl-create-ind 'lt '(and text (fills textstring "&#60;")
   (fills /name "HTML-entity")))

(cl-create-ind 'gt '(and text (fills textstring "&#62;")
   (fills /name "HTML-entity")))

(cl-create-ind 'amp '(and text (fills textstring "&#38;")
   (fills /name "HTML-entity")))

(defun IMG-test1 (ind role)
   (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
         '(and (all ISMAP (one-of ISMAP))))))))

(defun IMG-test2 (ind role)
   (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
         '(and (all ALIGN (one-of top middle bottom))))))))

(defun IMG-test3 (ind role)
   (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
```

```
        '(and (all ALT string)))))))

(defun A-test4 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all METHODS string)))))))

(defun A-test5 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all TITLE string)))))))

(defun A-test6 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all URN string)))))))

(defun A-test7 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all REV string)))))))

(defun A-test8 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all REL string)))))))

(defun A-test9 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all NAME string)))))))

(defun A-test10 (ind role) (or
  (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all HREF string)))))))

(defun PRE-test11 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
```

```
        '(and (all WIDTH string)))))))

(defun DL-test12 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all COMPACT (one-of COMPACT))))))))

(defun MENU-test13 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all COMPACT (one-of COMPACT))))))))

(defun DIR-test14 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all COMPACT (one-of COMPACT))))))))

(defun UL-test15 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all COMPACT (one-of COMPACT))))))))

(defun OL-test16 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all COMPACT (one-of COMPACT))))))))

(defun LINK-test17 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all METHODS string)))))))

(defun LINK-test18 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
        '(and (all TITLE string)))))))

(defun LINK-test19 (ind role)
  (or (not (member role (cl-role-restrs ind)))
      (cl-instance? ind (cl-normalize-concept (cl-parse-expression
```

```
        '(and (all URN string)))))))

(defun LINK-test20 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
      '(and (all REV string)))))))

(defun LINK-test21 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
      '(and (all REL string)))))))

(defun META-test22 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
      '(and (all NAME string)))))))

(defun META-test23 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
      '(and (all HTTP-EQUIV string)))))))

(defun HTML-test24 (ind role)
  (or (not (member role (cl-role-restrs ind)))
    (cl-instance? ind (cl-normalize-concept (cl-parse-expression
      '(and (all VERSION string)))))))

(cl-define-concept 'HR '(and (fills /name "HR")))
(defun HR-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
  (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el)
      '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
      (return t))))))
(cl-add-description-rule 'HR-role-rule @c{HR} 'generate-role-warning
  :filter '(test-c HR-unallowed-roles?))

(cl-define-concept 'BR '(and (fills /name "BR")))
(defun BR-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
  (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el)
```

```
        '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
        (return t))))))
(cl-add-description-rule 'BR-role-rule @c{BR} 'generate-role-warning
  :filter '(test-c BR-unallowed-roles?))

(cl-define-concept 'IMG '(and (test-c IMG-test1 @r{ISMAP})
  (test-c IMG-test2 @r{ALIGN}) (test-c IMG-test3 @r{ALT})
  (all SRC string) (atleast 1 SRC) (fills /name "IMG")))
(cl-add-rule 'IMG-rule1 @c{IMG} '(and IMG (fills ISMAP ISMAP))
  :filter '(test-c cl-no-fillers? ISMAP))
(cl-add-rule 'IMG-rule2 @c{IMG} '(and IMG (fills ALIGN top))
  :filter '(test-c cl-no-fillers? ALIGN))
(cl-add-rule 'IMG-rule3 @c{IMG} '(and IMG (fills ALT "arne-anka"))
  :filter '(test-c cl-no-fillers? ALT))
(defun IMG-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
  (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el)
      '("@r{/NAME}" "@r{BEFORE}" "@r{ISMAP}" "@r{ALIGN}"
      "@r{ALT}" "@r{SRC}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'IMG-role-rule @c{IMG} 'generate-role-warning
  :filter '(test-c IMG-unallowed-roles?))

(cl-define-concept 'A '(and (allp string-p text) (allp IMG-p IMG)
  (allp BR-p BR) (test-c A-test4 @r{METHODS})
  (test-c A-test5 @r{TITLE}) (test-c A-test6 @r{URN})
  (test-c A-test7 @r{REV}) (test-c A-test8 @r{REL})
  (test-c A-test9 @r{NAME}) (test-c A-test10 @r{HREF})
  (fills /name "A")))
(cl-add-rule 'A-rule4 @c{A} '(and A (fills METHODS "arne-anka"))
  :filter '(test-c cl-no-fillers? METHODS))
(cl-add-rule 'A-rule5 @c{A} '(and A (fills TITLE "arne-anka"))
  :filter '(test-c cl-no-fillers? TITLE))
(cl-add-rule 'A-rule6 @c{A} '(and A (fills URN "arne-anka"))
  :filter '(test-c cl-no-fillers? URN))
(cl-add-rule 'A-rule7 @c{A} '(and A (fills REV "arne-anka"))
  :filter '(test-c cl-no-fillers? REV))
(cl-add-rule 'A-rule8 @c{A} '(and A (fills REL "arne-anka"))
  :filter '(test-c cl-no-fillers? REL))
(cl-add-rule 'A-rule9 @c{A} '(and A (fills NAME "arne-anka"))
  :filter '(test-c cl-no-fillers? NAME))
```

```
(cl-add-rule 'A-rule10 @c{A} '(and A (fills HREF "arne-anka"))
   :filter '(test-c cl-no-fillers? HREF))
(defun A-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@r{/NAME}" "@r{BEFORE}" "@r{METHODS}"
         "@r{TITLE}" "@r{URN}" "@r{REV}" "@r{REL}"
         "@r{NAME}" "@r{HREF}")
         :test #'equal) nil) (return t))))))
(cl-add-description-rule 'A-role-rule @c{A} 'generate-role-warning
   :filter '(test-c A-unallowed-roles?))

(defun A-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
   (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@pn{STRING-P}" "@pn{IMG-P}" "@pn{BR-P}")
         :test #'equal) nil) (return t))))))
(cl-add-description-rule 'A-part-rule @c{A} 'generate-part-warning
   :filter '(test-c A-unallowed-parts?))

(cl-define-concept 'H6 '(and (allp string-p text) (allp A-p A)
   (allp IMG-p IMG) (allp BR-p BR) (fills /name "H6")))
(defun H6-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
         (return t))))))
(cl-add-description-rule 'H6-role-rule @c{H6} 'generate-role-warning
   :filter '(test-c H6-unallowed-roles?))

(defun H6-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
   (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
         "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'H6-part-rule @c{H6} 'generate-part-warning
   :filter '(test-c H6-unallowed-parts?))

(cl-define-concept 'H5 '(and (allp string-p text) (allp A-p A)
   (allp IMG-p IMG) (allp BR-p BR) (fills /name "H5")))
```

```
(defun H5-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'H5-role-rule @c{H5} 'generate-role-warning
    :filter '(test-c H5-unallowed-roles?))

(defun H5-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'H5-part-rule @c{H5} 'generate-part-warning
    :filter '(test-c H5-unallowed-parts?))

(cl-define-concept 'H4 '(and (allp string-p text) (allp A-p A)
    (allp IMG-p IMG) (allp BR-p BR) (fills /name "H4")))
(defun H4-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'H4-role-rule @c{H4} 'generate-role-warning
    :filter '(test-c H4-unallowed-roles?))

(defun H4-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'H4-part-rule @c{H4} 'generate-part-warning
    :filter '(test-c H4-unallowed-parts?))

(cl-define-concept 'H3 '(and (allp string-p text) (allp A-p A)
    (allp IMG-p IMG) (allp BR-p BR) (fills /name "H3")))
(defun H3-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
```

```
        (return t))))))
(cl-add-description-rule 'H3-role-rule @c{H3} 'generate-role-warning
    :filter '(test-c H3-unallowed-roles?))

(defun H3-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'H3-part-rule @c{H3} 'generate-part-warning
    :filter '(test-c H3-unallowed-parts?))

(cl-define-concept 'H2 '(and (allp string-p text) (allp A-p A)
    (allp IMG-p IMG) (allp BR-p BR) (fills /name "H2")))
(defun H2-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'H2-role-rule @c{H2} 'generate-role-warning
    :filter '(test-c H2-unallowed-roles?))

(defun H2-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'H2-part-rule @c{H2} 'generate-part-warning
    :filter '(test-c H2-unallowed-parts?))

(cl-define-concept 'H1 '(and (allp string-p text) (allp A-p A)
    (allp IMG-p IMG) (allp BR-p BR) (fills /name "H1")))
(defun H1-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'H1-role-rule @c{H1} 'generate-role-warning
    :filter '(test-c H1-unallowed-roles?))
```

```
(defun H1-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
   (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
         "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'H1-part-rule @c{H1} 'generate-part-warning
   :filter '(test-c H1-unallowed-parts?))

(cl-define-concept 'P '(and (allp string-p text) (allp A-p A)
   (allp IMG-p IMG) (allp BR-p BR) (fills /name "P")))
(defun P-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
         (return t))))))
(cl-add-description-rule 'P-role-rule @c{P} 'generate-role-warning
   :filter '(test-c P-unallowed-roles?))

(defun P-unallowed-parts? (ind) (let ((part-list (cl-part-name-restrs ind)))
   (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
         "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'P-part-rule @c{P} 'generate-part-warning
   :filter '(test-c P-unallowed-parts?))

