# SemanticWeb im WS 2004/05

von: Jan Hinzmann
Vorlesung: Prof. Dr. N. Henze

Erstellt am:

12. März 2005

# Inhaltsverzeichnis

# 1 Semantic Web: Einführung und Übersicht

Dieser Abschnitt erklärt die Entstehung des Internets und des WWWs. Er gibt eine Einführung in die Thematik *Semantic Web* und stellt schliesslich Markup-sprachen vor.

## 1.1 Internet, World Wide Web und W3C

### 1.1.1 Internet

Zu Begin der 60'er Jahre entstand die Idee zu einem dezentralisierten Netzwerk, welches Daten redundant auf mehreren weit voneinander entfernten Rechnern halten sollte. Bei neuen oder geänderten Daten sollten sich alle angeschlossenen Rechner binnen kürzester Zeit den aktuellen Datenstand zusenden. Jeder Rechner sollte dabei über mehrere Wege mit jedem anderen Rechner kommunizieren können. Es war die Zeit des Kalten Krieges und neue Impulse für die EDV kamen hauptsächlich aus militärische Initiativen. So sollten wichtige Daten sicher sein und auch schlimmste Katastrophen (Atombombenangriff) überstehen. Dieses Projekt des "dezentralens Netzwerkes" des amerikanischen Verteidigungsministeriums scheiterte allerdings.

Die Advanced Research Projects Agency (ARPA), eine seit 1958 bestehende wissenschaftliche Einrichtung, deren Forschungsergebnisse in militärische Zwecke einflossen, entschloss sich 1966 zur Vernetzung der ARPA-eigenen Großrechner. Dabei wurde die Idee des "dezentralen Netzwerks" wieder aufgegriffen. Ende 1969 waren die ersten vier Rechner an das ARPA-Net angeschlossen. Drei Jahre später waren es bereits 40 Rechner. In dieser Zeit war es jedoch das ARPA-eigene Netz. In den ersten Jahren wurde das Netz deshalb auch ARPA-Net genannt. Aus ihm sollte später das Internet erwachsen.

Man erkannte nun schnell auch den akademischen Nutzen dieses ARPA-Netzes, welcher allerdings weniger in der Synchronisation von Daten, sondern eher im Datenaustausch lag. Wegen der offenen Architektur des ARPA-Net's stand einer solchen Verwendung nichts im Wege und Wissenschaftler können so seit den frühen 70er Jahren Forschungsergebnisse anderer Institute abrufen oder der wissenschaftlichen Gemeinschaft zur Verfügung stellen.

Die Anzahl der angeschlossenen Rechner wuchs nun ständig und stellte das Netz vor eine neue Herausforderung, unterschiedliche Rechnertypen mit verschiedenen nicht kompatiblen Betriebssystemen und unterschiedlichen Netzzugängen zu integrieren. Als Konsequenz wurde ein neues Datenübertragungsprotokoll, welches nicht an bestimmte Computersysteme, Übertragungswege oder -geschwindigkeiten gebunden ist, entwickelt. Das TCP/IP-Protokoll konnte dies leisten und stellt bis heute einen einheitlichen Standard dar.

Auch die Studenten entdeckten das Netz auf ihre Weise und es entstand das Usenet, die Hauptader der heutigen Newsgroups.

Anfang der 80er Jahre koppelte sich das MIL-Net vom ARPA-Net ab, da die Militärs ihre eigenen Interessen waren wollten. Im Laufe der 80er Jahre nahm die Anzahl der angeschlossenen Rechner nun sprunghaften zu. In Europa gab es zeitgleich ähnliche Entwicklungen, aber der Siegeszug von TCP/IP, welches nicht ISO-normiert war, war nicht mehr aufzuhalten und in Europa entstand schließlich ein europäisches Datennetz, das multiprotokollfähig war und unter

anderem TCP/IP unterstützte. Dieses Netz lief zunächst unter der Bezeichnung EuropaNET. Verschiedene nationale wissenschaftliche Netzwerke, etwa das Deutsche Forschungsnetz (DFN), wurden daran angeschlossen.

Das, was wir heute als *Internet* bezeichnen ist also ein Verbund von vielen kleinen, territorial oder organisatorisch begrenzten Netzen, welche eine Anbindung an die Backbones besitzen und so zu einem Gesamtnetzwerk zusammen geschlossen sind.

### 1.1.2 WWW

Im Internet gibt es eine vielzahl an Diensten (E-Mail, Telnet/SSH, Gopher, IRC, NEWS, WWW), wobei Gopher (kommt von "go for") als Vorgänger des *WWW* gilt. Mit ihm kann man große Informationsbestände leichter durchsuchbar machen. Allerdings gibt es keine Standards wie z.B. HTML. Im *WWW*, dem jüngsten Dienst im Internet, gibt diesen Standard.

Tim Berners-Lee beschloss '88 am Genfer Hochenergieforschungszentrum CERN sein Programm "Enquire" (de.: "sich erkundigen") systemunabhängig weiterzuentwickeln, wozu er 1990 die Genehmigung erhielt und bis zum Herbst eine erste Version des Projektes **World Wide Web** erstellte. Das *WWW* basiert auf den drei Säulen

1. HTTP (Hypertext Transfer Protocol) Die Spezifikation für die Kommunikation zwischen Web-Clients und Web-Servern
2. URI (Unified Resource Identifier) Die Spezifikation für die Addressierung beliebiger Dateien und Datenquellen im Web und übrigen Internet.
3. HTML (Hypertext Markup Language) Die Spezifikation einer Auszeichnungssprache für Web-Dokumente.

Nun ging es an die Entwicklung von Browsern, ohne welche die Web-Seiten nicht abgerufen werden konnten (1992, Erwise, Viola; Marc Andreesens Mosaic). Gleichzeitig stieg die Zahl der Web-Server ständig. 1995/96 kam der Boom von Netscape und 97 schlug Microsoft mit dem Internet Explorer als Teil seines Betriebssystems zurück. Ein Problem ist seither die Etablierung von Standards, es werden zwar immer neue Features in das *WWW* eingebaut, allerdings sind diese dann aus finanzpolitischen Gründen oft nicht standardisiert.

### 1.1.3 W3C

Das *W3C* ist ein unabhängiges Konsortium, welches technische Standards für das WWW entwickelt. Es ist trotz der Vormachtstellung von Microsoft auf dem Browsermarkt (80-90%) der mächtigste Faktor für die Weiterentwicklung des Webs geworden, weil es sich nicht gegen die Marktinteressen der Software-Firmen stellt, sondern seine Mitglieder gerade aus diesen rekrutiert.

Ende 1994 wurden die Grundlagen für das Konsortium geschaffen, da das CERN, dessen eigentliches Betätigungsfeld ja ein ganz anderes war, damit überfordert war. Die Mitglieder des W3C sind keine Einzelpersonen, sondern Firmen, die für 3-Jahresverträge Mitgliedsbeiträge bezahlen — wodurch sich das W3C finanziert — und im Gegenzug Zugang zu nichtöffentlichen Informationen erhalten und an der Weiterentwicklung von Standards (HTML, CSS, XML usw.) beteiligt werden.

Das W3C selbst unterteilt seine Arbeit in sogennannte Aktivitäten (*Activities*) und bildet für jede dieser Aktivitäten Arbeitsgruppen (*Working Groups*) und Interessengruppen (*Interest Groups*). Die Arbeitsgruppen beschäftigen sich mit der Ausarbeitung von technischer Referenzdokumentation und die Interessengruppen nehmen Einfluss auf diese. Das W3C veröffentlicht in einem definierten Prozess *Recommendations* zu den einzelnen Standards. Eine Recommendation durchläuft einen mehrfachen Review-Prozess und reift so von einem *Working Draft* durch Reviews der Arbeits- und Interessengruppe zu einer *Candidate Recommendation* und wird schliesslich nach Feedback der Öffentlichkeit und etwaigen Fehlerbereinigungen/Verbesserungen zu einer *Proposed Recommendation*, welche nach einem abschliessenden Review zur offiziellen, mit Versionsnummer versehenen *Recommendation* avanciert.

Die *IETF (Internet Engineering Task Force)* stellt eine andere Organsation zu Entwicklung von allgemeinen technischen Standards im Internet dar. Sie hat ein ähnliches, wenn auch nicht so straff organisertes Verfahren zur Veröffentlichung von sogenannten *RFC*s (Request for Comments). Die Sammlung von solchen RFCs reicht bis in die Anfänge des ARPA-Nets zurück. Es gibt beiepielsweise das RFC-1543, was den formalen Aufbau von RFC-Dokumenten beschreibt, was die Qualität solcher Dokumente sicherstellen soll.

## 1.2 Semantic Web – wofür?

Das heutige Web hat eine Reihe von Techniken hervorgebracht, die sich auf die Struktur, den Inhalt und die Nutzung des WWWs beziehen. So untersuchen Suchmaschinen die Strukturen und den Inhalt des Webs und Recommender den Inhalt und die Nutzung (Zugriffsstatistiken,...). Um nun das Web noch besser nutzen zu können liegen die Bemühungen zum einen in der Verbesserung der momentanen Techniken, also der Entwicklung von besseren Suchmaschinen, besseren KI-Werkzeugen usw. zum anderen wird versucht, den Maschinen ein Verständnis derAufgaben und des Inhalts zu geben. Man arbeitet also in Richtung von machinenprozessierbarem Inhalt, sodass Mensch und Maschine die Daten nutzen und bearbeiten können. Das ist das Ziel vom Semantic Web.

**Definition 1** *(Semantic Web)*
*The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation. — Tim Berners-Lee, James Hendler, Ora Lassila, May 17, 2001*

**Definition 2** *(Semantic Web)*
*The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a varity of applications using XML for syntax and URIs for naming. — Semantic Web Activity Group @ W3C*

http://www.w3.org/2001/sw/

### 1.2.1 Der "Semantic Web Tower" / "Semantic Web Protocol Stack"

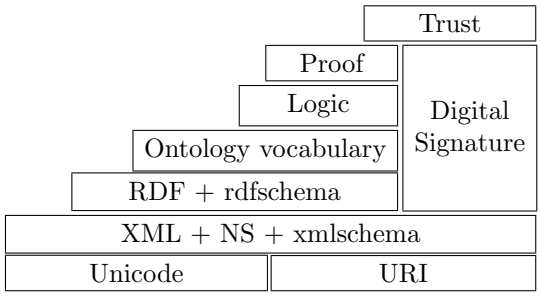



Abbildung 1: Semantic Web Tower

## 1.3 Markup Languages

### 1.3.1 SGML (Standard Generalized Markup Language)

Ist ISO-8879 Standard seit 1986. Über sogenannte DTDs ( Document Type Definitions) werden Markup-Sprachen definiert, also festgelegt, welche Elemente vorkommen dürfen, und welche Attribute diese haben und wie und ob diese Elemente verschachtelt werden können. HTML ist eine Markup-Sprache, die mit SGML definiert ist. Ebenso XML, wobei dieses eine besondere Stellung einnimmt, da es eine Teilmenge von SGML ist, keine fixe Tag-Menge hat, der Benutzer also neue Tags hinzufügen kann und die Erstellung von eigenen DTDs erlaubt, sodass die Benutzer-Tags dann definiert sind.

### 1.3.2 HTML (HyperText Markup Language)

Die Markup-Sprache des Web ist seit 1995 in ihrer Verson 2.0 offizieller Sprachstandard. Sie ist durch SGML definiert und besteht aus öffnenden Tags "`<tagname>`" und schliessenden Tags "`</tagname>`". Ein typisches HTML-Dokument hat einen umschliessenden Root-Knoten `<html>...</html>` und darin einen `<head>...</head>` und einen `<body>...</body>`. Im Kopf des Dokumentes können Metainformationen angegeben werden, dies sind dann von der Form `<meta name="..." content="...">`. Die Vorlesung stellt einige der HTML-Tags vor

- `<h[1-6]> Text </h[1-6]>` sind verschiedene Überschriften, von groß nach klein sortiert (1=GROSS, 2=Gross,...).
- `<br>` bricht die aktuelle Zeile um (kein End-Tag nötig in HTML).
- `<p>block of text </p>` definiert einen Absatz.
- `<hr>` veranlasst eine horizontale Linie über die gesamte Breite (kein End-Tag nötig in HTML).
- `<ul><li>einListenelement</li><li>noch eins</li></ul>` stellt eine unnummerierte Liste dar ((ul: unnumbered list) so wie diese hier).
- `<ol type="typ"><li>...</li></ol>` wird eine nummerierte Liste erzeugt, wobei `typ` die Werte a=alphbetisch, klein; A=alphabetisch, groß; i= römisch, klein und I=römisch, groß annehmen kann.
- `<b>text</b>` lässt den Text **dickgedruckt** erscheinen.