(cl-define-concept 'PRE '(and (allp string-p text) (allp A-p A)
   (allp HR-p HR) (allp BR-p BR) (test-c PRE-test11 @r{WIDTH})
   (fills /name "PRE")))
(cl-add-rule 'PRE-rule11 @c{PRE} '(and PRE (fills WIDTH "arne-anka"))
   :filter '(test-c cl-no-fillers? WIDTH))
(defun PRE-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el) '("@r{/NAME}"
         "@r{BEFORE}" "@r{WIDTH}") :test #'equal) nil)
         (return t))))))
(cl-add-description-rule 'PRE-role-rule @c{PRE} 'generate-role-warning
   :filter '(test-c PRE-unallowed-roles?))

(defun PRE-unallowed-parts? (ind)
```

```
(let ((part-list (cl-part-name-restrs ind)))
  (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
    (if (eq (member (format nil "~a" el)
      '("@pn{STRING-P}" "@pn{A-P}" "@pn{HR-P}"
      "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'PRE-part-rule @c{PRE} 'generate-part-warning
  :filter '(test-c PRE-unallowed-parts?))

(cl-define-concept 'ADDRESS '(and (allp string-p text) (allp A-p A)
  (allp IMG-p IMG) (allp BR-p BR) (allp P-p P)
  (fills /name "ADDRESS")))
(defun ADDRESS-unallowed-roles? (ind)
  (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
        '("@r{/NAME}" "@r{BEFORE}")
        :test #'equal) nil) (return t))))))
(cl-add-description-rule 'ADDRESS-role-rule @c{ADDRESS}
  'generate-role-warning
  :filter '(test-c ADDRESS-unallowed-roles?))

(defun ADDRESS-unallowed-parts? (ind)
  (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
        '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
        "@pn{BR-P}" "@pn{P-P}")
        :test #'equal) nil) (return t))))))
(cl-add-description-rule 'ADDRESS-part-rule @c{ADDRESS}
  'generate-part-warning
  :filter '(test-c ADDRESS-unallowed-parts?))

(cl-define-concept 'DD '(and (allp string-p text) (allp A-p A)
  (allp IMG-p IMG) (allp BR-p BR) (allp P-p P)
  (allp PRE-p PRE) (fills /name "DD")))
(defun DD-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
  (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el) '("@r{/NAME}"
      "@r{BEFORE}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'DD-role-rule @c{DD} 'generate-role-warning
```

```
   :filter '(test-c DD-unallowed-roles?))
(defun DD-unallowed-parts? (ind)
   (let ((part-list (cl-part-name-restrs ind)))
      (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
         (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}" "@pn{P-P}" "@pn{PRE-P}")
            :test #'equal) nil) (return t))))))
(cl-add-description-rule 'DD-part-rule @c{DD} 'generate-part-warning
   :filter '(test-c DD-unallowed-parts?))

(cl-define-concept 'DT '(and (allp string-p text) (allp A-p A)
   (allp IMG-p IMG) (allp BR-p BR) (fills /name "DT")))
(defun DT-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
         (return t))))))
(cl-add-description-rule 'DT-role-rule @c{DT} 'generate-role-warning
   :filter '(test-c DT-unallowed-roles?))

(defun DT-unallowed-parts? (ind)
   (let ((part-list (cl-part-name-restrs ind)))
      (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
         (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'DT-part-rule @c{DT} 'generate-part-warning
   :filter '(test-c DT-unallowed-parts?))

(cl-define-concept 'DL '(and (allp DT-p DT) (allp DD-p DD)
   (test-c DL-test12 @r{COMPACT}) (fills /name "DL")))
(cl-add-rule 'DL-rule12 @c{DL} '(and DL (fills COMPACT COMPACT))
   :filter '(test-c cl-no-fillers? COMPACT))
(defun DL-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@r{/NAME}" "@r{BEFORE}" "@r{COMPACT}")
         :test #'equal) nil) (return t))))))
(cl-add-description-rule 'DL-role-rule @c{DL} 'generate-role-warning
```

```
   :filter '(test-c DL-unallowed-roles?))

(defun DL-unallowed-parts? (ind)
   (let ((part-list (cl-part-name-restrs ind)))
      (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
         (if (eq (member (format nil "~a" el)
            '("@pn{DT-P}" "@pn{DD-P}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'DL-part-rule @c{DL} 'generate-part-warning
   :filter '(test-c DL-unallowed-parts?))

(cl-define-concept 'LI '(and (allp string-p text) (allp A-p A)
   (allp IMG-p IMG) (allp BR-p BR) (allp P-p P) (allp DL-p DL)
   (allp PRE-p PRE) (fills /name "LI")))
(defun LI-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
         '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
         (return t))))))
(cl-add-description-rule 'LI-role-rule @c{LI} 'generate-role-warning
   :filter '(test-c LI-unallowed-roles?))

(defun LI-unallowed-parts? (ind)
   (let ((part-list (cl-part-name-restrs ind)))
      (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
         (if (eq (member (format nil "~a" el)
            '("@pn{STRING-P}" "@pn{A-P}" "@pn{IMG-P}"
            "@pn{BR-P}" "@pn{P-P}" "@pn{DL-P}"
            "@pn{PRE-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'LI-part-rule @c{LI} 'generate-part-warning
   :filter '(test-c LI-unallowed-parts?))

(cl-define-concept 'MENU '(and (allp LI-p LI) (atleastp 1 LI-p)
   (test-c MENU-test13 @r{COMPACT}) (fills /name "MENU")))
(cl-add-rule 'MENU-rule13 @c{MENU} '(and MENU
   (fills COMPACT COMPACT))
   :filter '(test-c cl-no-fillers? COMPACT))
(defun MENU-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
   (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
      (if (eq (member (format nil "~a" el)
```

```
        '("@r{/NAME}" "@r{BEFORE}" "@r{COMPACT}")
        :test #'equal) nil) (return t))))))
(cl-add-description-rule 'MENU-role-rule @c{MENU}
  'generate-role-warning
  :filter '(test-c MENU-unallowed-roles?))

(defun MENU-unallowed-parts? (ind)
  (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
          '("@pn{LI-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'MENU-part-rule @c{MENU}
  'generate-part-warning
  :filter '(test-c MENU-unallowed-parts?))

(cl-define-concept 'DIR '(and (allp LI-p LI) (atleastp 1 LI-p)
  (test-c DIR-test14 @r{COMPACT}) (fills /name "DIR")))
(cl-add-rule 'DIR-rule14 @c{DIR} '(and DIR (fills COMPACT COMPACT))
  :filter '(test-c cl-no-fillers? COMPACT))
(defun DIR-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
  (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el)
        '("@r{/NAME}" "@r{BEFORE}" "@r{COMPACT}")
        :test #'equal) nil) (return t))))))
(cl-add-description-rule 'DIR-role-rule @c{DIR} 'generate-role-warning
  :filter '(test-c DIR-unallowed-roles?))

(defun DIR-unallowed-parts? (ind)
  (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
          '("@pn{LI-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'DIR-part-rule @c{DIR} 'generate-part-warning
  :filter '(test-c DIR-unallowed-parts?))

(cl-define-concept 'UL '(and (allp LI-p LI) (atleastp 1 LI-p)
  (test-c UL-test15 @r{COMPACT}) (fills /name "UL")))
(cl-add-rule 'UL-rule15 @c{UL} '(and UL (fills COMPACT COMPACT))
  :filter '(test-c cl-no-fillers? COMPACT))
(defun UL-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
```

```
(dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el)
        '("@r{/NAME}" "@r{BEFORE}" "@r{COMPACT}")
        :test #'equal) nil) (return t))))))
(cl-add-description-rule 'UL-role-rule @c{UL} 'generate-role-warning
  :filter '(test-c UL-unallowed-roles?))

(defun UL-unallowed-parts? (ind)
  (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el) '("@pn{LI-P}")
        :test #'equal) nil) (return t))))))
(cl-add-description-rule 'UL-part-rule @c{UL} 'generate-part-warning
  :filter '(test-c UL-unallowed-parts?))

(cl-define-concept 'OL '(and (allp LI-p LI) (atleastp 1 LI-p)
  (test-c OL-test16 @r{COMPACT}) (fills /name "OL")))
(cl-add-rule 'OL-rule16 @c{OL} '(and OL (fills COMPACT COMPACT))
  :filter '(test-c cl-no-fillers? COMPACT))
(defun OL-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
  (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
    (if (eq (member (format nil "~a" el)
      '("@r{/NAME}" "@r{BEFORE}" "@r{COMPACT}")
      :test #'equal) nil) (return t))))))
(cl-add-description-rule 'OL-role-rule @c{OL} 'generate-role-warning
  :filter '(test-c OL-unallowed-roles?))

(defun OL-unallowed-parts? (ind)
  (let ((part-list (cl-part-name-restrs ind)))
    (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
      (if (eq (member (format nil "~a" el)
        '("@pn{LI-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'OL-part-rule @c{OL} 'generate-part-warning
  :filter '(test-c OL-unallowed-parts?))

(cl-define-concept 'BLOCKQUOTE '(and (allp H1-p H1) (allp H2-p H2)
  (allp H3-p H3) (allp H4-p H4) (allp H5-p H5) (allp H6-p H6)
  (allp P-p P) (allp UL-p UL) (allp OL-p OL) (allp DIR-p DIR)
  (allp MENU-p MENU) (allp PRE-p PRE) (allp HR-p HR)
  (allp ADDRESS-p ADDRESS) (allp IMG-p IMG)
```

```
    (fills /name "BLOCKQUOTE")))
(defun BLOCKQUOTE-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@r{/NAME}" "@r{BEFORE}")
                :test #'equal) nil) (return t))))))
(cl-add-description-rule 'BLOCKQUOTE-role-rule @c{BLOCKQUOTE}
    'generate-role-warning :filter '(test-c BLOCKQUOTE-unallowed-roles?))

(defun BLOCKQUOTE-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@pn{H1-P}" "@pn{H2-P}" "@pn{H3-P}"
                "@pn{H4-P}" "@pn{H5-P}" "@pn{H6-P}"
                "@pn{P-P}" "@pn{UL-P}" "@pn{OL-P}"
                "@pn{DIR-P}" "@pn{MENU-P}" "@pn{PRE-P}"
                "@pn{HR-P}" "@pn{ADDRESS-P}" "@pn{IMG-P}")
                :test #'equal) nil) (return t))))))
(cl-add-description-rule 'BLOCKQUOTE-part-rule @c{BLOCKQUOTE}
    'generate-part-warning :filter '(test-c BLOCKQUOTE-unallowed-parts?))

(cl-define-concept 'BODY '(and (allp H1-p H1) (allp H2-p H2)
    (allp H3-p H3) (allp H4-p H4) (allp H5-p H5) (allp H6-p H6)
    (allp P-p P) (allp UL-p UL) (allp OL-p OL) (allp DIR-p DIR)
    (allp MENU-p MENU) (allp PRE-p PRE)
    (allp BLOCKQUOTE-p BLOCKQUOTE)
    (allp HR-p HR) (allp ADDRESS-p ADDRESS) (allp IMG-p IMG)
    (fills /name "BODY")))
(defun BODY-unallowed-roles? (ind)
    (let ((role-list (cl-role-restrs ind)))
        (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@r{/NAME}" "@r{BEFORE}")
                :test #'equal) nil) (return t))))))
(cl-add-description-rule 'BODY-role-rule @c{BODY} 'generate-role-warning
    :filter '(test-c BODY-unallowed-roles?))

(defun BODY-unallowed-parts? (ind)
```

```
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@pn{H1-P}" "@pn{H2-P}" "@pn{H3-P}"
                "@pn{H4-P}" "@pn{H5-P}" "@pn{H6-P}"
                "@pn{P-P}" "@pn{UL-P}" "@pn{OL-P}"
                "@pn{DIR-P}" "@pn{MENU-P}" "@pn{PRE-P}"
                "@pn{BLOCKQUOTE-P}" "@pn{HR-P}"
                "@pn{ADDRESS-P}" "@pn{IMG-P}")
                :test #'equal) nil) (return t))))))
(cl-add-description-rule 'BODY-part-rule @c{BODY}
    'generate-part-warning
    :filter '(test-c BODY-unallowed-parts?))

(cl-define-concept 'TITLE '(and (allp string-p text) (atleastp 1 string-p)
    (atmostp 1 string-p) (fills /name "TITLE")))
(defun TITLE-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'TITLE-role-rule @c{TITLE}
    'generate-role-warning
    :filter '(test-c TITLE-unallowed-roles?))

(defun TITLE-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@pn{STRING-P}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'TITLE-part-rule @c{TITLE}
    'generate-part-warning
    :filter '(test-c TITLE-unallowed-parts?))

(cl-define-concept 'LINK '(and (test-c LINK-test17 @r{METHODS})
    (test-c LINK-test18 @r{TITLE}) (test-c LINK-test19 @r{URN})
    (test-c LINK-test20 @r{REV}) (test-c LINK-test21 @r{REL})
    (all HREF string) (atleast 1 HREF) (fills /name "LINK")))
(cl-add-rule 'LINK-rule17 @c{LINK} '(and LINK
    (fills METHODS "arne-anka"))
```

```
    :filter '(test-c cl-no-fillers? METHODS))
(cl-add-rule 'LINK-rule18 @c{LINK} '(and LINK
    (fills TITLE "arne-anka"))
    :filter '(test-c cl-no-fillers? TITLE))
(cl-add-rule 'LINK-rule19 @c{LINK} '(and LINK (fills URN "arne-anka"))
    :filter '(test-c cl-no-fillers? URN))
(cl-add-rule 'LINK-rule20 @c{LINK} '(and LINK (fills REV "arne-anka"))
    :filter '(test-c cl-no-fillers? REV))
(cl-add-rule 'LINK-rule21 @c{LINK} '(and LINK (fills REL "arne-anka"))
    :filter '(test-c cl-no-fillers? REL))
(defun LINK-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}" "@r{METHODS}"
            "@r{TITLE}" "@r{URN}" "@r{REV}" "@r{REL}"
            "@r{HREF}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'LINK-role-rule @c{LINK} 'generate-role-warning
    :filter '(test-c LINK-unallowed-roles?))

(cl-define-concept 'ISINDEX '(and (fills /name "ISINDEX")))
(defun ISINDEX-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'ISINDEX-role-rule @c{ISINDEX}
    'generate-role-warning
    :filter '(test-c ISINDEX-unallowed-roles?))

(cl-define-concept 'BASE '(and (all HREF string) (atleast 1 HREF)
    (fills /name "BASE")))
(defun BASE-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}" "@r{HREF}")
            :test #'equal) nil) (return t))))))
(cl-add-description-rule 'BASE-role-rule @c{BASE} 'generate-role-warning
    :filter '(test-c BASE-unallowed-roles?))

(cl-define-concept 'META '(and (all CONTENT string)
```

```
    (atleast 1 CONTENT) (test-c META-test22 @r{NAME})
    (test-c META-test23 @r{HTTP-EQUIV})
    (fills /name "META")))
(cl-add-rule 'META-rule22 @c{META} '(and META
    (fills NAME "arne-anka"))
    :filter '(test-c cl-no-fillers? NAME))
(cl-add-rule 'META-rule23 @c{META} '(and META
    (fills HTTP-EQUIV "arne-anka"))
    :filter '(test-c cl-no-fillers? HTTP-EQUIV))
(defun META-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}" "@r{CONTENT}"
            "@r{NAME}" "@r{HTTP-EQUIV}")
            :test #'equal) nil) (return t))))))
(cl-add-description-rule 'META-role-rule @c{META}
    'generate-role-warning
    :filter '(test-c META-unallowed-roles?))

(cl-define-concept 'NEXTID '(and (all N string) (atleast 1 N)
    (fills /name "NEXTID")))
(defun NEXTID-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el) '("@r{/NAME}"
            "@r{BEFORE}" "@r{N}") :test #'equal) nil) (return t))))))
(cl-add-description-rule 'NEXTID-role-rule @c{NEXTID}
    'generate-role-warning
    :filter '(test-c NEXTID-unallowed-roles?))

(cl-define-concept 'HEAD '(and (allp TITLE-p TITLE)
    (atleastp 1 TITLE-p) (atmostp 1 TITLE-p)
    (allp ISINDEX-p ISINDEX) (atmostp 1 ISINDEX-p)
    (allp BASE-p BASE) (atmostp 1 BASE-p) (fills /name "HEAD")))
(defun HEAD-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}") :test #'equal) nil)
            (return t))))))
(cl-add-description-rule 'HEAD-role-rule @c{HEAD}
    'generate-role-warning
```

```
    :filter '(test-c HEAD-unallowed-roles?))

(defun HEAD-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@pn{TITLE-P}" "@pn{ISINDEX-P}" "@pn{BASE-P}")
                :test #'equal) nil) (return t))))))
(cl-add-description-rule 'HEAD-part-rule @c{HEAD}
    'generate-part-warning
    :filter '(test-c HEAD-unallowed-parts?))

(cl-define-concept 'HTML '(and (allp HEAD-p HEAD)
    (atleastp 1 HEAD-p) (atmostp 1 HEAD-p) (allp BODY-p BODY)
    (atleastp 1 BODY-p) (atmostp 1 BODY-p)
    (pp-constraint before HEAD-p BODY-p)
    (test-c HTML-test24 @r{VERSION}) (fills /name "HTML")))
(cl-add-rule 'HTML-rule24 @c{HTML} '(and HTML
    (fills VERSION "//IETF//DTD HTML 2.0//EN"))
    :filter '(test-c cl-no-fillers? VERSION))
(defun HTML-unallowed-roles? (ind) (let ((role-list (cl-role-restrs ind)))
    (dolist (el role-list nil) (if (not (equal (cl-at-most ind el) 0))
        (if (eq (member (format nil "~a" el)
            '("@r{/NAME}" "@r{BEFORE}" "@r{VERSION}")
            :test #'equal) nil) (return t))))))
(cl-add-description-rule 'HTML-role-rule @c{HTML}
    'generate-role-warning
    :filter '(test-c HTML-unallowed-roles?))

(defun HTML-unallowed-parts? (ind)
    (let ((part-list (cl-part-name-restrs ind)))
        (dolist (el part-list nil) (if (not (equal (cl-at-mostp ind el) 0))
            (if (eq (member (format nil "~a" el)
                '("@pn{HEAD-P}" "@pn{BODY-P}")
                :test #'equal) nil) (return t))))))
(cl-add-description-rule 'HTML-part-rule @c{HTML}
    'generate-part-warning
    :filter '(test-c HTML-unallowed-parts?))
```

A.3 Modeling HTML documents

A.3.1 An HTML document

In this section we show how a real HTML document can be modeled using our modeling system. In Figure A.1 we show what the uppermost part of the document looks like in Netscape. The use of the document in this thesis is with permission from the author.

In the listing below we present the HTML document instance. We have changed the original instance slightly by removing all “b” tags (boldface tags) and inserting proper end tags for all elements.