- `<i>text</i>` lässt den Text *kursiv* erscheinen.
- `<tt>text</tt>` lässt den Text in der `tele-typer`-Schriftart erscheinen.
- `<u>text</u>` unterstreicht den Text.
- `<strike>text</strike>` streicht den Text durch.
- ...

Dies sind alles Textstrukturierungs-Tags. Was HTML im Gegensatz zu normalem Text so nützlich macht, oder was aus normalem Text Hypertext macht, sind die Ankerpunkte und die sogenannten Links.

Mit den Ankerpunkten (kurz: Anker) kann man den Text navigierbar machen, dem Leser also ermöglichen, von einer Stelle im Text zu anderen Stellen zu springen. Ein Anker (anchor) wird durch `<a name="anchorName"> content </a>` erzeugt und stellt einen URI (Uniform Resource Identifier) dar.

Mit Hyperlinks (kurz: Link) kann man auf URIs verweisen und so dem Leser einen Sprung zu einer anderen Stelle im gleichen oder einem völlig anderen Text ermöglichen. Dieser klickt den Link einfach an und wird zu der referenzierten Stelle navigiert. So kann man zum Beispiel einen Link auf den oben vorgestellten Anker setzen, oder auf ein anderes lokales Dokument (./myfile.html) oder auf irgendwas, was durch einen URI beschrieben werden kann (WWW, gopher, FTP, Telnet, news,...). Links werden durch `<a href="URI" target="targetWindow">link text</a>` erzeugt, wobei `targetWindow` die Werte

**_blank** neues Fenster

**_self** gleiches Fenster

**_parent** gleiches Fenster, zerstöre momentanes Frameset

**_top** gleiches Fenster, zerstöre alle Framesets

annehmen kann. Auch besteht die Möglichkeit, sogenannte E-Mail-Links zu setzen, also Links, die durch klicken eine neue E-Mail-Nachricht erzeugen. Dies wird z.B. durch `<a href="mailto:adr@abc.de">linktext</a>` erreicht. Man kann nicht nur den Empfänger der Nachricht vorkonfigurieren, sondern mittels eines `?` hinter der Adresse weitere Felder deklarieren, z.B. `subject="..."` , `cc="..."`, `bcc="..."` oder `body="..."`, jeweils durch das Zeichen `&` getrennt.

# 2 XML: Einführung (DTDs und XML Schema)

## 2.1 XML (eXtensible Markup Language)

XML ist eine Teilmenge von SGML ohne feste Tag-Menge und erlaubt so seinen Benutzern das Erstellen von eigenen, xml-basierten, Auszeichnungssprachen durch Benutzung von eigenen Tags. Mit Hilfe von DTDs lassen sich diese definieren. Ein XML-Dokument besteht aus einem Prolog, einigen Elementen, die ihrerseits Attribute enthalten können und einem optionalen Epilog.

Im Prolog steht mindestens `<?xml version="1.0"?>` und optional werden hier auch Angaben über die Kodierung der Datei gemacht (z.B. `<?xml version="1.0" encoding="ISO-8859-1"?>`). Für das encoding-Attribut kann "UTF-8" (UTF: Unicode Transformation Format), "UTF-16" (ISO 10646 Standard), "UCS" oder "ISO-8859-1" gewählt werden. Ebenso kann mittels des standalone-attributs (`<?xml ... standalone="yes/no"?>`) angegeben werden, ob das Dokument alleinstehend ist oder seine Struktur in einer externen DTD-Datei festgelegt ist. Dies wäre dann als zweites im Prolog durch beispielsweise `<!DOCTYPE book SYSTEM "book.dtd">` zu definieren. Durch `SYSTEM` wird verdeutlicht, dass es sich bei der Strukturdatei um eine lokale Datei handelt. Ist dies nicht der Fall, kann eine DTD-Datei auch durch `PUBLIC` und einer URI angegeben werden.

Die XML-Elemente bestehen aus einem Anfangs- und einem End-Tag, in deren Mitte der Inhalt steht (`<foo>bar</foo>`). Die Tags können, bis auf die folgenden Beschränkungen, beliebig gewählt werden.

- der erste Buchstabe muss aus `letter|underscore|colon`, also z.B. `mytag|_mytag|:mytag` sein.
- Elemente dürfen nicht mit `xml|Xml|xML|...` beginnen.
- xml ist case-sensitive.

Der Inhalt eines Elements darf aus Text, weiteren Elementen oder nichts bestehen, wobei leere Elemente `<foo></foo>` durch `<foo/>` abgekürzt werden können.

Die XML-Attribute definieren die Eigenschaften der Elemente und sind Name-Wert-Paare innerhalb des Anfangs-Tags eines Elements. So zum Beispiel `<foo name="bar"/>`. Zu beachten ist hier, das Elemente verschachtelt werden können, Attribute hingegen nicht. Die Reihenfolge der Attribute ist im Gegensatz zu der der Elemente egal.

XML-Kommentare sind von der Form `<!-- Hier steht ein Kommentar -->` und ferner kann man noch Prozessierinformationen durch `<? ... ?>` angeben. Dies ist zum Beispiel nützlich, wenn man ein Stylesheet zur Verarbeitung referenzieren will: `<? stylesheet type="text/css" href="./style/my.css"?>`.

Man sagt von einem XML-Dokument, dass es *wellformed* ist, wenn es syntaktisch korrekt geschrieben ist und dass es *valide* ist, wenn es wellformed ist und die Strukturregeln befolgt, die in einer zugehörigen Schemadatei (DTD, XSD) definiert worden sind. Die syntaktischen Regeln sind die folgenden

- Es gibt nur ein Wurzelelement.
- Jedes Element besteht aus genau einem Anfangs- und einem End-Tag. Abkürzungen für leere Elemente sind möglich.
- Verschachtelte Tags dürfen sich nicht überlappen.

- Attribute eines Elementes müssen eindeutig sein.
- Die Elemente befolgen die oben genannten Namensrestriktionen.

## 2.2 XML – Document Type Definitions (DTD)

Die bereits erwähnten DTD-Dateien sind Dateien, in denen man die Struktur von XML-Dokumenten definieren kann. Man kann dann nicht nur prüfen, ob ein Dokument wellformed ist, sonderen darüber hinaus auch seine Validität.

DTD-Dateien können als eigene Datei erstellt werden oder in dem zugehörigen XML-Dokument enthalten sein. Oft hat man mehrere Dateien gleicher Struktur und deshalb sind externe DTD-Dateien zu bevorzugen, da dies dann doppelten Code vermeidet.

In der DTD-Sprache gibt es als einzigen atomaren Datentyp `#PCDATA` für Elemente; alle höheren Typen sind dann aus vorhandenen aufgebaut.
Ein Beispiel (XML-Code und zugehörige DTD-Informationen):

```
XML:
<author1>
    <name> J. Hinzmann </name>
    <courseOfStudies>Angewandte Informatik</courseOfStudies>
</author1>

DTD:
<!ELEMENT author1 (name, courseOfStudies)>
<!ELEMENT author2 (name | courseOfStudies)>
<!ELEMENT author3 ((name | courseOfStudies) & (name | courseOfStudies))>
<!ELEMENT name (#PCDATA)>
<!ELEMENT course_of_studies (#PCDATA)>
```

Die Bedeutung der DTD ist hier

- in der XML-Datei können die Elemente `author[1,2,3]`, `name` und `courseOfStudies` benutzt werden.
- Elemente vom Typ `author1` haben zwei Subelemente in der Reihenfolge `name` gefolgt von `courseOfStudies`.
- Elemente vom Typ `author2` haben entweder das Subelement `name` oder `courseOfStudies`.
- Elemente vom Typ `author3` enthalten `name` und `courseOfStudies` in beliebiger Reihenfolge.
- Die Elemente `name` und `courseOfStudies` können irgendeinen Inhalt haben (Strings).

In DTDs kann man zusätzlich die gewohnten Kardinalitäten `*` (keinmal oder beliebig oft), `+` (mindestens einmal) oder `?` (ein- oder keinmal) angeben, wobei der default-Wert keine Kardinalität vorsieht, was "genau einmal" heisst. Die Kardinalitätsoperatoren werden an den Namen angehängt `...<!ELEMENT author1 (name, courseOfStudies+)>...`.

Attrubite von XML-Elementen werden durch `<!ATTLIST tagname attrname TYP MUSS >` definiert, wobei

`tagname` der Name des Tags ist, für den Attribute definiert werden sollen,

`attrname` der Name des zu definierenden Attributes ist,

`TYP` den Typ des Attributs angibt; es kommen `CDATA` (String), `ID` (eindeutiger name), `IDREF` (Referenz auf einen ID) oder `IDREFs` (Reihe von Referenzen auf IDs) oder `(`$v_1$`|...|`$v_n$`)` als Aufzählung von möglichen Werten in Frage,

`MUSS` die Notwendigkeit des Attributs angibt; es kommen `#REQUIRED`(Attr. muss vorkommen), `#IMPLIED`(Attr. kann vorkommen), `#FIXED "value "`(Attr. muss mit genau den Wert `value` vorkommen) oder `"value "`(gibt einen default-Wert an, der im XML-Dokument überschrieben werden kann)

Ein Beispiel (XML-Code und zugehörige DTD-Informationen unter Nutzung von Attributen):

```
XML:
<?xml version="1.0"?>
<author surname="Hinzmann" givenname="Jan">
    <courseOfStudies>Angewandte Informatik</courseOfStudies>
</author>

DTD:
<!ELEMENT author (courseOfStudies)>
<!ATTLIST author
    surname      CDATA    #REQUIRED
    givenname    CDATA    #REQUIRED
    middlename   CDATA    #IMPLIED>

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

## 2.3 XML Schema

XML-Schema-Dateien (auch XSD: XML Schema Definition) definieren, wie auch die DTDs, die Struktur von XML-Dateien, mit dem Unterschied, dass sie in XML geschrieben sind. Das besondere ist hier, dass man neue Typen durch Erweiterung oder Beschränkung erstellen kann. Alle XSD-Dokumente sind Erweiterungen des XSD-Dokumentes von der W3C-Seite ("http://www.w3c.org/2000/10/XMLSchema"). In den Strukturdateien können wieder Elemente, Attribute, die Anzahl der Kindelemente, Datentypen, default-Werte und feste Werte definiert werden. Das folgende Beispiel zeigt eine XSD-Datei mit dem typischen Prolog, gefolgt von einer minimalen Elementdefinition, wobei `xsd:` den Namespace von `schema` darstellt. Man kann ihn innerhalb des Dokumentes allerdings auch weglassen, da er standardmäßig angenommen wird, sofern im Prolog angegeben (So wäre `<schema xmlns="http://www.w3c.org/2000/10/XMLSchema" version="1.0">` äquivalent zu dem im Beispiel angegebenem Prolog).
Die Kardinalitäten werden hier mit einer Ober- und einer Untergrenze als Ganze Zahl inklusive null angegeben. Die Kardinalität * würde man als `... minOccurs="0" maxOccurs="unbounded" ...` angeben.

```
<xsd:schema xmlns:xsd="http://www.w3c.org/2000/10/XMLSchema" version="1.0">
    <element name="myTag"/>
    <element name="myOtherTag" minOccurs="0" maxOccurs="1">
        <attribute name="name" use="optional" type="string"/>
        <attribute name="value" use="required" type="integer"/>
    </element>
```

```
    <element name="special" type="myType"/>
</xsd:schema>
```

Die zweite Elementdefinition beinhaltet noch die Definition von Attributen, wobei es folgende eingebaute Datentypen gibt

**Nummerische Typen** *integer, Short, Byte, Long, Float*

**String Typen** *string, ID, IDREF, CDATA, Language*

**Date and Time** *time, Date, Month, Year*

Der Nutzer kann dann noch eigene Typen als `simpleTyp` oder `complexTyp` definieren, wobei der simple Typ aus einem eingebautem Typ durch Einschränkung entsteht und der komplexe Typ durch Erweiterung oder Beschränkung von vorhandenen Typen.

Zur Konstruktion von komplexen Typen kann man zusätzlich folgende Sammlungs-Elemente als Hilfe heranziehen

**sequence** eine Sequenz aus existierenden Datentypen (geordned)

**all** eine Sammlung von Elementen, die vorkommen müssen (nicht geordnet)

**choice** eine Auswahl von Elementen, von denen eins vorkommen muss

So erweitert das folgende Beispiel den Typ `myType`

```
<complexType name="myType">
    <attribute name="name" type="string" use="optional"/>
    <sequence>
        <element name="foo" type="string" minOccurs="0" maxOccurs="unbounded"/>
        <element name="bar" type="string"/>
    </sequence>
</complexType>

<complexType name="myExtendedType">
    <extension base="myType">
        <attribute name="value" type="integer" use="required"/>
        <sequence>
            <element name="boo" type="string" minOccurs="0" maxOccurs="1"/>
        </sequence>
    </extension>
</complexType>
```

Eine Einschränkung gestaltet sich als

```
<complexType name="myExtendedType">
    <restriction base="myType">
        <attribute name="name" type="string" use="required"/>
        <sequence>
            <element name="foo" type="string" minOccurs="0" maxOccurs="1"/>
        </sequence>
    </restriction>
</complexType>
```

Simple Datentypen entstehen hingegen aus Beschränkung von eingebauten Datentypen. Dies wird am folgenden Beispiel deutlich, das den Typ `integer` als Basis nimmt und ihn auf den Zahlenraum von 1-31 einschränkt

```
...
<simpleType name="dayOfMonth">
    <restriction base="integer">
        <minInclusive value="1"/>
        <maxInclusive value="31"/>
    </restriction>
</simpleType>
...
```

Die Klasse `string` kann durch eine gezielte Aufzählung von gültigen Strings eingeschränkt werden

```
...
<simpleType name="dayOfWeek">
    <restriction base="string">
        <enumeration value="Montag"/>
        <enumeration value="Dienstag"/>
        ...
        <enumeration value="Sonntag"/>
    </restriction>
</simpleType>
...
```

Nicht nur Attribute können also Typen haben, auch Elemente. Dies erlaubt eine gute Modularisierung bzw. eine hohe Granularität.

```
...
<element name="email" type="emailType"/>

<complexType name="emailType">
    <sequence>
        <element name="head" type="headType"/>
        <element name="body" type="bodyType"/>
    </sequence>
</complexType>
...
```

## 2.4 Namespaces

Namensräume (Namespaces) lösen das Problem der, durch die verteilte und unabhängig voneinander stattfindende Entwicklung von XML-Strukturen und der dadurch bedingten Mehrzahl von DTDs oder XSDs entstehenden Nameskonflikte. Namensräume identifizieren dazu eine bestimmte DTD- oder XSD-Datei durch ein Präfix, was dann den benutzten Tags vorangestellt wird. Die Definition von mehreren Namensräumen stellt dann eine Verkürzung dar und sichert die Eindeutigkeit.

```
<prefix:example
    xmlns:prefix="http://www.example.org/namespaces/exampleDTD"
    xmlns:prefix2=http://www.anders.org/exampleDTD">
    <prefix:einTag>Hier wird der Tag aus dem Namensraum "prefix" benutzt</prefix:einTag>
    <prefix2:einTag>und hier der aus dem anderen Namensraum "prefix2"</prefix2:einTag>
...
```

Ebenso wie für die Elemente funktioniert das Präfigieren für die Attribute.

## 2.5 XHTML

XHTML bildet das bekannte HTML in XML ab, die Motivation liegt in dem Beschluss, keine weiteren Versionen von HTML zu erstellen, deshalb liegt XHTML als W3C-Recommendation vor. Derzeit wird von den Browsern auch nicht valides HTML interpretiert bzw kompensiert. Hier einige weitere Unterschiede

- Mit XHTML wird Validität erreicht
- Die Portabilität wird erhöht (non-desktop devices)
- Die Erweiterung wird vereinfacht; in HTML muss zur Einführung von neuen Elementen die gesamte DTD angepasst werden, wobei in XHTML die wohlgeformten, konsistenten Elemente zu der bestehenden DTD hinzugefügt werden können.
- Die Dokumente müssen wohlgeformt sein (korrekte Schachtelung der Tags)
- XHTML ist case-sensitiv (Attribut- und Elementnamen werden nun kleingeschrieben)
- Alle Tags müssen geschlossen werden (`<br/>`)
- Attributwerte müssen in Anführungsstrichen stehen (`<td rowspan="3">`)
- Attribut-Name-Wert-Paare müssen ausgeschrieben werden (`<ul compact="compact">`)
- Der Mime-Typ von HTML war `text/html`, von XHTML ist er `text/html, text/xml, application/xml`
- In XHTML-Dokumenten muss der Prolog vorkommen (`<?xml version="1.0"?>`)
- Für XHTML-Dokumente gibt es eine andere DTD
- Das Root-Element eines XHTML-Dokumentes hat ein Attribut: `<html xmlns="http://www.w3.org/1999/xhtml">`
- Bei der Deklaration von Ankerpunkten (`<a name="meinAnker"/><a href="#meinAnker">ZumAnker</a>`) ist das Attribute `name` veraltet. Man benutzt nun `id` (`<a id="meinAnker"/><a href="#meinAnker">ZumAnker</a>`).

Da XHTML in XML geschrieben ist, sollte es die Endung *.xml haben, allerdings dient die Dateiendung auch den Browsern zur Wahl des Parsers. So bewirkt die Endung *.html den Einsatz des HTML-Parsers, wohingegen die Endung *.xml den XML-Parser anwirft. Ausserdem stellen die meisten Browser XML-Dokumente als Baum dar, was wohl nicht immer wünschenswert ist.

# 3 Friends and Members of the XML Family

## 3.1 CSS

In Cascading Style Sheets (CSS) werden Stildefinitionen gehalten, die der Browser zu Darstellung eines Dokumentes heranzieht. Eine solche Stildefinition besteht aus einem Selektor, der das Element angibt, für welches die Regel gelten soll und einer Stilbeschreibung. Diese Stielbeschreibungen sind dann Name-Werte-Paare und es können mehrere für ein Element definiert werden. Die Syntax für eine Regel lautet also `selector {property1: value1; property2: value2; ...}`.

Die Stildefinitionen können entweder innerhalb eines `style`-Tags im Kopf der HTML-Datei, direkt im betreffenden Element durch das Attribut `style` definiert werden oder in einer externen Datei stehen und per `@import` oder `<link>` eingebunden werden. Das würde dann ungefähr so aussehen

```
<html>
    <head>
        <style>
            body {  font-family: "arial";  font-size: 24pt  }
        </style>
        <title> Eingebettetes CSS-Beispiel </title>
    </head>
    ...
```

oder direkt im betreffenden Tag

```
<html>
    <head>
        <title> Stil für einen Tag </title>
    </head>
    <body style="font-family: 'arial';  font-size: 24pt">
    ...
    </body>
</html>
```

Bei externen CSS-Dateien würde es wie folgt aussehen

```
in der CSS-Datei:
BODY
{
    font-family:  "ARIAL";
    font-size: 24pt;
}
...

in der HTML-Datei:
<html>
    <head>
        <style type="text/css">
            @import url(dateiname.css);
        </style>
        <title> Import der CSS-Datei für den body </title>
    </head>
    ...
ODER
```

```
<html>
    <head>
        <link rel=Stylesheet href=dateiname.css type="text/css">
        <title> Import der CSS-Datei für den body </title>
    </head>
```

In CSS gibt es unter anderem Pseudoklassen zur Manipulation des Verhaltens von Links (`:ACTIVE, :HOVER, :LINK, :VISITED`) und des `<cite>`-Tags (`:BEFORE, :AFTER`). Diese werden wie folgt benutzt

```
BODY {font-family: "Verdana"; font-size: 10pt }
A:LINK {color: blue; text-decoration: none }
A:VISITED {color: red; font-style: olique }
...
CITE:BEFORE {content: "Dieser Text kommt vor dem Inhalt von <cite>"}
CITE:AFTER {content:"... und dieser danach."}
```

## 3.2 XPATH

XPath ist ein W3C-Standard und benutzt Pfadangaben, um Knoten (Elemente) in einem XML-Dokument zu identifizieren /addressieren. Dies ist wesentlich für die später beschriebenen XSL-Transformationen. Stellt man sich folgendes Beispieldokument vor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
  <cd country="USA">
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <price>10.90</price>
  </cd>
  <cd country="UK">
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <price>9.90</price>
  </cd>
  <cd country="USA">
    <title>Greatest Hits</title>
    <artist>Dolly Parton</artist>
    <price>9.90</price>
  </cd>
</catalog>
```

so addressiert/bezeichnet

`/` den absoluten Pfad, startend vom Rootelement (z.B. `/catalog`)

`//` Elemente ab einer beliebigen Ebene

`tagname/...` den relativen Pfad ab dem per `tagname` angegebenem Element

`*` kann als wildcard benutzt werden (für unbekannte Elemente)

`/catalog/cd`, `/*/cd` oder z.B. `//cd` selektiert also das `<cd>`-Element

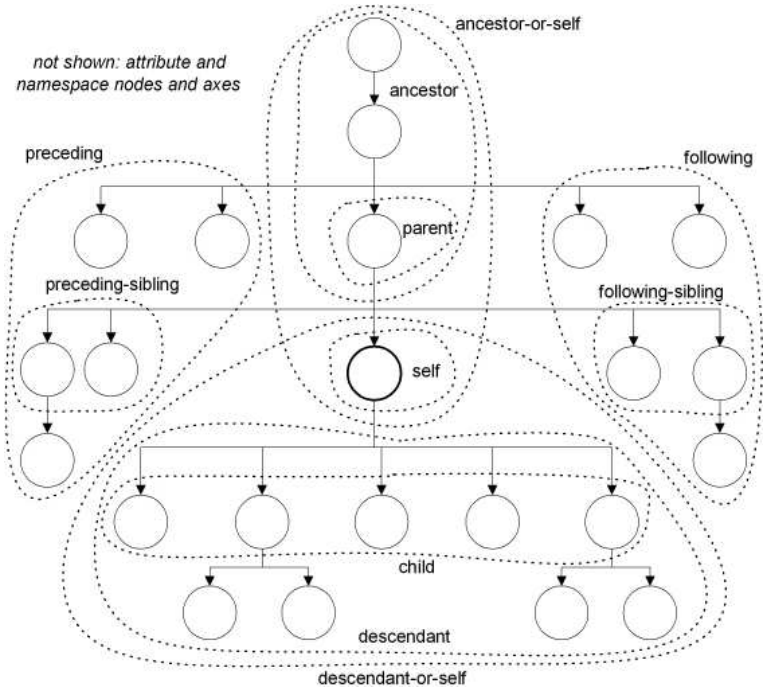
Durch eckige Klammern kann die Selektion noch eingeschränkt werden. So selektiert `/catalog/cd[1]` das erste cd-Element, `/catalog/cd[last()]` das letzte

(first() gibt es nicht.) oder `/catalog/cd[price]` alle `cd`-Elemente des `catalog`-Elements, die `price`-Elemente haben. Auch Bedingungen lassen sich so formulieren (`/catalog/cd[price=10.90]`). Der Oder-Operator `|` erlaubt die Auswahl von mehreren Pfaden (`/catalog/cd/title | /catalog/cd/artist`).

Auch die Attribute der Elemente können angesprochen werden, dies erfolgt mit Hilfe des Präfixes `@`. So liefert zum Beispiel `//cd[@country='UK']` alle `cd`-Elemente, deren Attribut `country` den Wert `UK` haben.

In einem XML-Dokument spricht man zusätzlich von Achsen und Knoten. Hierfür sind weitere Möglichkeiten der Auswahl vorgesehen, die im folgenden Beschrieben werden. Eine Achse definiert die Menge der Kindknoten des momentanen Knotens und ein Knotentest identifiziert einen Knoten aus dieser Menge. Diese Auswahl kann über den Namen oder den Typ erfolgen.

Das folgende Bild stammt von Crane Softwrights, gefunden auf http://www.nwalsh.com/docs/tutorials/xsl/xsl/slides.html




### 3.2.1 Achsentests

Folgende Achsen können spezifiziert werden

**ancestor** (Vorfahre) enthält alle Vorgänger des momentanen Knoten (Eltern, Großeltern,...)

**ancestor-or-self** wie `ancestor` nur mit dem momentanen Knoten inklusive

**attribute** alle Attribute des momentanen Knotens

**child** Alle Kindknoten des momentanen Knotens

**descendant** enthält alle Kind, Enkel, ...-knoten (Gegenteil von `ancestor`)
Note: enthält nie Attribute oder Namensraumknoten

**descendant-or-self** Der momentane Knoten plus alle Kinder, Enkel, ...

**following** enthält alles aus dem Dokument, ab dem Endtag des momentanen Knotens

**following-sibling** (Nachbar) enthält alle Nachbarn nach dem momentanen Knoten

**namespace** enthält alle Namensraumknoten des momentanen Knotens

**parent** enthält den umgebenden Knoten des momentanen Knotens

**preceding** enthält alle Knoten, die dem Anfangstag des momentanen Knotens vorangehen

**preceding-sibling** alle Nachbarn vor dem momentanen Knoten

**self** Der momentane Knoten selbst

### 3.2.2 Knotentests

Folgende Knotentests können angegeben werden

**name** Knoten, die zu dem angegebenem Namen passen

***** wird als Wildcard benutzt

**namespace:name** Knoten, die zu ddem angegebenem Namen im angegebenem Namensraum passen

**namespace:*** alle Knoten aus dem angegebenen Namensraum

**comment()** alle Kommentarknoten

**text()** alle Textknoten

**processing-instructions()** alle `<?...?>`-Knoten

Prädikate werden benutzt, um Teilmengen aus Knotenmengen zu filtern. So zum Beispiel `child::price[price=9.90]` ergibt alle Knoten, die Kinder des momentanen Knotens sind und ein `price`-Element mit dem Wert 9.90 haben.