```
<html>
<head>
<title>USA Volleyball Home Page</title>
</head>
<body>
<p>
<a href=“index.html”>[Click here for advanced version of this page
(gifs, advanced HTML)]</a>
</p>
<h1>USA Volleyball Home Page</h1>
<p>
USA Volleyball is the national governing body for the sport,<br></br>
and the parent organization of the USA Men’s and Women’s National
Teams.
</p>
<p>
Page under development by <a href=“http://www.volleyball.org/taj/”>
Tom Jack</a>
on behalf of USAV, with the assistance of
<a href=“http://www.volleyball.org/usav/jkessel.html”>
John Kessel</a> of USA Volleyball.
</p>

<h1> USAV (formerly USVBA)</h1>
<p>
<a href=“http://www.volleyball.org/usav/”>USAV Home</a>
<a href=“http://www.volleyball.org/usav/#GenInfo”>Info</a>
<a href=“http://volleyball.org/usav/regions.html”>Regions</a>
```

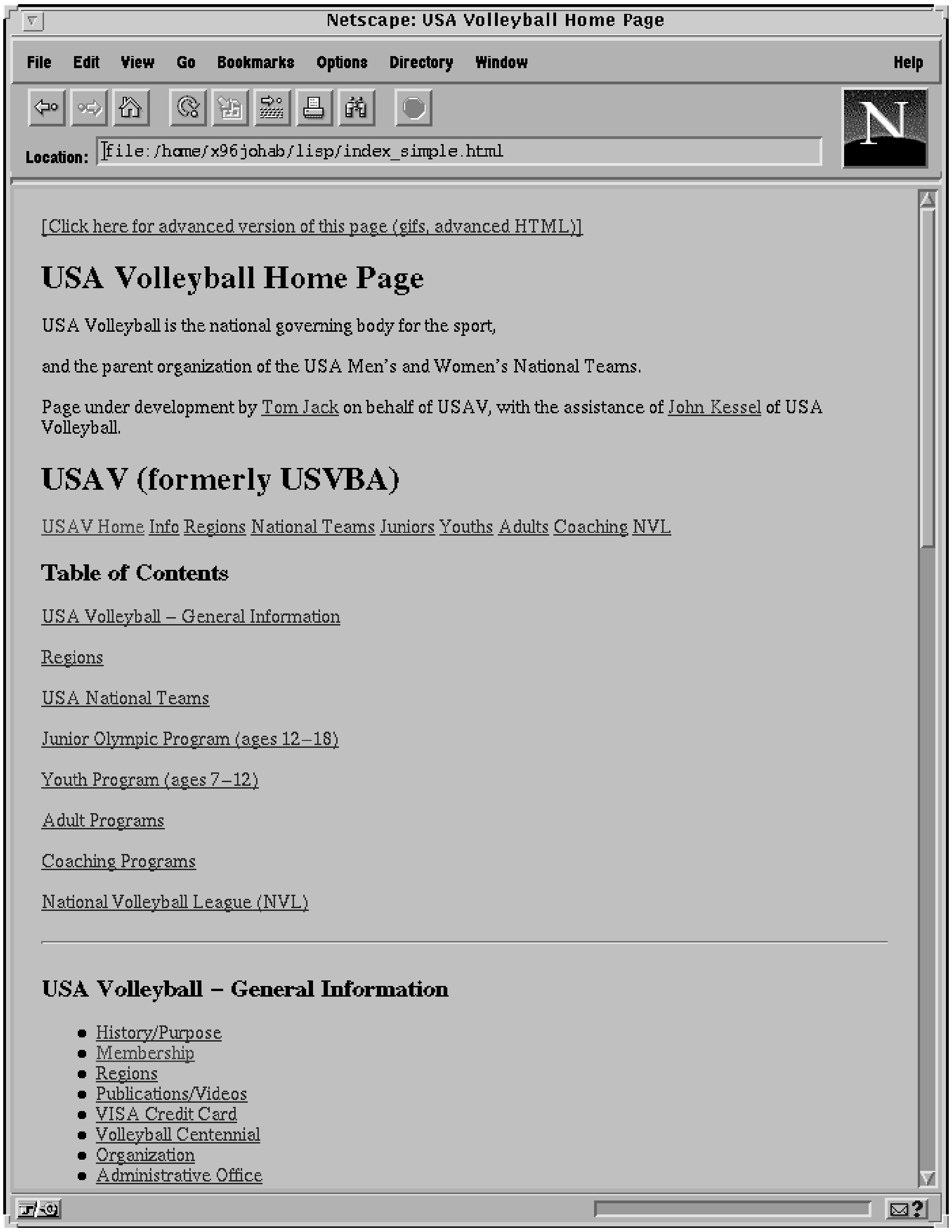

Figure A.1: What the uppermost part of the HTML document looks like in Netscape.

```
<a href=“http://www.volleyball.org/usav/#National”>
National Teams</a>
<a href=“http://www.volleyball.org/usav/#JO”>Juniors</a>
<a href=“http://www.volleyball.org/usav/#Youth”>Youths</a>
<a href=“http://www.volleyball.org/usav/#Adult”>Adults</a>
<a href=“http://www.volleyball.org/usav/#Coaching”>Coaching</a>
<a href=“http://www.volleyball.org/usav/#NVL”>NVL</a>
<br></br>
</p>

<h2>Table of Contents</h2>
<p>
<a href=“#GenInfo”>USA Volleyball - General Information</a>
<br></br>
<a href=“http://volleyball.org/usav/regions.html”>Regions</a>
<br></br>
<a href=“#National”>USA National Teams</a><br></br>
<a href=“#JO”>Junior Olympic Program (ages 12-18)</a><br></br>
<a href=“#Youth”>Youth Program (ages 7-12)</a><br></br>
<a href=“#Adult”>Adult Programs</a><br></br>
<a href=“#Coaching”>Coaching Programs</a><br></br>
<a href=“#NVL”>National Volleyball League (NVL)</a><br></br>
</p>
<hr></hr>

<h2><a name=“GenInfo”>USA Volleyball - General Information</a>
</h2>
<ul>
<li><a href=“history.html”>History/Purpose</a></li>
<li><a href=“membership.html”>Membership</a></li>
<li><a href=“regions.html”>Regions</a></li>
<li><a href=“publications.html”>Publications/Videos</a></li>
<li><a href=“visa.html”>VISA Credit Card</a></li>
<li><a href=“centennial.html”>Volleyball Centennial</a></li>
<li><a href=“organization.html”>Organization</a></li>
<li><a href=“organization.html#AdminOffice”>
Administrative Office</a></li>
<li><a href=“organization.html#NationalTeamOffice”>
National Teams Office</a></li>
<li><a href=“../fivb/index.html”>FIVB</a></li>
```

```
<li><a href="../olympics/index.html">1996 Olympics - Atlanta</a>
</li>
<li><a href="usofestival.html">1995 US Olympic Festival</a></li>
</ul>

<hr></hr>

<h2><a name="National"> USA National Teams</a></h2>
<h2>USA Men's National Team</h2>
<ul>
<li><a href="teamusa.html">USA Men Schedule</a></li>
<li><a href="us_men_roster.html">USA Men Roster</a></li>
</ul>

<h2>USA Women's National Team</h2>
<ul>
<li><a href="teamusa.html">USA Women Schedule</a></li>
<li><a href="us_women_roster.html">USA Women Roster</a></li>
<li><a href="national_teamw_history.html">
USA Women - History of Players on Team</a></li>
</ul>

<h2>USA Olympic Beach Volleyball</h2>
<ul>
<li><a href="../fivb/olympic_beach.html">
USA Olympic Beach Volleyball</a></li>
</ul>

<hr></hr>

<h2><a name="JO"> Junior Olympic Volleyball Program
(ages 12-18)</a></h2>
<ul>
<li><a href="juniors.html">Junior Programs in Your Area</a></li>
<li><a href="juniors96.html">Juniors -
1995-96 Inter-Regional Tournament Schedule</a></li>
<li><a href="juniors95.html">Juniors -
1994-95 Inter-Regional Tournament Schedule</a></li>
<li><a href="jos96.html">1996 Junior Olympics</a></li>
<li><a href="95jos_indoor.html">1995 Indoor Junior Olympics -
```

```
Results</a></li>
<li><a href=“95jos_beach.html”>1995 Beach Junior Olympics -
Results</a></li>
<li><a href=“summer_camps95.html”>
1995 Summer Camp Directory</a>
</li>
<li><a href=“jo_age_rule.html”>1995/96 Junior Age Definition</a>
</li>
</ul>

<hr></hr>

<h2><a name=“Youth”> Youth Volleyball Program (ages 7-12)</a>
</h2>
<ul>
<li><a href=“youth.html”>Youth Programs in Your Area</a>
</li> </ul>

<hr></hr>

<h2><a name=“Adult”> Adult Programs</a></h2>
<ul>
<li><a href=“adult.html”>Adult Programs in Your Area</a></li>
</ul>

<hr></hr>

<h2><a name=“Coaching”> Coaching Accreditation Program (CAP)
</a>
</h2>
<ul>
<li><a href=“cap.html”>What is CAP?</a></li>
<li><a href=“cap_level1.html”>CAP Level 1 Course</a></li>
<li><a href=“cap_1996.html”>1996 CAP Course Schedule</a></li>
<li><a href=“cap_1995.html”>1995 CAP Course Schedule</a></li>
<li><a href=“impact.html”>What is IMPACT?</a></li>
</ul>

<hr></hr>
```

```
<h2><a name="NVL">National Volleyball League (NVL)</a></h2>
<ul>
<li><a href="nvl.html">Introduction</a></li>
</ul>


<hr></hr>

<p>
<a href="http://www.volleyball.org/usav/">USAV Home</a>
<a href="http://www.volleyball.org/usav/#GenInfo">Info</a>
<a href="http://volleyball.org/usav/regions.html">Regions</a>
<a href="http://www.volleyball.org/usav/#National">
National Teams</a>
<a href="http://www.volleyball.org/usav/#JO">Juniors</a>
<a href="http://www.volleyball.org/usav/#Youth">Youths</a>
<a href="http://www.volleyball.org/usav/#Adult">Adults</a>
<a href="http://www.volleyball.org/usav/#Coaching">Coaching</a>
<a href="http://www.volleyball.org/usav/#NVL">NVL</a>
<br></br>
</p>

<p>
<a href="http://www.volleyball.org/index.html">VWW Home</a>
<a href="http://www.volleyball.org/#GenInfo">Info</a>
<a href="http://www.volleyball.org/usav/index.html">USAV</a>
<a href="http://www.volleyball.org/olympics/index.html">
Olympics</a>
<a href="http://www.volleyball.org/fivb/index.html">FIVB</a>
<a href="http://www.volleyball.org/#Pros">Pros</a>
<a href="http://www.volleyball.org/#College">College</a>
<a href="http://www.volleyball.org/www_sites/index.html">
Other Sites</a><br></br>
</p>

<hr></hr>
<p>
<a href="http://www.volleyball.org/taj/index.html">Webmaster -
Tom Jack</a><br></br>
Copyright &copy;<a href="http://www.volleyball.org">
```

```
Volleyball WorldWide.</a>
The information you receive on-line from Volleyball WorldWide
is protected by the copyright laws of the United States.
The copyright laws prohibit any copying, redistributing, retransmitting,
or repurposing of any copyright-protected material.<br></br>
Images Copyright &copy; USA Volleyball.<br></br>
Volleyball WorldWide on the Computer Internet/WWW<br></br>
http://www.volleyball.org/
</p>
</body>
</html>
```

A.3.2 The DL version of the HTML document

Below we show the CLASSIC(◁) code that resulted from our conversion program "convert-instance". It is a quite long listing, but the loading into CLASSIC(◁) takes only about 20 seconds if we use the compiled version.

```
(cl-create-ind 'string183 '(and (fills textstring
    "http://www.volleyball.org/")))

(cl-create-ind 'br182 '(and (fills before string183) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{br}) @i{br182})

(cl-create-ind 'string181 '(and (fills textstring "Volleyball WorldWide
    on the Computer Internet/WWW") (fills before br182)))

(cl-create-ind 'br180 '(and (fills before string181) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{br}) @i{br180})

(cl-create-ind 'string179 '(and (fills textstring
    "Images Copyright &copy; USA Volleyball.") (fills before br180)))

(cl-create-ind 'br178 '(and (fills before string179) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
```

```
    (cl-part-name-restrs @c{br}) @i{br178})

(cl-create-ind 'string177 '(and (fills textstring "The information you
    receive on-line from Volleyball WorldWideis protected by the copyright
    laws of the United States.The copyright laws prohibit any copying,
    redistributing, retransmitting, or repurposing of any
    copyright-protected material.") (fills before br178)))

(cl-create-ind 'texta176 '(and (fills textstring "Volleyball WorldWide.")))

(cl-create-ind 'a176 '(and (fills href "http://www.volleyball.org")
    (partfills string-p texta176) (fills before string177)
    (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a176})

(cl-create-ind 'string175 '(and (fills textstring "Copyright &copy;")
    (fills before a176)))

(cl-create-ind 'br174 '(and (fills before string175) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{br}) @i{br174})

(cl-create-ind 'texta173 '(and (fills textstring "Webmaster - Tom Jack")))

(cl-create-ind 'a173 '(and (fills href
    "http://www.volleyball.org/taj/index.html")
    (partfills string-p texta173) (fills before br174) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a173})

(cl-create-ind 'p172 '(and (partfills string-p string183)
    (partfills br-p br182) (partfills string-p string181)
    (partfills br-p br180) (partfills string-p string179)
    (partfills br-p br178) (partfills string-p string177)
    (partfills a-p a176) (partfills string-p string175)
    (partfills br-p br174) (partfills a-p a173) (fills /name "P")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p172})
```

```
(cl-create-ind 'hr171 '(and (fills before p172) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{hr}) @i{hr171})

(cl-create-ind 'br170 '(and (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br170})

(cl-create-ind 'texta169 '(and (fills textstring "Other Sites")))

(cl-create-ind 'a169 '(and (fills href
   "http://www.volleyball.org/www_sites/index.html")
   (partfills string-p texta169) (fills before br170) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a169})

(cl-create-ind 'texta168 '(and (fills textstring "College")))

(cl-create-ind 'a168 '(and (fills href
   "http://www.volleyball.org/#College") (partfills string-p texta168)
   (fills before a169) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a168})

(cl-create-ind 'texta167 '(and (fills textstring "Pros")))

(cl-create-ind 'a167 '(and (fills href
   "http://www.volleyball.org/#Pros") (partfills string-p texta167)
   (fills before a168) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a167})

(cl-create-ind 'texta166 '(and (fills textstring "FIVB")))

(cl-create-ind 'a166 '(and (fills href
   "http://www.volleyball.org/fivb/index.html")
   (partfills string-p texta166) (fills before a167) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a166})
```

```
(cl-create-ind 'texta165 '(and (fills textstring "Olympics")))

(cl-create-ind 'a165 '(and (fills href
    "http://www.volleyball.org/olympics/index.html")
    (partfills string-p texta165) (fills before a166) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a165})

(cl-create-ind 'texta164 '(and (fills textstring "USAV")))

(cl-create-ind 'a164 '(and (fills href
    "http://www.volleyball.org/usav/index.html")
    (partfills string-p texta164) (fills before a165) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a164})

(cl-create-ind 'texta163 '(and (fills textstring "Info")))

(cl-create-ind 'a163 '(and (fills href
    "http://www.volleyball.org/#GenInfo") (partfills string-p texta163)
    (fills before a164) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a163})

(cl-create-ind 'texta162 '(and (fills textstring "VWW Home")))

(cl-create-ind 'a162 '(and (fills href
    "http://www.volleyball.org/index.html") (partfills string-p texta162)
    (fills before a163) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a162})

(cl-create-ind 'p161 '(and (partfills br-p br170) (partfills a-p a169)
    (partfills a-p a168) (partfills a-p a167) (partfills a-p a166)
    (partfills a-p a165) (partfills a-p a164) (partfills a-p a163)
    (partfills a-p a162) (fills before hr171) (fills /name "P")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p161})

(cl-create-ind 'br160 '(and (fills /name "BR")))
```

```
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br160})

(cl-create-ind 'texta159 '(and (fills textstring "NVL")))

(cl-create-ind 'a159 '(and (fills href
   "http://www.volleyball.org/usav/#NVL") (partfills string-p texta159)
   (fills before br160) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a159})

(cl-create-ind 'texta158 '(and (fills textstring "Coaching")))

(cl-create-ind 'a158 '(and (fills href
   "http://www.volleyball.org/usav/#Coaching")
   (partfills string-p texta158) (fills before a159) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a158})

(cl-create-ind 'texta157 '(and (fills textstring "Adults")))

(cl-create-ind 'a157 '(and (fills href
   "http://www.volleyball.org/usav/#Adult")
   (partfills string-p texta157) (fills before a158) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a157})

(cl-create-ind 'texta156 '(and (fills textstring "Youths")))

(cl-create-ind 'a156 '(and (fills href
   "http://www.volleyball.org/usav/#Youth")
   (partfills string-p texta156) (fills before a157) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a156})

(cl-create-ind 'texta155 '(and (fills textstring "Juniors")))

(cl-create-ind 'a155 '(and (fills href
   "http://www.volleyball.org/usav/#JO")
   (partfills string-p texta155) (fills before a156) (fills /name "A")))
```

```
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a155})

(cl-create-ind 'texta154 '(and (fills textstring "National Teams")))

(cl-create-ind 'a154 '(and (fills href
    "http://www.volleyball.org/usav/#National")
    (partfills string-p texta154) (fills before a155) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a154})

(cl-create-ind 'texta153 '(and (fills textstring "Regions")))

(cl-create-ind 'a153 '(and (fills href
    "http://volleyball.org/usav/regions.html")
    (partfills string-p texta153) (fills before a154) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a153})

(cl-create-ind 'texta152 '(and (fills textstring "Info")))

(cl-create-ind 'a152 '(and (fills href
    "http://www.volleyball.org/usav/#GenInfo")
    (partfills string-p texta152) (fills before a153) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a152})

(cl-create-ind 'texta151 '(and (fills textstring "USAV Home")))

(cl-create-ind 'a151 '(and (fills href
    "http://www.volleyball.org/usav/") (partfills string-p texta151)
    (fills before a152) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a151})

(cl-create-ind 'p150 '(and (partfills br-p br160) (partfills a-p a159)
    (partfills a-p a158) (partfills a-p a157) (partfills a-p a156)
    (partfills a-p a155) (partfills a-p a154) (partfills a-p a153)
    (partfills a-p a152) (partfills a-p a151) (fills before p161)
    (fills /name "P")))
```

```
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{p}) @i{p150})

(cl-create-ind 'hr149 '(and (fills before p150) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{hr}) @i{hr149})

(cl-create-ind 'texta148 '(and (fills textstring "Introduction")))

(cl-create-ind 'a148 '(and (fills href "nvl.html")
   (partfills string-p texta148) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a148})

(cl-create-ind 'li147 '(and (partfills a-p a148) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li147})

(cl-create-ind 'ul146 '(and (partfills li-p li147)
   (fills before hr149) (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{ul}) @i{ul146})

(cl-create-ind 'texta145 '(and (fills textstring
   "National Volleyball League (NVL)")))

(cl-create-ind 'a145 '(and (fills name "NVL")
   (partfills string-p texta145) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a145})

(cl-create-ind 'h2144 '(and (partfills a-p a145)
   (fills before ul146) (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{h2}) @i{h2144})

(cl-create-ind 'hr143 '(and (fills before h2144)
   (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{hr}) @i{hr143})
```

```
(cl-create-ind 'texta142 '(and (fills textstring "What is IMPACT?")))