### 3.2.3 Abkürzungen

Als wichtigsten Abkürzung ist wohl zu nennen, dass `child::` weggelassen werden kann. Weitere Abkürzungen sind

**none** kurz für: `child::` z.B. `child::cd = cd`

**@** kurz für: `attribute::` z.B. `child::cd[attribute::type="classic"] = cd[@type="classic"]`

**.** kurz für: `self::node()` z.B. `self:`

`:node()/descendant-or-self::node()/child::cd = .//cd`

**..** kurz für: `parent::node()` z.B. `parent::node()/child::cd = ../cd`

**//** kurz für: `/descendant-or-self::node()` z.B. `/descandant-or-self::node()/child::cd = //cd`

### 3.2.4 Weitere Ausdrücke

Es stehen die Operatoren `+, -, *, div, mod` für Arithmetik zur Verfügung und für Vergleiche `=` und `!=`. Auch `<, <=, >, >=` sind möglich. Es sollte beachtet werden, dass die Operanten vor der Auswertung in Zahlen convertiert werden. Boolesche Ausdrücke werden mit `and` und `or` angegeben.

Bei einem Test auf Gleichheit (Ungleichheit) ergibt sich `true`, wenn mindestens ein Knoten gleich (ungleich) ist. Dies hat zur Folge, dass beide Testarten auf der gleichen Menge `true` ergeben können, bzw. zwei Mengen gleich und ungleich zu gleich sein können.

Des weiteren steht eine Funktionenbibliothek zur Verfügung mit den Funktionen

**count(node-set)** liefert die Anzahl von Knoten in einer Knotenmenge

**id(value)** Selektiert Elemente durch ihre IDs

**last()** liefert die Positionsnummer des letzten Knotens aus der Liste der abgearbeiteten Knoten

**local-name(node)** liefert den lokalen Teil des Knotennamens, der üblicherweise aus `prefix:localname` besteht

**name(node)** liefert den Namen des übergebenen Knotens

**namespace-uri(node)** liefert den Namensraum, des Knotens im Argument

**position()** liefert die Position des momentanen Knotens in der Knotenliste

String-Funktionen sind

**concat(string, string, ...)** liefert die Konkatenation derArgumente

**contains(string, substr)** liefert `true`, wenn `substr` in `string` enthalten ist, `false` sonst

**normalize-space(string)** entfernt Leerzeichen vom Anfang und vom Ende das Argumentes

**starts-with(string, substr)** liefert `true`, wenn das `string`-Argument mit dem `substr`-Argument startet, `flase` sonst

**string()** konvertiert das Argument in einen String

**string-length(string)** liefert die Anzahl der Buchstaben des Strings

**substring(string, start, length)** liefert einen Teilstring ab der Stelle `start` mit der angegebenen Länge.

**substring-after(string, find)** liefert den Teil des Strings, der nach dem `find`-String steht

**substring-before(string, find)** liefert den Teil des Strings, der vor dem `find`-String steht

**translate(string, find, replace)** realisiert Ersetzungen (`translate('hallo', 'allo', 'i') -> hi`

Nummerische Funktionen

**ceiling()** liefert die kleinste ganze Zahl, die noch größer als das Argument ist

**floor()** liefert die größte ganze Zahl, die noch kleiner als das Argument ist

**number()** konvertiert das Argument in eine Zahl

**round()** undet das Argument auf die nächste ganze Zahl

**sum()** gibt das Ergebnis der Summe einer Menge von Zahlen zurück

Boolesche Funktiontn

**boolean()** convertiert das Argument in boolean und gibt `true` oder `false` zurück

**false()** gibt `false` zurück

**true()** gibt `true` zurück

**lang()** `true`, wenn die übergebene Sprache mit der aus `xsl:lang` übereinstimmt.

**not()** `true`, wenn das Argument `false` ist.

# 4 XSL: XSLT, XSL-FO

Mittels einer Stylesheet (XSL)-Datei lässt sich ein, aus einem XML-Dokument eingelesener XML-Baum in einen Ergebnisbaum (result-tree) überführen.

## 4.1 XSL:XSLT

Die XSL (Extensible Stylesheet Language) Sprache setzt sich aus drei W3C-Recommendations zusammen

**XPath** Die Sprache zur Identifikation von Elementen innerhalb eines XML-Dokumentes

**XSLT** Die Sprache zur Beschreibung, wie ein XML-Dokument in ein anderes transformiert werden soll

**XSL** Besteht aus XSLT und einer Menge von Formatierungsobjekten und Formatierungseigenschaften.

XSL-Dateien halten die Informationen, zur Darstellung von XML-Dateien. So wird es möglich, den Inhalt vom Stil zu trennen und zum Beispiel den gleichen Inhalt in verschiedenen Kontexten in dem passendem Stil anzubieten (www, wap, audio, html, pdf, ...). (Dies nennt man auch *"separation of concerns"*.)

XSL bietet folgende Transformationsfähigkeiten

- Generation von konstantem Text
- Unterdrückung von Inhalten
- Verschiebung von Text
- Vervielfältigung von Text
- Sortierung
- komplexere Transformationen, die neue Informationen aus den vorhandenen errechnen/generieren

Der Ergebnisbaum einer Transformation kann leicht als HTML oder XML serialisiert werden.

Im Wesentlichen besteht ein XSL-Stylesheet aus einer Reihe von Templates, die auf Elemente aus dem Ausgangsbaum "matchen" und dann Anweisungen parat halten, was für diese Elemente in den Ausgabebaum geschrieben werden soll. Dazu tragen Elemente aus dem XSL-Namensraum das Präfix `xsl:` und die anderen Elemente kommen in den Ergebnisbaum.
XSL-Stylesheets sind XML-Dokumente und haben als Rootknoten entweder das Tag `<xsl:stylesheet>` oder `<xsl:transform>`. Diese beiden Tags stammen aus dem gleichen Namensraum und werden synonym gebraucht. Es soll nun ein Beispiel folgen, dazu wird eine XML-Datei, ein Stylesheet und der generierte output angegeben

Listing 1: Die XML-Datei

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <?xml-stylesheet type="text/xsl" href="authors.xsl"?>
3 <authors>
4   <author>
```

```
    <name> Feuchtwanger </name>
    <book> Erfolg </book>
    <book> Die Geschwister Oppermann </book>
    <book> Exil </book>
  </author>
  <author>
    <name> Zweig </name>
    <book> Erziehung vor Verdun </book>
    <book> Der Streit um den Sergeanten Grischa </book>
  </author>
</authors>
```

Listing 2: Das Stylesheet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
 <xsl:template match="/authors">
        <html>
  <head><title>This is an example html</title>
  </head>
  <body>
      <xsl:for-each select="author">
   Author (<xsl:value-of select="position()"/>):
   <ul>
     <li><b><xsl:value-of select="name"/></b></li>
     <ul>
       <xsl:for-each select="book">
    <li><xsl:apply-templates/></li>
       </xsl:for-each>
     </ul>
   </ul>
      </xsl:for-each>
    </body>
      </html>
 </xsl:template>
</xsl:stylesheet>
```

Listing 3: Der Output

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>This is an example html</title>
</head>
<body>
   Author (1):
   <ul>
<li><b> Feuchtwanger </b></li>
<ul>
<li> Erfolg </li>
<li> Die Geschwister Oppermann </li>
<li> Exil </li>
</ul>
```

```
15 </ul>
16    Author (2):
17    <ul>
18 <li><b> Zweig </b></li>
19 <ul>
20 <li> Erziehung vor Verdun </li>
21 <li> Der Streit um den Sergeanten Grischa </li>
22 </ul>
23 </ul>
24 </body>
25 </html>
```

Hier ist schön zu sehen, wie in der XML-Datei nur der Inhalt steht und im Stylesheet die Stilinformationen, hier zur Darstellung einer HTML-Datei. In einem Browser sieht man dann die formatierte Ausgabe. Dieser generische Ansatz ermöglicht es auch, hinterher den Inhalt zu verändern, also Autoren oder Bücher hinzuzufügen, zu ändern oder zu löschen, ohne den Stil zu verändern.

Es ist prinzipiell möglich, ein Stylesheet aus nur einem Template aufzubauen, allerdings wird man in der Regel mehrere Templates benutzen, alleiner aus Gründen der Wartbarkeit. Hier kann es dann allerdings zu Problemen der Zuständigkeit kommen, welche durch das Regelwerk der Konfliktresolution gelöst werden sollen. Dieses Regelwerk sieht vor, dass zutreffende Templates aus importierten Modulen nicht berückdichtigt werden, wenn es ein Template im aktuellen Modul gibt. Ausserdem gibt es ein Prioritätensystem, welches Templates mit höherer Priorität den Vorrang gibt. Prioritäten werden wie folgt vergeben

- unqualifizierte Kinder oder Attribute haben eine Priorität von 0
- Prozessierinformationen haben eine Priorität von 0
- Durch Namensräume qualifizierte Wildcards (z.B. `html:*`) Knoten oder Attribute erhalten die Priorität -0.25
- unqualifizierte Wildcards (z.B. `*`) erhalten eine Priorität von -0.5
- alle anderen Templates erhalten eine default-priority von 0.5
- ferner ist es möglich, eine Priorität durch das `priority`-Attribut direkt für das Template zu setzen

Technisch gesehen ist es ein Fehler, wenn der Konfliktresolutionsprozess mehr als ein Template auswählt. Ein XSLT-Prozessor kann in der Bearbeitung fortfahren, durch Auswahl des letzten Templates im Stylesheet-Dokument. Als Konsequenz bedeutet das, dass die Reihenfolge im Stylesheet als letzte Entscheidungsinstanz fungiert.

Für die Transformation stehen nun einige Sprachelemtne zur Verfügung

- `<xsl:text>` Der Inhalt dieses Tags wrd in den Ergebnisbaum kopiert
- `<xsl:value-of>` der Inhalt des über das Attribut `select="...">` zuspezifizierenden Knotens aus dem Eingabebaum wird in den Ergebnisbaum kopiert
- `<xsl:attribute name="...">` fügt dem nächsten Knoten das Attribute mit dem Namen und Wert hinzu
- `<xsl:if>` ein einfaches `if`, ohne `else` (`<xsl:if test="$condition">`)

- `<xsl:choose>` realisiert ein "switch" mit `<xsl:when test=...>` Statements und abschliessendem `<xsl:otherwise>`
- `<xsl:message>` dient zur Meldung von Fehlern
- `<xsl:number>` realisiert Formatierungen von Überschrifften, Listen, etc.2
- Alle Elemente (in einem Template), die nicht aus dem Namensraum `<xsl:...>` stammen, werden in den Ergebnisbaum kopiert

## 4.2 XSL-FO

Formatting Objects (FO) sind Formatierungsobjekte, die in XSL durch die *CSS&FP (Cascading Style Sheet and Formatting Property) Working Group* definiert wurden. Sie stellen eine Art Strandardschnittstelle oder -strukturierung dar, die durch eine Transformation erreicht werden kann, wodurch ein ein XSL-Formatierer in die Lage versetzt wird, das gewünschte Ausgabeformat zu liefern. Die Objekte sind grundsätzlich in sogenannte inline- und block-Objekte zu unterscheiden und erinnern an die Gliederung von LaTeX-Dokumenten. So gibt es unteranderem

**page-sequence** stellt eine übergeordnete Struktureinheit dar

**flow** stellt ein Kapitel oder eine Sektion innerhalb einer *page-sequence* dar

**block** stellt eine *subsubsection* oder einen Absatz dar

**inline** Formatierungsobjekte, die keinen Absatz bewirken (z.B. eine Schriftartänderung in LaTeX: `foo \textit{bar} end.`)

**wrapper** realisiert ein transparentes Objekt, welches *inline* oder als Block benutzt werden kann und dessen einziger Zweck darin besteht, einen Platz zu schaffen, um vererbbare Eigenschaften unterzubringen

**lists** es gibt `list-block, list-item, list-item-label, list-item-body`

**graphic-references** realisieren Zeiger auf externe Grafikobjekte

**table FOs** sind im wesentlichen analog zu den Standard-Tabellen-Modelle (CALS, OASIS, HTML)

Zusätzlich lassen sich Schrift-, Rand-, Abstands-, Umbruchseigenschaften, horizontale Ausrichtung, Einrückung und viele weitere Formateigenschaften einstellen. Die Formatierungsmöglichkeiten von XSL FO 1.0 (W3C Recommendation) entsprechen ungefähr denen von HMTL + CSS.

Nicht möglich sind komplexe Layouts, wie Magazin- oder Zeitungslayout oder "looseleaf pagination" (Ringbuch/Lose-Blattsammlung)

Es gibt schon einige XSLT-Prozessoren (James Clarks XT, Saxon, Oracle XSLT, XALAN)

FOP ist der Welt erste Druckformatierer, der durch XSL Formatierungsobjekte gestreuert wird. Er liest, in Java$^{tm}$ geschrieben, einen aus Formatierungsobjekten bestehenden Eingabebaum ein und generiert ein PDF-Dokument oder zeigt es auf dem Bildschirm an.
FOP ist Teil des Apache XML-Projektes (http://xml.apache.org/fop)

# 5 RDF: Resource Description Framework

Das vorangegangene bildet einen Grundstein für das Semantic Web, mit XML-Strukturen lassen sich Gegebenheiten abbilden, eine Eindeutigkeit kann durch Namensräume gewährleistet werden und durch XSLT kann die Struktur transformiert werden. Zur Einführung in RDF, welches nun auf dem Grundstein aufbaut, folgt ein Beispiel.

## 5.1 Das Supermarktbeispiel

Dient beispielsweise XML zur Etikettierung der Waren und XML-Schema zur Beschreibung des Aufbaus der Etiketten, können Firmen über diesen einfach zu schaffenden Pseudostandard Waren austauschen.
Problematisch bleibt dann noch die Beschreibung des Restes vom Supermarkt, also z.B. Beziehungen, die zwischen den Waren und dem Supermarkt herrschen. So z.B.

- Ware – hat – Etikett
- Diese Ware – gehört in – Kühlregal

Mit RDF kann man nun Subjekt-Prädikat-Objekt-Sätze bilden, die solche Beziehungen beschreiben können. Zusätzlich gibt es dann noch RDF-Schema, mit dem die "Welt der Artikel im Supermarkt" definiert werden kann. RDFS stellt eine einfache Modellierungssprache dar, in der man domain-abhängig Klassen, Subklassen, Eigenschaften und Subeigenschaften definieren kann. In RDFS gibt es Mehrfachvererbung und man kann Objekte und Wertebereiche einschränken.

## 5.2 RDF: Resource Description Framework

Im rdf-primer (http://www.w3.org/TR/rdf-primer/) wird die RDF-Sprache beschrieben. Der Zweck ist es, ein Modell zu haben, mit dem man Aussagen über vorhandene Ressourcen im Web machen kann. Im wesentlichen handelt es sich um Metadaten wie Autor, Titel, Version, Erstellungsdatum,... aber auch Dinge, die nicht unmittelbar im Web zu finden sind, wie Preise, Spezifikationen, Personen,...
Um diese Informationen, die ja schon menschenlesbar im Web vorhanden sind, nun auch maschinenlesbar zu machen werden sie durch RDF ausgezeichnet. Dazu werden *Uniform Resource Identifications (URI)* benutzt, die, anders als die *Uniform Resource Locators (URL)*, auch immaterielle Ressourcen bezeichnen können. Aussagen über Bezeichnungen von materiellen und imateriellen Ressourcen zueinander werden durch *Subjekt–Prädikat–Objekt*-Sätze gemacht. So werden Ressourcen Eigenschaften und Werte zugeordnet (index.html – Autor – Jan; index.html hat die Eigenschaft Autor, mit dem Wert 'Jan'). Da RDF domänenunabhängig ist, definieren die Benutzer ihre eigenen Terminologien durch das *Resource Description Framework Schema (RDFS).*

### 5.2.1 Sichten auf RDF-Aussagen

Es sei beispielhaft eine Person durch die URI *http://www.w3.org/People/EM/contact#me* identifiziert, deren Name *Eric Miller* ist, ihre E-Mail-Adresse *em@w3.org* lautet und deren Titel *Dr.* ist. Dieses kann auch als Graph mit Knoten für Subject und Objekt und Kanten für Eigenschaften gesehen werden. Hier würde es einen Knoten für *Eric Miller* geben, von dem aus mit *E-Mail*, *Name*, *Titel* beschriftete Kanten als Eigenschaften zu den Knoten mit den Werten dieser Eigenschaften führen würden. Hierbei ist dann noch zu unterscheiden zwischen referenzierbaren URIs (wie E-Mail, Homepage)

und einfachen Werten ('Dr.', 'Eric Miller'), wobei das erstere als *URIref* und das letztere als *Literal* bezeichnet wird. In RDF-Graphen sind URIrefs als Ovale dargestellt und Literale als Rechtecke. Auch kann man das Statement als XML serialisieren

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">

    <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
        <contact:fullName>Eric Miller</contact:fullName>
        <contact:mailbox rdf:resource="mailto:em@w3.org"/>
        <contact:personalTitle>Dr.</contact:personalTitle>
    </contact:Person>
</rdf:RDF>
```

Ein RDF-Dokument fängt als XML-Dokument mit dem üblichen Prolog an und hat als Root-Element `<rdf:RDF>`. Es enthält dann eine Reihe von Beschreibungen über Ressourcen. Es gibt drei Wege Aussagen zu machen

- mittels dem `rdf:about`-Attribut kann eine existierende Resource referenziert werden
- mittels eines `id`-Attributes wird eine neue Resource kreiert
- ohne Name wird eine anonyme Ressource erstellt

Schliesslich kan man RDF-Statement auch als Tripel schreiben, die von der Form $(x, P, y)$ sind, wobei $P$ als binäres Prädikat $P(x, y)$ zu verstehen ist, mit $x$ als Subjekt und $y$ als Objekt. Im obigen Beispiel wäre das Prädikat `mailbox`, mit dem Subjekt `...#me` und dem Objekt `me@w3.org`. RDF erlaubt nur binäre Prädikate.

```
(http://www.w3.org/People/EM/contact#me,
http://www.w3.org/2000/10/swap/pim/contact#mailbox,
em@w3.org)

(http://www.w3.org/People/EM/contact#me,
http://www.w3.org/2000/10/swap/pim/contact#personalTitle,
"Dr.")

(http://www.w3.org/People/EM/contact#me,
http://www.w3.org/2000/10/swap/pim/contact#fullName,
"Eric Miller")

(http://www.w3.org/People/EM/contact#me,
http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
http://www.w3.org/2000/10/swap/pim/contact#Person)
```

### 5.2.2 Reifikation (Verdinglichung)

Die Reifikatoin oder Vergegenständlichung von Statements erlaubt es Aussagen über Aussagen zu machen. Die Aussage

> The Monthy Pythons believe that there is a market for used shrubberies.

wird aufgeteilt in die Aussagen

There is a market for used shrubberies. (1)

und

The Monthy Pythons believe in (1).

Dies wird nötig, da RDF nur Tripel erlaubt. Man vergibt also für eine Aussage eine ID und bezieht sich dann darauf. (Dies härtet die These: "Jedes Problem in der Informatik lässt sich durch die Einführung einer Indirektionsebene lösen.")

### 5.2.3 Datentypen in RDF

Für die Literale benötigt man eine Typisierung, z.B. kann in

```
("Johnny",
http://www.example.org/age,
"99"^^http://www.w3.org/2001/XMLSchema\#integer)
```

die Zahl 99 als Integer typisiert werden. Als Konsequenz ergibt sich daraus, dass RDF externe Typdefinitionen erlaubt. Nützlicherweise können die Datentypen aus XML Schema benutzt werden.

### 5.2.4 Diskussion und Kritik

Folgende Situation

$X$ ist Schiedsrichter im Schachspiel zwischen den Spielern $Y$ und $Z$.

würde zu dem Quadrupel *referee(X,Y,Z)* führen. RDF erlaubt aber nur Tripel. Durch diese Beschränkung müsste man folgendes Modell bilden, um die obige Situation abbilden zu können:

```
ref(chessGame, X)

player1(chessGame, Y)

player2(chessGame, Z)
```

Die Eigenschaften sind als spezielle Ressourcen modelliert und können deshalb als Objekte in Objekt–Attribut–Wert Tripeln genutzt werden, was schön und mächtig ist, aber keinen Standard in Modellierungssprachen darstellt.

Der Mechanismus der Reifikation scheint sehr mächtig in der einfachen RDF-Sprache in der Basisschicht des SemanticWeb.

Die XML-Serialisierung von RDF hat zum Vorteil, dass sie maschinenlesbar ist und zum Nachteil, dass sie nicht unbedingt menschenlesbar ist.

Ressourcen sind wiederverwertbar.

### 5.2.5 RDF-Syntax

Als Rootelement eines RDF-Dokuments wird `rdf:RDF` benutzt, mit dem Namensraum `xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"` und innerhalb folgen eine Reihe von `rdf:descriptions`. Zum Beispiel

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:uni="http://www.example.org"?>

    <rdf:Description rdf:about="http://www.example.org/uni-ns/#949318">
        <uni:name>David Billington</uni:name>
        <uni:title>Associate Professor</uni:title>
    </rdf:Description>

    <rdf:Description rdf:about="http://www.example.org/uni-ns/#CIT1111">
        <uni:courseName>Discrete Mathematics</uni:courseName>
        <uni:isTaughtBy>David Billington</uni:isTaughtBy>
    </rdf:Description>

    ...
</rdf:RDF>
```

Das `about`-Attribut einer Beschreibung hat eine äquivalente Bedeutung zum `ID`-Attribut. Man benutzt konventionellerweise das `about`-Attribut, wenn man sich auf eine vorhandene Ressource bezieht und das `id`-Attribut, wenn eine neue erschaffen wird. RDF ein Graph ist und es gibt keine Referenzen auf etwas, was irgendwo anderes definiert ist. Den Inhalt der Beschreibungen bezeichnet man als *property elements*, welche die Eigenschaft-Werte Paare definieren (uni:name, "David Billington"). Diese property elements müssen verbindend gelesen werden, also hat die Ressource mit der Nummer 949318 den Namen "David Billington" **und** den Titel "Associated Professor".
Um eventuellen Namensgleichheiten aus dem Weg zu gehen, kann man die Ressource "David Billington" mit einer ID definieren und sich dann auf diese beziehen

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:uni="http://www.example.org"?>

    <rdf:Description rdf:ID="#949318">
        <uni:name>David Billington</uni:name>
        <uni:title>Associate Professor</uni:title>
    </rdf:Description>

    <rdf:Description rdf:about="http://www.example.org/uni-ns/#CIT1111">
        <uni:courseName>Discrete Mathematics</uni:courseName>
        <uni:isTaughtBy rdf:resource="#949318"/>
    </rdf:Description>

    ...
</rdf:RDF>
```

Für kurze Ressourcen, wie in diesem Beipiel, empfiehlt es sich die Möglichkeit der Verschachtelung zu nutzen

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:uni="http://www.example.org"?>

    <rdf:Description rdf:about="http://www.example.org/uni-ns/#CIT1111">
        <uni:courseName>Discrete Mathematics</uni:courseName>
```

```
        <uni:isTaughtBy>
            <rdf:Description rdf:ID="#949318">
                <uni:name>David Billington</uni:name>
                <uni:title>Associate Professor</uni:title>
            </rdf:Description>
        </uni:isTaughtBy>
    </rdf:Description>
</rdf:RDF>
```

In dem Beispiel kommen zwei *Typen* von Ressourcen vor, Dozenten und Lehrveranstaltungen. In RDF kann man das durch `rdf:type`-Elemente realisieren

```
...
    <rdf:Description rdf:ID="#949318">
        <rdf:type rdf:resource="http://www.example.org/uni-ns/#lecturer"/>
        <uni:name>David Billington</uni:name>
        <uni:title>Associated Professor</uni:title>
    <rdf:Description>