(cl-create-ind 'a142 '(and (fills href "impact.html")
   (partfills string-p texta142) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a142})

(cl-create-ind 'li141 '(and (partfills a-p a142)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li141})

(cl-create-ind 'texta140 '(and (fills textstring
   "1995 CAP Course Schedule")))

(cl-create-ind 'a140 '(and (fills href "cap_1995.html")
   (partfills string-p texta140) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a140})

(cl-create-ind 'li139 '(and (partfills a-p a140) (fills before li141)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li139})

(cl-create-ind 'texta138 '(and (fills textstring
   "1996 CAP Course Schedule")))

(cl-create-ind 'a138 '(and (fills href "cap_1996.html")
   (partfills string-p texta138) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a138})

(cl-create-ind 'li137 '(and (partfills a-p a138) (fills before li139)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li137})

(cl-create-ind 'texta136 '(and (fills textstring "CAP Level 1 Course")))
```

```
(cl-create-ind 'a136 '(and (fills href "cap_level1.html")
   (partfills string-p texta136) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a136})

(cl-create-ind 'li135 '(and (partfills a-p a136) (fills before li137)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li135})

(cl-create-ind 'texta134 '(and (fills textstring "What is CAP?")))

(cl-create-ind 'a134 '(and (fills href "cap.html")
   (partfills string-p texta134) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a134})

(cl-create-ind 'li133 '(and (partfills a-p a134) (fills before li135)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li133})

(cl-create-ind 'ul132 '(and (partfills li-p li141)
   (partfills li-p li139) (partfills li-p li137) (partfills li-p li135)
   (partfills li-p li133) (fills before hr143) (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{ul}) @i{ul132})

(cl-create-ind 'texta131 '(and (fills textstring
   "Coaching Accreditation Program (CAP)")))

(cl-create-ind 'a131 '(and (fills name "Coaching")
   (partfills string-p texta131) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a131})

(cl-create-ind 'h2130 '(and (partfills a-p a131) (fills before ul132)
   (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
```

```
    (cl-part-name-restrs @c{h2}) @i{h2130})

(cl-create-ind 'hr129 '(and (fills before h2130) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{hr}) @i{hr129})

(cl-create-ind 'texta128 '(and (fills textstring
    "Adult Programs in Your Area")))

(cl-create-ind 'a128 '(and (fills href "adult.html")
    (partfills string-p texta128) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a128})

(cl-create-ind 'li127 '(and (partfills a-p a128) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li127})

(cl-create-ind 'ul126 '(and (partfills li-p li127) (fills before hr129)
    (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{ul}) @i{ul126})

(cl-create-ind 'texta125 '(and (fills textstring "Adult Programs")))

(cl-create-ind 'a125 '(and (fills name "Adult")
    (partfills string-p texta125) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a125})

(cl-create-ind 'h2124 '(and (partfills a-p a125) (fills before ul126)
    (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h2}) @i{h2124})

(cl-create-ind 'hr123 '(and (fills before h2124) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{hr}) @i{hr123})

(cl-create-ind 'texta122 '(and (fills textstring
```

```
    "Youth Programs in Your Area")))

(cl-create-ind 'a122 '(and (fills href "youth.html")
    (partfills string-p texta122) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a122})

(cl-create-ind 'li121 '(and (partfills a-p a122) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li121})

(cl-create-ind 'ul120 '(and (partfills li-p li121) (fills before hr123)
    (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{ul}) @i{ul120})

(cl-create-ind 'texta119 '(and (fills textstring
    "Youth Volleyball Program (ages 7-12)")))

(cl-create-ind 'a119 '(and (fills name "Youth")
    (partfills string-p texta119) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a119})

(cl-create-ind 'h2118 '(and (partfills a-p a119) (fills before ul120)
    (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h2}) @i{h2118})

(cl-create-ind 'hr117 '(and (fills before h2118) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{hr}) @i{hr117})

(cl-create-ind 'texta116 '(and (fills textstring
    "1995/96 Junior Age Definition")))

(cl-create-ind 'a116 '(and (fills href "jo_age_rule.html")
    (partfills string-p texta116) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a116})
```

```
(cl-create-ind 'li115 '(and (partfills a-p a116) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li115})

(cl-create-ind 'texta114 '(and (fills textstring
   "1995 Summer Camp Directory")))

(cl-create-ind 'a114 '(and (fills href "summer_camps95.html")
   (partfills string-p texta114) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a114})

(cl-create-ind 'li113 '(and (partfills a-p a114) (fills before li115)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li113})

(cl-create-ind 'texta112 '(and (fills textstring
   "1995 Beach Junior Olympics - Results")))

(cl-create-ind 'a112 '(and (fills href "95jos_beach.html")
   (partfills string-p texta112) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a112})

(cl-create-ind 'li111 '(and (partfills a-p a112) (fills before li113)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li111})

(cl-create-ind 'texta110 '(and (fills textstring
   "1995 Indoor Junior Olympics - Results")))

(cl-create-ind 'a110 '(and (fills href "95jos_indoor.html")
   (partfills string-p texta110) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a110})

(cl-create-ind 'li109 '(and (partfills a-p a110) (fills before li111)
```

```
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li109})

(cl-create-ind 'texta108 '(and (fills textstring
    "1996 Junior Olympics")))

(cl-create-ind 'a108 '(and (fills href "jos96.html")
    (partfills string-p texta108) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a108})

(cl-create-ind 'li107 '(and (partfills a-p a108) (fills before li109)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li107})

(cl-create-ind 'texta106 '(and (fills textstring
    "Juniors - 1994-95 Inter-Regional Tournament Schedule")))

(cl-create-ind 'a106 '(and (fills href "juniors95.html")
    (partfills string-p texta106) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a106})

(cl-create-ind 'li105 '(and (partfills a-p a106) (fills before li107)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li105})

(cl-create-ind 'texta104 '(and (fills textstring
    "Juniors - 1995-96 Inter-Regional Tournament Schedule")))

(cl-create-ind 'a104 '(and (fills href "juniors96.html")
    (partfills string-p texta104) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a104})

(cl-create-ind 'li103 '(and (partfills a-p a104) (fills before li105)
    (fills /name "LI")))
```

```
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li103})

(cl-create-ind 'texta102 '(and (fills textstring
   "Junior Programs in Your Area")))

(cl-create-ind 'a102 '(and (fills href "juniors.html")
   (partfills string-p texta102) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a102})

(cl-create-ind 'li101 '(and (partfills a-p a102)
   (fills before li103) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li101})

(cl-create-ind 'ul100 '(and (partfills li-p li115) (partfills li-p li113)
   (partfills li-p li111) (partfills li-p li109) (partfills li-p li107)
   (partfills li-p li105) (partfills li-p li103) (partfills li-p li101)
   (fills before hr117) (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{ul}) @i{ul100})

(cl-create-ind 'texta99 '(and (fills textstring
   "Junior Olympic Volleyball Program (ages 12-18)")))

(cl-create-ind 'a99 '(and (fills name "JO") (partfills string-p texta99)
   (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a99})

(cl-create-ind 'h298 '(and (partfills a-p a99) (fills before ul100)
   (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{h2}) @i{h298})

(cl-create-ind 'hr97 '(and (fills before h298) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{hr}) @i{hr97})
```

```
(cl-create-ind 'texta96 '(and (fills textstring
    "USA Olympic Beach Volleyball")))

(cl-create-ind 'a96 '(and (fills href "../fivb/olympic_beach.html")
    (partfills string-p texta96) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a96})

(cl-create-ind 'li95 '(and (partfills a-p a96) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li95})

(cl-create-ind 'ul94 '(and (partfills li-p li95) (fills before hr97)
    (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{ul}) @i{ul94})

(cl-create-ind 'texth293 '(and (fills textstring
    "USA Olympic Beach Volleyball")))

(cl-create-ind 'h293 '(and (partfills string-p texth293)
    (fills before ul94) (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h2}) @i{h293})

(cl-create-ind 'texta92 '(and (fills textstring
    "USA Women - History of Players on Team")))

(cl-create-ind 'a92 '(and (fills href "national_teamw_history.html")
    (partfills string-p texta92) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a92})

(cl-create-ind 'li91 '(and (partfills a-p a92) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li91})

(cl-create-ind 'texta90 '(and (fills textstring "USA Women Roster")))

(cl-create-ind 'a90 '(and (fills href "us_women_roster.html")
```

```
    (partfills string-p texta90) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a90})

(cl-create-ind 'li89 '(and (partfills a-p a90) (fills before li91)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li89})

(cl-create-ind 'texta88 '(and (fills textstring "USA Women Schedule")))

(cl-create-ind 'a88 '(and (fills href "teamusa.html")
    (partfills string-p texta88) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a88})

(cl-create-ind 'li87 '(and (partfills a-p a88) (fills before li89)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li87})

(cl-create-ind 'ul86 '(and (partfills li-p li91) (partfills li-p li89)
    (partfills li-p li87) (fills before h293) (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{ul}) @i{ul86})

(cl-create-ind 'texth285 '(and (fills textstring
    "USA Women's National Team")))

(cl-create-ind 'h285 '(and (partfills string-p texth285)
    (fills before ul86) (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h2}) @i{h285})

(cl-create-ind 'texta84 '(and (fills textstring "USA Men Roster")))

(cl-create-ind 'a84 '(and (fills href "us_men_roster.html")
    (partfills string-p texta84) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a84})
```

```
(cl-create-ind 'li83 '(and (partfills a-p a84) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li83})

(cl-create-ind 'texta82 '(and (fills textstring "USA Men Schedule")))

(cl-create-ind 'a82 '(and (fills href "teamusa.html")
   (partfills string-p texta82) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a82})

(cl-create-ind 'li81 '(and (partfills a-p a82) (fills before li83)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li81})

(cl-create-ind 'ul80 '(and (partfills li-p li83) (partfills li-p li81)
   (fills before h285) (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{ul}) @i{ul80})

(cl-create-ind 'texth279 '(and (fills textstring
   "USA Men's National Team")))

(cl-create-ind 'h279 '(and (partfills string-p texth279)
   (fills before ul80) (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{h2}) @i{h279})