    <rdf:Description rdf:about="http://www.example.org/uni-ns/#CIT1111">
        <rdf:type rdf:resource="http://www.example.org/uni-ns/#course"/>
        <uni:courseName>Discrete Mathematics</uni:courseName>
        <uni:isTaughtBy rdf:resource="#949318"/>
    </rdf:Description>
...
```

Schliesslich kan man hier noch den Namensraum durch `<rdf:type rdf:resource="&uni;lecturer"/>` abkürzen. Für die Abkürzungen gelten zwei Regeln

1. Kinderlose property elements können durch XML-Attribute ersetzt werden
2. Bei `rdf:Description`-Elementen, die `rdf:type`-Elemente haben, kann man das im Typ deklarierte Element anstelle des `rdf:Description`-Elements benutzen (siehe Beipiel).

In diesem Beispiel

```
...
    <rdf:Description rdf:ID="#CIT1111">
        <rdf:type rdf:resource="&uni;course"/>
        <uni:courseName>Discrete Mathematics</uni:courseName>
        <uni:isTaughtBy rdf:resource="#949318"/>
    </rdf:Description>
...
wird duch Regel 1 zu
...
    <rdf:Description rdf:ID="#CIT1111"
                    uni:courseName="Discrete Mathematics">
        <rdf:type rdf:resource="&uni;course"/>
        <uni:isTaughtBy rdf:resource="#949318"/>
    </rdf:Description>
...
und mit Regel 2 zu
...
    <uni:course rdf:about="#CIT1111"
```

```
                uni:courseName="Discrete Mathematics">
        <uni:isTaughtBy rdf:resource="#949318"/>
    </uni:course>
...
```

RDF kennt nun noch folgende Container-Elemente

- rdf:Bag – eine *ungeordnete* Sammlung von begriffen, die mehrfach vorkommen können
- rdf:Seq – eine *geordnete* Sequenz von Begriffen, die mehrfach vorkommen können
- rdf:Alt – eine Sammlung von Alternaiven, von denen nur eine ausgewählt werden kann

Der Inhalt der Container-Elemente wird dann durch `rdf:_1, rdf:_2, ...` definiert. Die Container-Elemente haben auch noch ein optionales `rdf:ID`-Attribut Beispiel einer `Bags`

```
...
    <rdf:Bag rdf:ID="gehalteneVorlesungen">
        <rdf:_1:rdf:resource="CIT1111"/>
        <rdf:_2:rdf:resource="CIT1112"/>
        ...
    </rdf:Bag>
```

Typische Anwendungen von Container-Elementen ist die Repräsentation von Prädikaten, mit mehr als zwei Argumenten (Schach-Beispiel)

```
...
<referee rdf:about="...#X">
    <players>
        <rdf:Bag>
            <rdf:_1:rdf:resource=".../#Y"/>
            <rdf:_2:rdf:resource=".../#Z"/>
        </rdf:Bag>
    </players>
</referee>
```

Zu beachten ist bei den Containern noch, dass es keine Möglichkeit gibt, einen Container "abzuschliessen" und da RDF ein Graph ist, ist nicht auszuschliessen, dass es irgendwo noch einen anderen Graphen gibt, der weitere Einträge definiert. Zu diesem Zweck gibt es aber neben den Containern noch die *Collections*, welche dies durch das Vokabular `rdf:List, rdf: first, rdf:rest, rdf:nil` verhindern. (Die Listen werden dann wie in Scheme aufgebaut.)

```
...
<rdf:Description rdf:about="CIT1112">
    <uni:isTaughtBy>
        <rdf:List>
            <rdf:first>
                <rdf:Description rdf:about="949318"/>
            </rdf:first>
            <rdf:rest>
                <rdf:List>
                    <rdf:first>
```

```
                    <rdf:Description rdf:about="949322"/>
                </rdf:first>
                <rdf:rest>
                    <rdf:Description rdf:about"&rdf;nil"/>"
                </rdf:rest>
            </rdf:List>
        </rdf:rest>
    </rdf:List>
</uni:isTaughtBy>
</rdf:Description>
...
```

Mit Hilfe des `rdf:parseType` kann diese Beschreibung der Liste abgekürzt werden

```
...
<rdf:Description rdf:about="CIT1112">
    <uni:isTaughtBy rdf:parseType="Collection">
        <rdf:Description rdf:about="949318"/>
        <rdf:Description rdf:about="949322"/>
    </uni:isTaughtBy>
</rdf:Description>
```

Die bereits angesprochene Reifikation (Verdinglichung) kann auch explizit durch RDF ausgedrückt werden, hierzu steht das Vokabular `rdf:Statement, rdf:subject, rdf:predicate, rdf:object` zur Verfügung.

```
...
<rdf:Statement rdf:about="StatementAbout949318">
    <rdf:subject rdf:resource="949318"/>
    <rdf:predicate rdf:resource="&uni;name"/>
    <rdf:object rdf:resource="David Billington"/>
</rdf:Statement>
...
```

## 5.3 RDFS: Resource Description Framework Schema

Das Ziel von RDFS ist es, die Bedeutung von bestimmten Anwendngsdomämen auszudrücken. RDF ist eine universelle Sprache, mit der Benutzer Ressourcen beschreiben können. Hierzu wird das bekannte Konzept von Klassen und Eigenschaften benutzt. Klassen repräsentieren Dinge, über die Aussagen gemacht werden sollen. Die Definition von Eigenschaften erfolgt durch Einschränkungen ihrer Geltungsbereiche (`Discrete Mathematics is taught by David Billington` verlangt die Einschränkung von "`is taught by`" auf den Typ `lecturers`.). Die `subclass-of`-Beziehung ($A$ ist Subklasse von $B$, wenn alle Instanzen von $A$ auch Instanzen von $B$ sind) definiert eine Klassenhierarchie. In RDF können Klassen mehrere Superklassen haben (Mehrfachvererbung). Im Gegensatz zur Objektorientierten Programmierung, wo Eigenschaften in der Klasse definiert sind (neue Eigenschaften hinzufügen heißt, die Klasse zu verändern), sind in RDF die Eigenschaften global definiert und somit nicht innerhalb von Klassen gekapselt. Es ist also möglich Eigenschaften hinzuzufügen, ohne die Klasse zu ändern.

### 5.3.1 Eigenschaftshierarchien

In RDF ist es auch möglich, Eigenschaften zu hierarchiesieren, mit Hilfe des `subproperty-of` so wäre im foglenden Beispiel *"is-taught-by"* eine Subeigenschaft

von *"involves"*. Ein Kurs $k$ wird gehalten von Professor $p$, also gilt $k$ *is-taught-by* $p$ und dann gilt auch $k$ *involves* $p$. Das *"is-taught-by"* hier eine Subeigenschaft von *"involves"* ist, liegt daran, das im Gegenbeispiel ein Tutor t in einen Kurs involviert sein kann, ihn aber nicht hält. Dann gilt nämlich $k$ *involves* $t$, aber $k$ *is-taught-by* $t$ nicht. In beiden Fällen gilt die *involves*-Relation und stellt also die allgemeinere Eigenschaft dar. Während die *is-taught-by*-Relation einen Spezialfall darstellt und somit eine Subeigenschaft ist. Formal kann man sagen, $P$ ist eine Subeigenschaft von $Q$, wenn gilt $P(x, y) \Rightarrow Q(x, y)$.

### 5.3.2 Die RDFS-Sprache

Die formale Sprache, um RDF-Schema zu definieren ist RDF, also Ressourcen und Eigenschaften. Um die Aussage *"lecturer is subClassOf academicStaff"* zu definieren werden zunächst die Ressourcen *"lecturer"*, *"subClassOf"* und *"academicStaff"* definiert, dann wird definiert, dass *"subClassOf"* eine Eigenschaft ist und schließlich kann man das Tripel *"(subClassOf, lecturer, academicStaff)"* hinschreiben.

Als Kernklassen kennt RDFS

- `rdfs:Resource` – Die Klasse aller Ressourcen
- `rdfs:Class` – Die Klasse allerKlassen
- `rdfs:Literal` – Die Klasse aller Literale (Strings sind im Moment der einzige Datentyp von RDF/RDFS)
- `rdf:Property` – Die Klasse aller Eigenschaften
- `rdf:Statement` – Die Klasse aller reifizierten Aussagen

Dann folgen die Kerneigenschaften

- `rdf:type` – definiert die Klasse einer Ressource
- `rdfs:subClassOf` – definiert die Superklasse einer Klasse
- `rdfs:subPropertyOf` – definiert die Supereigenschaft einer Eigenschaft

`rdfs:subClassOf` und `rdfs:subPropertyOf` sind transitiv (z.B. aus $x < y$ und $y < z$ folgt stets $x < z$).

Um Eigenschaften einzuschränken, stehen

- `rdfs:domain` – Spezifiziert die Domäne einer Eigenschaft, also die Klasse von Ressourcen, die als Subjekt auftreten dürfen
- `rdfs:range` – Spezifiziert den Bereich einer Eigenschaft, also die Klassen von Ressourcen, die als Werte auftreten dürfen

zur Verfügung. Auch `rdfs:domain` und `rdfs:range` sind transitiv. Beispiel (domain und range):

```
...
<rdf:Property rdf:ID="phone">
    <rdfs:domain rdf:resource="#staffMember"/>\\
    <rdfs:range rdf:resource="&rdf;Literal"/>
</rdf:Property>
...
```

Weiterhin gibt es

- `rdfs:ConstraintResource` – Die Klasse aller *constraints*
- `rdfs:ConstraintProperty` – Ist definiert als Subklasse von `rdfs:ConstraintResource` und `rdf:Property`. Es gibt nur zwei Instanzen, `rdfs:domain` und `rdfs:range`.

An Eigenschaften für die Reifikation sind

- `rdf:subject` – Das Subjekt des reifizierten Statements
- `rdf:predicate` – Das Prädikat des reifizierten Statements
- `rdf:object` – Das Objekt des reifizierten Statements

vorgesehen.

Weiter stehen die Container `rdf:Bag`, `rdf:Seq` und `rdf:Alt` zur Verfügung und `rdfs:Container` ist ihre Superklasse.
Utility Properties sind

- `rdfs:seeAlso` – setzt eine Ressource in Beziehung mit einer anderen, erklärenden Ressource
- `rdfs:isDefinedBy` – setzt als Subeigenschaft von `rdfs:seeAlso` eine Ressource in Beziehung mit einer anderen Resource, die die Definition enthält
- `rdfs:comment` – erlaubt Kommentare
- `rdfs:label` – erlaubt das kennzeichnen von Knoten

Es folg nun ein Beispiel von Professoren (Lecturers) an einer Universität

```
...
<rdfs:Class rdf:ID="staffMember">
    <rdfs:comment>Dies ist die Klasse staffMember</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:ID="academicStaffMember">
    <rdfs:comment>
        Dies ist die Klasse academicStaffMember;
        Subklasse von staffMember
    </rdfs:comment>
    <rdfs:subClassOf rdf:resource="#staffMember"/>
</rdfs:Class>

<rdfs:Class rdf:ID="course">
    <rdfs:comment>Dies ist die Klasse course</rdfs:comment>
</rdfs:Class>