(cl-create-ind 'texta78 '(and (fills textstring
   "USA National Teams")))

(cl-create-ind 'a78 '(and (fills name "National")
   (partfills string-p texta78) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a78})

(cl-create-ind 'h277 '(and (partfills a-p a78) (fills before h279)
   (fills /name "H2")))
```

```
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{h2}) @i{h277})

(cl-create-ind 'hr76 '(and (fills before h277) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{hr}) @i{hr76})

(cl-create-ind 'texta75 '(and (fills textstring
   "1995 US Olympic Festival")))

(cl-create-ind 'a75 '(and (fills href "usofestival.html")
   (partfills string-p texta75) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a75})

(cl-create-ind 'li74 '(and (partfills a-p a75) (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li74})

(cl-create-ind 'texta73 '(and (fills textstring
   "1996 Olympics - Atlanta")))

(cl-create-ind 'a73 '(and (fills href "../olympics/index.html")
   (partfills string-p texta73) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a73})

(cl-create-ind 'li72 '(and (partfills a-p a73) (fills before li74)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li72})

(cl-create-ind 'texta71 '(and (fills textstring "FIVB")))

(cl-create-ind 'a71 '(and (fills href "../fivb/index.html")
   (partfills string-p texta71) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a71})

(cl-create-ind 'li70 '(and (partfills a-p a71) (fills before li72)
```

```
    (fills /name "LI"))) (close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li70})

(cl-create-ind 'texta69 '(and (fills textstring "National Teams Office")))

(cl-create-ind 'a69 '(and (fills href
    "organization.html#NationalTeamOffice") (partfills string-p texta69)
    (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a69})

(cl-create-ind 'li68 '(and (partfills a-p a69) (fills before li70)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li68})

(cl-create-ind 'texta67 '(and (fills textstring "Administrative Office")))

(cl-create-ind 'a67 '(and (fills href
    "organization.html#AdminOffice") (partfills string-p texta67)
    (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a67})

(cl-create-ind 'li66 '(and (partfills a-p a67) (fills before li68)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li66})

(cl-create-ind 'texta65 '(and (fills textstring "Organization")))

(cl-create-ind 'a65 '(and (fills href "organization.html")
    (partfills string-p texta65) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a65})

(cl-create-ind 'li64 '(and (partfills a-p a65) (fills before li66)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li64})
```

```
(cl-create-ind 'texta63 '(and (fills textstring "Volleyball Centennial")))

(cl-create-ind 'a63 '(and (fills href "centennial.html")
   (partfills string-p texta63) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a63})

(cl-create-ind 'li62 '(and (partfills a-p a63) (fills before li64)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li62})

(cl-create-ind 'texta61 '(and (fills textstring "VISA Credit Card")))

(cl-create-ind 'a61 '(and (fills href "visa.html")
   (partfills string-p texta61) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a61})

(cl-create-ind 'li60 '(and (partfills a-p a61) (fills before li62)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li60})

(cl-create-ind 'texta59 '(and (fills textstring "Publications/Videos")))

(cl-create-ind 'a59 '(and (fills href "publications.html")
   (partfills string-p texta59) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a59})

(cl-create-ind 'li58 '(and (partfills a-p a59) (fills before li60)
   (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{li}) @i{li58})

(cl-create-ind 'texta57 '(and (fills textstring "Regions")))

(cl-create-ind 'a57 '(and (fills href "regions.html")
```

```
    (partfills string-p texta57) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a57})

(cl-create-ind 'li56 '(and (partfills a-p a57) (fills before li58)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li56})

(cl-create-ind 'texta55 '(and (fills textstring "Membership")))

(cl-create-ind 'a55 '(and (fills href "membership.html")
    (partfills string-p texta55) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a55})

(cl-create-ind 'li54 '(and (partfills a-p a55) (fills before li56)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li54})

(cl-create-ind 'texta53 '(and (fills textstring "History/Purpose")))

(cl-create-ind 'a53 '(and (fills href "history.html")
    (partfills string-p texta53) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a53})

(cl-create-ind 'li52 '(and (partfills a-p a53) (fills before li54)
    (fills /name "LI")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{li}) @i{li52})

(cl-create-ind 'ul51 '(and (partfills li-p li74) (partfills li-p li72)
    (partfills li-p li70) (partfills li-p li68) (partfills li-p li66)
    (partfills li-p li64) (partfills li-p li62) (partfills li-p li60)
    (partfills li-p li58) (partfills li-p li56) (partfills li-p li54)
    (partfills li-p li52) (fills before hr76) (fills /name "UL")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{ul}) @i{ul51})
```

```
(cl-create-ind 'texta50 '(and (fills textstring
    "USA Volleyball - General Information")))

(cl-create-ind 'a50 '(and (fills name "GenInfo")
    (partfills string-p texta50) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a50})

(cl-create-ind 'h249 '(and (partfills a-p a50) (fills before ul51)
    (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h2}) @i{h249})

(cl-create-ind 'hr48 '(and (fills before h249) (fills /name "HR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{hr}) @i{hr48})

(cl-create-ind 'br47 '(and (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{br}) @i{br47})

(cl-create-ind 'texta46 '(and (fills textstring
    "National Volleyball League (NVL)")))

(cl-create-ind 'a46 '(and (fills href "#NVL")
    (partfills string-p texta46) (fills before br47) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a46})

(cl-create-ind 'br45 '(and (fills before a46) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{br}) @i{br45})

(cl-create-ind 'texta44 '(and (fills textstring "Coaching Programs")))

(cl-create-ind 'a44 '(and (fills href "#Coaching")
    (partfills string-p texta44) (fills before br45) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a44})
```

```
(cl-create-ind 'br43 '(and (fills before a44) (fills /name “BR”)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br43})

(cl-create-ind 'texta42 '(and (fills textstring “Adult Programs”)))

(cl-create-ind 'a42 '(and (fills href “#Adult”)
   (partfills string-p texta42) (fills before br43) (fills /name “A”)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a42})

(cl-create-ind 'br41 '(and (fills before a42) (fills /name “BR”)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br41})

(cl-create-ind 'texta40 '(and (fills textstring
   “Youth Program (ages 7-12)”)))

(cl-create-ind 'a40 '(and (fills href “#Youth”)
   (partfills string-p texta40) (fills before br41) (fills /name “A”)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a40})

(cl-create-ind 'br39 '(and (fills before a40) (fills /name “BR”)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br39})

(cl-create-ind 'texta38 '(and (fills textstring
   “Junior Olympic Program (ages 12-18)”)))

(cl-create-ind 'a38 '(and (fills href “#JO”)
   (partfills string-p texta38) (fills before br39) (fills /name “A“)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a38})

(cl-create-ind 'br37 '(and (fills before a38) (fills /name “BR”)))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br37})
```

(cl-create-ind 'texta36 '(and (fills textstring "USA National Teams")))

(cl-create-ind 'a36 '(and (fills href "#National")
  (partfills string-p texta36) (fills before br37) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
  (cl-part-name-restrs @c{a}) @i{a36})

(cl-create-ind 'br35 '(and (fills before a36) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
  (cl-part-name-restrs @c{br}) @i{br35})

(cl-create-ind 'texta34 '(and (fills textstring "Regions")))

(cl-create-ind 'a34 '(and (fills href
  "http://volleyball.org/usav/regions.html")
  (partfills string-p texta34) (fills before br35) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
  (cl-part-name-restrs @c{a}) @i{a34})

(cl-create-ind 'br33 '(and (fills before a34) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
  (cl-part-name-restrs @c{br}) @i{br33})

(cl-create-ind 'texta32 '(and (fills textstring
  "USA Volleyball - General Information")))

(cl-create-ind 'a32 '(and (fills href "#GenInfo")
  (partfills string-p texta32) (fills before br33) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
  (cl-part-name-restrs @c{a}) @i{a32})

(cl-create-ind 'p31 '(and (partfills br-p br47) (partfills a-p a46)
  (partfills br-p br45) (partfills a-p a44) (partfills br-p br43)
  (partfills a-p a42) (partfills br-p br41) (partfills a-p a40)
  (partfills br-p br39) (partfills a-p a38) (partfills br-p br37)
  (partfills a-p a36) (partfills br-p br35) (partfills a-p a34)
  (partfills br-p br33) (partfills a-p a32) (fills before hr48)
  (fills /name "P")))
(close-concept-restricted-part-names-for-ind
  (cl-part-name-restrs @c{p}) @i{p31})

```
(cl-create-ind 'texth230 '(and (fills textstring "Table of Contents")))

(cl-create-ind 'h230 '(and (partfills string-p texth230)
   (fills before p31) (fills /name "H2")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{h2}) @i{h230})

(cl-create-ind 'br29 '(and (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{br}) @i{br29})

(cl-create-ind 'texta28 '(and (fills textstring "NVL")))

(cl-create-ind 'a28 '(and (fills href
   "http://www.volleyball.org/usav/#NVL") (partfills string-p texta28)
   (fills before br29) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a28})

(cl-create-ind 'texta27 '(and (fills textstring "Coaching")))

(cl-create-ind 'a27 '(and (fills href
   "http://www.volleyball.org/usav/#Coaching")
   (partfills string-p texta27) (fills before a28) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @{a27})

(cl-create-ind 'texta26 '(and (fills textstring "Adults")))

(cl-create-ind 'a26 '(and (fills href
   "http://www.volleyball.org/usav/#Adult")
   (partfills string-p texta26) (fills before a27) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
   (cl-part-name-restrs @c{a}) @i{a26})

(cl-create-ind 'texta25 '(and (fills textstring "Youths")))

(cl-create-ind 'a25 '(and (fills href
   "http://www.volleyball.org/usav/#Youth")
```

```
    (partfills string-p texta25) (fills before a26) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a25})

(cl-create-ind 'texta24 '(and (fills textstring "Juniors")))

(cl-create-ind 'a24 '(and (fills href
    "http://www.volleyball.org/usav/#JO") (partfills string-p texta24)
    (fills before a25) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a24})

(cl-create-ind 'texta23 '(and (fills textstring "National Teams")))

(cl-create-ind 'a23 '(and (fills href
    "http://www.volleyball.org/usav/#National")
    (partfills string-p texta23) (fills before a24) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a23})

(cl-create-ind 'texta22 '(and (fills textstring "Regions")))

(cl-create-ind 'a22 '(and (fills href
    "http://volleyball.org/usav/regions.html")
    (partfills string-p texta22) (fills before a23) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a22})

(cl-create-ind 'texta21 '(and (fills textstring "Info")))

(cl-create-ind 'a21 '(and (fills href
    "http://www.volleyball.org/usav/#GenInfo")
    (partfills string-p texta21) (fills before a22) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a21})

(cl-create-ind 'texta20 '(and (fills textstring "USAV Home")))

(cl-create-ind 'a20 '(and (fills href
    "http://www.volleyball.org/usav/") (partfills string-p texta20)
```

```
    (fills before a21) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a20})

(cl-create-ind 'p19 '(and (partfills br-p br29) (partfills a-p a28)
    (partfills a-p a27) (partfills a-p a26) (partfills a-p a25)
    (partfills a-p a24) (partfills a-p a23) (partfills a-p a22)
    (partfills a-p a21) (partfills a-p a20) (fills before h230)
    (fills /name "P")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p19})

(cl-create-ind 'texth118 '(and (fills textstring
    "USAV (formerly USVBA)")))

(cl-create-ind 'h118 '(and (partfills string-p texth118)
    (fills before p19) (fills /name "H1")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h1}) @i{h118})

(cl-create-ind 'string17 '(and (fills textstring "of USA Volleyball.")))

(cl-create-ind 'texta16 '(and (fills textstring "John Kessel")))

(cl-create-ind 'a16 '(and (fills href
    "http://www.volleyball.org/usav/jkessel.html")
    (partfills string-p texta16) (fills before string17) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a16})

(cl-create-ind 'string15 '(and (fills textstring
    "on behalf of USAV, with the assistance of") (fills before a16)))

(cl-create-ind 'texta14 '(and (fills textstring "Tom Jack")))

(cl-create-ind 'a14 '(and (fills href
    "http://www.volleyball.org/taj/") (partfills string-p texta14)
    (fills before string15) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a14})
```

```
(cl-create-ind 'string13 '(and (fills textstring
    "Page under development by ") (fills before a14)))

(cl-create-ind 'p12 '(and (partfills string-p string17)
    (partfills a-p a16) (partfills string-p string15) (partfills a-p a14)
    (partfills string-p string13) (fills before h118) (fills /name "P")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p12})

(cl-create-ind 'string11 '(and (fills textstring
    "and the parent organization of the USA Men's and Women's
    National Teams.")))

(cl-create-ind 'br10 '(and (fills before string11) (fills /name "BR")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{br}) @i{br10})

(cl-create-ind 'string9 '(and (fills textstring
    "USA Volleyball is the national governing body for the sport,")
    (fills before br10)))

(cl-create-ind 'p8 '(and (partfills string-p string11)
    (partfills br-p br10) (partfills string-p string9) (fills before p12)
    (fills /name "P")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p8})

(cl-create-ind 'texth17 '(and (fills textstring
    "USA Volleyball Home Page")))

(cl-create-ind 'h17 '(and (partfills string-p texth17) (fills before p8)
    (fills /name "H1")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{h1}) @i{h17})

(cl-create-ind 'texta6 '(and (fills textstring
    "[Click here for advanced version of this page
    (gifs, advanced HTML)]")))
```

```
(cl-create-ind 'a6 '(and (fills href "index.html")
    (partfills string-p texta6) (fills /name "A")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{a}) @i{a6})

(cl-create-ind 'p5 '(and (partfills a-p a6) (fills before h17)
    (fills /name "P")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{p}) @i{p5})

(cl-create-ind 'body4 '(and (partfills p-p p172) (partfills hr-p hr171)
    (partfills p-p p161) (partfills p-p p150) (partfills hr-p hr149)
    (partfills ul-p ul146) (partfills h2-p h2144) (partfills hr-p hr143)
    (partfills ul-p ul132) (partfills h2-p h2130) (partfills hr-p hr129)
    (partfills ul-p ul126) (partfills h2-p h2124) (partfills hr-p hr123)
    (partfills ul-p ul120) (partfills h2-p h2118) (partfills hr-p hr117)
    (partfills ul-p ul100) (partfills h2-p h298) (partfills hr-p hr97)
    (partfills ul-p ul94) (partfills h2-p h293) (partfills ul-p ul86)
    (partfills h2-p h285) (partfills ul-p ul80) (partfills h2-p h279)
    (partfills h2-p h277) (partfills hr-p hr76) (partfills ul-p ul51)
    (partfills h2-p h249) (partfills hr-p hr48) (partfills p-p p31)
    (partfills h2-p h230) (partfills p-p p19) (partfills h1-p h118)
    (partfills p-p p12) (partfills p-p p8) (partfills h1-p h17)
    (partfills p-p p5) (fills /name "BODY")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{body}) @i{body4})

(cl-create-ind 'texttitle3 '(and (fills textstring
    "USA Volleyball Home Page")))

(cl-create-ind 'title3 '(and (partfills string-p texttitle3)
    (fills /name "TITLE")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{title}) @i{title3})

(cl-create-ind 'head2 '(and (partfills title-p title3)
    (fills before body4) (fills /name "HEAD")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{head}) @i{head2})
```

```
(cl-create-ind 'html1 '(and (partfills body-p body4)
    (partfills head-p head2) (fills /name "HTML")))
(close-concept-restricted-part-names-for-ind
    (cl-part-name-restrs @c{html}) @i{html1})
```

Bibliography

[1] Baader F. and Hollunder B., 'A Terminological Knowledge Representation System with Complete Inference Algorithms', *Proceedings of the International Workshop on Processing Declarative Knowledge*, pp. 67-86, 1991.

[2] Baader F., Hollunder B., Nebel B., Profitlich H.-J., Franconi E., 'An Empirical Analysis of Optimization Techniques for Terminological Representation Systems - or - Making KRIS get a move on', *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference - KR 92*, pp. 270-281, Cambridge, MA, USA, 1992.

[3] Berners-Lee T., Connolly D., 'Hypertext Markup Language - 2.0', HTML Working Group, *INTERNET-DRAFT*, September 22, 1995.

[4] Blake G., Consens P., Kilpeläinen P., Larson P., Snider T., and Tompa F., 'Text/Relational Database Management System: Harmonizing SQL and SGML', *Proceedings of the International Conference on Applications of Databases*, LNCS 819, pp. 267-280, Vadstena, Sweden, 1994.

[5] Boolos G. S., Jeffrey R. C., *Computability and Logic, Third Edition*, Cambridge University Press, 1989.

[6] Borgida A., Brachman R., McGuinness D., Resnick L., 'CLASSIC: A structural Data Model for Objects', *Proceedings of the ACM International Conference on Management of Data - SIGMOD 89*, pp. 59-67, Association for Computing Machinery.

[7] Borgida A., Patel-Schneider P., 'A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic', *Journal of Artificial Intelligence Research*, Vol 1, pp. 277-308, 1994.

[8] Brachman R., McGuinness D., Patel-Schneider P., Resnick L., Borgida A., 'Living with CLASSIC: When and How to Use a KL-ONE-Like Language', *Principles of Semantic Networks*, ed. Sowa, pp. 401-456, 1991.

[9] Brachman R., Schmolze J., 'An Overview of the KL-ONE Knowledge Representation System', *Cognitive Science*, pp. 171-216, August 1985.

[10] Calvanese D., Giacomo G., Lenzerini M., 'Representing SGML Documents in Description Logics', *Proceedings of the International Workshop on Description Logics*, 1996.

[11] Donini F.M., Lenzerini M., Nardi D., Nutt W., 'Tractable Concept Languages', *IJCAI-91*, pp. 458-465, 1991.

[12] Goldfarb C., *The SGML Handbook*, Oxford University Press, New York, USA, 1990.

[13] Heinsohn J., Kudenko D., Nebel B., Profitlich H.-J., 'An Empirical Analysis of Terminological Representation Systems', *Proceedings of the National Conference on Artificial Intelligence - AAAI 92*, pp. 676773, 1992.

[14] Hopcroft J.E., Ullman J.D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing Company, 1979.

[15] Lambrix P., *Part-Whole Reasoning in Description Logics*, Linköping Studies in Science and Technology, Dissertation No. 448, 1996.

[16] Larocchia P., *Learning Composite Objects in Description Logics*, M.Sc. Thesis, Department of Computer and Information Science, Linköping University, forthcoming, 1996.

[17] McGregor R., Brill D., 'Recognition Algorithms for the LOOM classifier', *AAAI-92*, pp. 774-779, 1992.

[18] Nebel B., *Reasoning and Revision in Hybrid Representation Systems*, LNAI 422, Springer-Verlag, Berlin, Heidelberg, New York, 1990.

[19] Resnick L., Borgida A., Brachman R., McGuinness D., Patel-Schneider P., Zalondek K., *CLASSIC Description and Reference Manual for the COMMON LISP Implementation Version 2.2*, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1993.

[20] Rich E., Knight K., *Artificial Intelligence*, McGraw-Hill, New York, USA, second edition, 1991.

[21] Sandewall E., *Tillämpad Logik*, Department of Computer and Information Science, Linköping University, Sweden, 1993.

[22] ISO 8879. *Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML)*, 1986.