<rdf:Property rdf:ID="isTaughtBy">
    <rdfs:comment>
        Dies stellt die Relation von Kursen und Dozenten dar.
    </rdfs:comment>
    <rdfs:domain rdf:resource="#course"/>
    <rdfs:range rdf:resource="#academicStaffMember"/>
</rdf:Property>
...
```

# 6 News from the World Wide Web Conference 2004

Dieses Kapitel ist nicht Prüfungsrelevant und wird ggf. nachgereicht, wenn ich noch Zeit habe (... also nie).

# 7 Web Ontology Language - OWL

In diesem Kapitel beginnt die nächste Schicht im SemanticWebTower; die der *"Ontology vocabulary"*. Die Modellierungssprache OWL setzt auf RDF und RDF Schema auf und liefert so erweiterte Beschreibungsmöglichkeiten.

## 7.1 Grundsätzliche Gedanken zu OWL

Sprachen zur Modellierung von Ontologien ermöglichen die explizite, formale Konzeptualisierung von Modellen für Themenbereiche. Dafür benötigt man eine wohldefinierte Syntax, eine effiziente Schussfolgerungsunterstützung (reasoning support), eine formale Semantik und eine ausreichende, aber noch komfortable Expressivität. Hier gilt es die Expressivität der Sprache gegen ihre Berechenbarkeit abzuwiegen. Je größer die Expressivität ist, desto schwieriger gestaltet sich das Problem der Berechenbarkeit. Manchmal wird die Grenze der Berechenbarkeit auch überschritten. Schlussfolgerungen wären zum Beispiel *Klassenzugehörigkeit*, *Klassenäquivalenz*, *Konsistenz* und *Klassifikation*. So kann man aus der Tatsache, dass $x$ Instanz der Klasse $C$ und $C$ Subklasse von $D$ ist schliessen, dass $x$ auch eine Instanz von $D$ ist (Klassenzugehörigkeit). Ebenso muss aus der Tatsache, dass $A$ äquivalent zu $B$ und $B$ äquivalent zu $C$ ist folgen, dass $A$ äquivalent zu $C$ ist (Klassenäquivalenz). Wenn $x$ Instanz der Klassen $A$ und $B$ ist, $A$ und $B$ aber disjunkt sind, dann ist dies eine Indikation für einen Fehler in der Ontologie (Konsistenz). Bestimmte Eigenschaft-Werte Paare liefern ausreichende Bedingungen für eine Zugehörigkeit zur Klasse $A$, wenn also eine Instanz diesen Bedingungen genügt, dann kann man schliessen, dass $x$ eine Instanz von $A$ sein muss (Klassifikation).

Diese Schlussfolgerungen sind wichtig, um die Konsistenz von Ontologien zu prüfen, unbeabsichtigte Beziehungen zwischen Klassen aufzudecken oder automatisch Instanzen zu ihren Klassen zuzuordnen. Diese Tests sind vor allem nützlich, wenn Probleme es erfordern, große Ontologien zu entwickeln, an denen eine vielzahl von Autoren beteiligt sind. Auch bei der Integration oder der gemeinsamen Benutzung von verschiedenen, verteilten Quellen sind diese Tests nützlich. Eine formale Semantik ist eine Grundanforderung für die Folgerungsunterstützung und beides wird üblicherweise durch Abbildung einer Ontologiesprache auf einen existierenden Logikformalismus bewerkstelligt. Ist dies erfolgt, kann man auch die existierenden Schlussfolgerer dieses Formalismusses benutzen. OWL ist (teilweise) auf die Deskriptive Logik abgebildet und als Schlussfolgerer kommen *FaCT* und *RACER* zum Einsatz. Die Deskriptive Logik ist einer Teilmenge der Prädikatenlogik und für die letztere ist effiziente Schlussfolgerung möglich.

RDF Schema findet seine Beschränkung der Ausdruckskraft in der globalen Reichweite von Eigenschaften; `rdfs:range` definiert den Gültigkeitsbereich einer Eigenschaft für alle Klassen, diesen kann man aber nicht auf bestimmte Klassen einschränken. So könnte man zum Beispiel nicht sagen, dass Kühe *nur* Pflanzen essen während andere Tiere auch FLeisch essen können. Auch lässt sich keine Disjunktheit von Klassen definieren (z.B. *Mann* und *Frau*). Manchmal will man aus vorhandenen Klassen duch Vereinigung, Schnitt oder Komplementbildung neue Klassen bilden (*Person* sei die disjunkte Vereinigung von *Mann* und *Frau*), was ebenfalls nicht geht. Kardinalitäten, wie in *"Ein Kurs wird von mindestens einem Lehrer gehalten"* lassen sich ebenfalls nicht abbilden und spezielle Charakteristika wie zum Beispiel die Transitivität (*"größer als"*$(a > b \wedge b > c \Rightarrow a > c)$) oder die Eindeutigkeit (*"istMutterVon"*) von Eigenschaften, sowie die Gegenteiligkeit von Eigenschaften (*"isst"* und *"wirdGegessenVon"*) machen Probleme.

## 7.2 Beschränkungen der Expressivität von RDF Schema

- local scope of properties – `rdfs:range` definiert den Gültigkeitsbereich für alle Klassen und es ist nicht möglich, dies nur für ein paar bestimmte Klassen zu tun
- disjointness of classes – (*male* und *female*)
- boolean combinations of classes – Definition durch Verwendung von `union`, `intersection` und `complement` (`person` sei die disjunkte Vereinigung von `male` und `female`)
- cardinality restricions – Eine Person soll zum Beispiel genau zwei Elternteile haben oder *"Ein Kurs wird von mindestens einem Lehrer gehalten"*
- special characteristics of properties – Dies gilt zum Beispiel für transitive Eigenschaften (*"größer als"*) oder eindeutige Eigenschaften (*isMotherOf*). Aber auch für gegenteilige Eigenschaften wie *"eats"* und *"isEatenBy"*

Idealerweise würde nun OWL eine Erweiterung von RDF-Schema darstellen und so die geschichtete Architektur des Semantic Webs konsistent halten. Allerdings würde man dann nicht genügend an Expressivität gewinnen und effiziente Schlussfolgerungen wären nicht gegeben. Die Verbindung von Logic mit RDF Schema führt zu unkontrollierbaren (Rechen-)Eigenschaften.

## 7.3 OWL

Die *Ontology Working Group* des W3C definiert die *Web Ontology Language* in drei Teilsprachen, *OWL Full*, *OWL D*L und *OWL Lite*, die untereinander aufwärtskompatibel sind. So ist jede gültige OWL Lite Ontologie eine gültige OWL DL Ontologie und eine solche ist eine gültige OWL Full Ontologie. Das gleiche gilt für die Folgerungen.

**Definition 3** *On|to|lo|gie, die: Lehre vom Sein, von den Ordnungs-, Begriffs- u. Wesensbestimmungen des Seienden.*

**Definition 4** *(Ontologie)*

*An ontology is a specification of a conceptualization.*

*In the context of knowledge sharing, I use the term ontology to mean a specification of a conceptualization. That is, an ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set-of-concept-definitions, but more general. And it is certainly a different sense of the word than its use in philosophy. – Tom Gruber gruber@ksl.stanford.edu*

### 7.3.1 OWL Full

*OWL Full*

- benutzt alle Elemente aus OWL
- erlaubt die beliebige Kombination dieser Elemente mit RDF und RDFS
- ist voll aufwärtskompatibel mit RDF, und zwar syntaktisch und semantisch
- ist so mächtig, das es nicht entscheidbar ist. $\rightsquigarrow$ Es gibt keinen vollständigen (oder effizienten) *"reasoning support"* (Schlussfolgerungen).

### 7.3.2 OWL DL

*OWL DL* (Description Logic) ist eine Teilsprache von *OWL Full* und beschränkt die Anwendung der Konstruktoren von OWL und RDF.

- die Anwendung der Konstuktoren von OWL aufeinander ist verboten
- dafür entspricht es der gut studierten Deskriptiven Logik
- OWL DL stellt effiziene Schlussfolgerungsunterstützung bereit
- dafür verliert man aber die vollständige Kompatibilität mit RDF, d.h. insbesondere, dass nicht jedes RDF-Dokument ein gültiges OWL-Dokument ist, aber jedes gültiges OWL-Dokument ein gültiges RDF-Dokument.

### 7.3.3 OWL Lite

*OWL Lite* schänkt OWL DL noch weiter ein, in dem es nicht alle Konstruktoren zulässt. So sind zum Beispiel die Aufzählungsklassen, Aussagen zur Disjunktheit und beliebige Kardinalitäten nicht erlaubt. Vorteile der so entstehenden Sprache sind der einfachere Zugang für neue Benutzer und dass es einfacher ist, Werkzeuge zu implementieren. Der Nachteil ist allerdings die eingeschränkte Expressivität.

Im Allgemeinen lässt sich zur Kompatibilität von OWL zu RDF sagen, dass alle Varianten von OWL die Syntax von RDF benutzen. So werden Instanzen wie in RDF durch Benutzung der `rdf:Description` deklariert und die Konstruktoren von OWL für Typen sind Spezialisierungen der korrespondierenden RDF-Gegenstücke. Der Entwurf des Semantic Web ziehlt auf eine Abwärtskompatibliliät ab. Dies ist nur für OWL Full erfüllt, auf Kosten der Unattraktivität.

### 7.3.4 Die Syntax der Sprache OWL

Die *Web Onology Language* baut auf dem *Resource Description Framework* auf und benutzt dessen XML-basierte Syntax. Es sind aber auch noch andere Syntaxen für OWL definiert worden, so eine lesbarere Alternative auf Basis von XML, eine abstraktere, viel kompaktere und lesbarere Syntax und eine graphische, die auf den Konventionen von UML basiert. Im Folgenden werden nun die einzelnen Komponenten der OWL vorgestellt. Es gibt im Einzelnen

Klassen, Eigenschaften, Datentypeigenschaften, Objekteigenschaften, Inverse Eigenschaften, Gleichheitseigenschaften, Beschränkung von Eigenschaften, Beschränkung von Kardinalitäten, Spezielle Eigenschaften, Boolesche Kombinationen (Verschachtelung), Aufzählungen, Deklaration von Instanzen, No Unique Names Assumption (Annahme), Verschiedenheit von Objekten, Datentypen in OWL, Versionierungs Informationen, Kombination von Merkmalen, Einschränkung von Merkmalen in OWL DL und Vererbung in Klassenhierarchien.

Eine Ontologie fängt mit dem Header, in dem die benötigten Namensräume deklariert werden und dem `owl:Ontologie`-Element an

```
<rdf:RDF
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://ww.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

    <owl:Ontology rdf:about="">
```

```
        <rdfs:comment>Beispiel für eine OWL Ontologie </rdfs:comment>
        <owl:priorVersion
            rdf:resource="http://www.example.org/onto-old"/>

        <!-- imports ist eine transitive Eigenschaft -->
        <owl:imports rdf:resource="http://www.example.org/persons"/>
        <rdfs:label>University Ontology</rdfs:label>
...
```

**Klassen** werden durch das `owl:Class`-Element definiert, welches Subklasse von `rdfs:Class` ist. Eine eventuelle Disjunktheit kann durch `owl:disjointWith` deklariert werden und `owl:equivalentClass` definiert die Gleichheit von Klassen. `owl:Thing` ist die oberste Klasse in der Hierarchie und enthält alles (`Object` in Java), während `owl:Nothing` die leere Klasse repräsentiert.

```
...
<owl:Class rdf:about="#associateProfessor">
    <owl:disjointWith rdf:resource="#professor"/>
    <owl:disjointWith rdf:resource="#assistantProfessor"/>
</owl:Class>

<owl:Class rdf:ID="faculty">
    <owl:equivalentClass rdf:resource="#academicStaffMember"/>
</owl:Class>
...
```

In OWL werden zwei Arten von **Eigenschaften** unterschieden. Die Objekteigenschaften setzen Objekte zueinander in Beziehung (`isTaughtBy` oder `supervises`) und die Datentypeigenschaften, welche Objekten ihre Datentypen zuweisen (phone, title, age,...). Ausserdem ist es möglich, durch `owl:inverseOf` die Richtung von Beziehungen zu verdeutlichen, sowie mit `owl:equivalentProperty` Synonyme zu schaffen.

```
...
<owl:DatatypeProperty rdf:ID="age">
    <rdfs:range
        rdf:resource="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="isTaughtBy">
    <owl:domain rdf:resource="#course"/>
    <owl:range rdf:resource="#academicStaffMember"/>
    <rdfs:subPropertyOf rdf:resource="#involves"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="teaches">
    <rdfs:domain rdf:resource=  "#academicStaffMember"/>
    <rdfs:range rdf:resource="#course"/>
    <owl:inverseOf rdf:resource="#isTaughtBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="lecturesIn">
    <owl:equivalentProperty rdf:resource="#teaches"/>
</owl:ObjectProperty>
...
```

Nun soll es möglich sein, Eigenschaften einzuschränken. Hierzu steht das Element `owl:Restriction` zur Verfügung, welches ein `owl:onProperty`-Element beinhaltet und ein oder mehrere Einschränkungsdeklarationen. Hier gibt es zum einen den Typ der Kardinalitätseinschränkung und zum anderen drei Typen von Einschränkungen auf die Werte, die die Eigenschaft einnehmen darf:

- `owl:allValuesFrom` spezifiziert die universale Quantifikation
- `owl:hasValue` spezifiziert einen bestimmten Wert
- `owl:someValuesFrom` spezifiziert existentielle Quantifikation

```
...
<owl:Class rdf:about="#firstYearCourse">
    <rdfs:subClassOf>
       <owl:Restriction>
        <owl:onProperty rdf:resource="#isTaughtBy"/>
        <owl:allValuesFrom rdf:resource="#Professor"/>
      </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#mathCourse">
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#isTaughtBy"/>
                <owl:hasValue rdf:resource="#949352"/>
            </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#academicStaffMember">
    <rdfs:subClassOf>
       <owl:Restriction>
        <owl:onProperty rdf:resource="#teaches"/>
        <owl:someValuesFrom rdf:resource="#undergraduateCourse"/>
       </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
...
```

Kardinalitäten werden über die Angabe von Minimum (`owl:minCardinality`) und Maximum (`owl:maxCardinality`) angegeben. Will man einen bestimmten Wert angeben, so kann man das erreichen, in dem man für das Minimum und das Maximum den gleichen Wert angibt. OWL liefert allerdings auch als Komfort das `owl:cardinality`-Element. Will man dagegen eine Art "unbounded" erreichen, so lässt man das Max- bzw. das Minimum weg.

**Spezielle Eigenschaften** werden in OWL über die folgenden Möglichkeiten realisiert

- `owl:TransitiveProperty` – zum Beispiel *"is greater than"* oder *"has better grade than"*
- `owl:SymmetricProperty` – zum Beispiel *"has same grade like"* oder *" is sibling of"*

- `FunctionalProperty` – definiert eine Eigenschaft, die höchstens einen Wert für ein Objekt hat (*"age"*, *"height"*,...)
- `owl:InverseFunctionalProperty` – definiert eine Eigenschaft, sodass zwei verschiedene Objekte nicht den gleichen Wert annehmen können

Über boolesche Kombinationen (Vereinigung, Schnitt, Komplement) lassen sich aus vorhandenen neue Klassen erzeugen.

```
...
<owl:Class rdf:about="#course">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:complementOf rdf:resource="#staffMember"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="peopleAtUni">
    <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#staffMember"/>
        <owl:Class rdf:about="#student"/>
    </owl:unionOf>
</owl:Class>

<owl:Class rdf:ID="facultyInCS">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#faculty"/>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#belongsTo"/>
            <owl:hasValue rdf:resource="#CSDepartment"/>
        </owl:Restriction>
    </owl:intersectionOf>
</owl:Class>
...
```

Die booleschen Operationen lassen sich auch verschachteln, das ist sinnvoll, wenn zum Beispiel der Schitt aus der Menge der `#staffMember` und dem Komplement der Vereinigung von `#faculty` und `#techSupportStaff` gebildet werden soll

```
...
<owl:Class rdf:ID="adminStaff">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#staffMember"/>
        <owl:complementOf>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#faculty"/>
                <owl:Class rdf:about="#techSupportStaff"/>
            </owl:unionOf>
        </owl:complementOf>
    </owl:intersectionOf>
</owl:Class>
...
```

Aufzählungen (Enumerations) lassen sich mit `owl:oneOf` realisieren

```
...
<owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#Monday"/>
        <owl:Thing rdf:about="#Tuesday"/>
        <owl:Thing rdf:about="#Wednesday"/>
        <owl:Thing rdf:about="#Thursday"/>
        <owl:Thing rdf:about="#Friday"/>
        <owl:Thing rdf:about="#Saturday"/>
        <owl:Thing rdf:about="#Sunday"/>
</owl:oneOf>
...
```

Instanzen von Klassen werden wie in RDF erstellt

```
...
<rdf:Description rdf:ID="949352">
    <rdf:type rdf:resource="#academicStaffMember"/>
</rdf:Description>
<academicStaffMember rdf:ID="949352">
        <uni:age rdf:datatype="&xsd;integer">   39<uni:age>
</academicStaffMember>
...
```

In Datenbanksystemen geht man von eindeutigen Namen bzw IDs aus (*Unique Names Assumption*). In OWL wird aus der Tatsache, das zwei Instanzen verschiedene Namen oder IDs haben **nicht** gefolgert, dass es sich um verschiedene Individuen handelt. So würde ein *reasoner* beispielsweise keinen Fehler melden, wenn definiert ist, dass ein Kurs von genau einem Dozenten gehalten wird, es aber zwei Dozenten zu diesem Kurs gibt. Stattdessen wird er annehmen, das diese beiden Ressourcen äquivalent sind. Um nun zu gewährleisten, das sich zwei Induviduen voneinander unterscheiden, deklariert man mittels

```
...
<lecturer rdf:about="949318">
    <owl:differentFrom rdf:resource="949352"/>
</lecturer>
...
```

ihren Unterschied. Für mehrere verschiedene Ressourcen gibt es die Möglichkeit, eine Menge von paarweise disjunkten Elementen zu deklarieren

```
...
<owl:allDifferent>
    <owl:distinctMembers rdf:parseType="Collection">
        <lecturer rdf:about="949318"/>
        <lecturer rdf:about="949352"/>
        <lecturer rdf:about="949111"/>
    </owl:distinctMembers>
</owl:allDifferent>
...
```

Während man XML Schema seine eigenen Datentypen zum Beispiel durch Einschränkung von vorhandenen erstellen konnte (`adultAge` sind alle Integer größer als 18) ist dies in OWL nicht möglich. Die OWL Referenzdokumentation listet alle Datentypen aus XML Schema auf, die benutzt werden dürfen. Darunter befinden sich die oft genutzten Typen `string`, `integer`, `Boolean`, `time` und `date`.

Um *Versionierungsinformationen* zu deklarieren stehen die Elemente `owl:priorVersion` und `owl:versioninfo` zur Verfügung. Zusätzlich gibt es `owl:backwardCompatibleWith`, was eine Referenz zu einer andern Ontologie enthält und `owl:incompatibleWith` ebenfalls mit einer Referenz auf eine andere Ontologie, zur der die vorliegende Version nicht kompatibel ist.

In den verschiedenen OWL Sprachen gibt es verschiedene Beschränkungen hinsichtlich der Anwendung von Fähigkeiten. In **OWL Full** können alle Konstruktoren in allen möglichen Kombinationen benutzt werden, solange das Ergebnis gültiges RDF ist. In **OWL DL** gibt es folgende Beschränkungen

- Vocabulary partitionung
- Explicit typing
- PropertySeparation
- No transitive cardinality restrictions
- Restricted anonymous classes

In **OWL Lite** gelten die Beschränkngen von OWL DL und zusätzlich

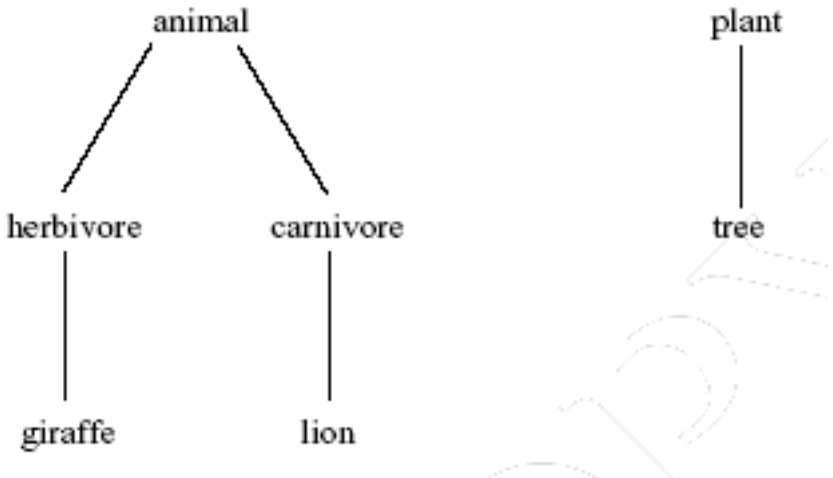
- sind die Elemente `owl:oneOf, owl:disjointWith, owl:unionOf, owl:complementOf und owl:hasValue` nicht erlaubt.
- können Aussagen zur Kardinalität (minimal, maximal, exakt) nur mit den Werten 0 und 1 gemacht werden.
- kann die Aussage mittels `owl:equivalentClass` nur zwischen Klassenbezeichnern (ClassID) gemacht werden und nicht mehr zwischen anonymen Klassen.

## 7.4 Beispiele

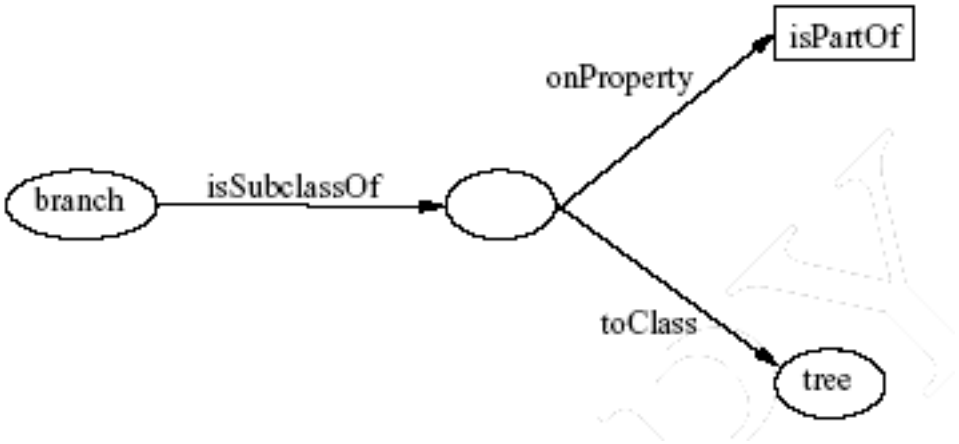
Es werden nun zwei Beispiele für Ontologien vorgestellt, zum einen eine Ontologie über die Tierwelt in Afrika und dann noch eine über Drucker.

### 7.4.1 African Wildlife Ontology

Zur *,, African Wildlife Ontology"* stelle man sich folgende Klassenhierarchie vor



Die Aussage ,,Äste sind Teile von Bäumen" kann man dann folgendermaßen modellieren




Nun werden einige grundlegende Beziehungen und die Klassen ,,Pflanze" und ,,Baum" definiert.

```
<!-- einige drundlegene eigenschaften -->
<owl:TransitiveProperty rdf:ID="is-part-of"/>

<owl:ObjectProperty rdf:ID="eats">
    <rdfs:domain rdf:resource="#animal"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="eaten-by">
    <owl:inverseOf rdf:resource="#eats"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="plant">
    <rdfs:comment>Plants are disjoint from animals. </rdfs:comment>
    <owl:disjointWith="#animal"/>
</owl:Class>
<owl:Class rdf:ID="tree">
    <rdfs:comment>Trees are a type of plant. </rdfs:comment>
    <rdfs:subClassOf rdf:resource="#plant"/>
</owl:Class>
```

Um nun die Aussage über Äste zu machen geht man wie folgt vor

```
<owl:Class rdf:ID="branch">
    <rdfs:comment>Branches are parts of trees. </rdfs:comment>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#is-part-of"/>
            <owl:allValuesFrom rdf:resource="#tree"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

und Blätter ergeben sich durch

```
<owl:Class rdf:ID="leaf">
    <rdfs:comment>Leaves are parts of branches. </rdfs:comment>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#is-part-of"/>
            <owl:allValuesFrom rdf:resource="#branch"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Carnivoren (Fleischfresser) durch

```
<owl:Class rdf:ID="carnivore">
    <rdfs:comment>
        Carnivores are exactly those animals that eat also animals.
    </rdfs:comment>
    <owl:intersectionOf rdf:parsetype="Collection">
        <owl:Class rdf:about="#animal"/>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#eats"/>
            <owl:someValuesFrom rdf:resource="#animal"/>
        </owl:Restriction>
    </owl:intersectionOf>
</owl:Class>
```

Herbivoren (Pflanzenfresser) durch

```
<owl:Class rdf:ID="herbivore">
        <rdfs:comment>
            Herbivores are exactly those animals
            that eat only plants or parts of plants.
        </rdfs:comment>
        <owl:intersectionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#animal"/>
            <owl:restriction>
                <owl:onProperty rdf:resource="#eats"/>
                <owl:someValuesFrom rdf:resource="#plant"/>
            </owl:restriction>
        </owl:intersectionOf>
</owl:Class>
```

Die Giraffe ergibt sich nun als Subklasse der Herbivoren und der Löwe als Subklasse der Carnivoren

```
<owl:Class rdf:ID="giraffe">
    <rdfs:comment>
        Giraffes are herbivores, and they eat only leaves.
    </rdfs:comment>
    <rdfs:subClassOf rdf:type="#herbivore"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#eats"/>
            <owl:allValuesFrom rdf:resource="#leaf"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

```
<owl:Class rdf:ID="lion">
    <rdfs:comment>
        Lions are animals that eat only herbivores.
    </rdfs:comment>
    <rdfs:subClassOf rdf:type="#carnivore"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#eats"/>
            <owl:allValuesFrom rdf:resource="#herbivore"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

Zum Abschluss noch die Klasse der ,,Tasty plants"

```
<owl:Class rdf:I="tasty-plants">
    <rdfs:comment>
        Plants eaten by herbivores and carnivores
    </rdfs:comment>
    <rdfs:subClassOf rdf:type="#plant"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#eaten-by"/>
            <owl:someValuesFrom rdf:resource="#herbivore"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#eaten-by"/>
            <owl:someValuesFrom rdf:resource="#carnivore"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

### 7.4.2 Printer Ontology

folgt ...

## 7.5 Der Namensraum von OWL

Ein Teil von OWL ist in OWL geschrieben, so ist die Superklasse aller OWL-Klassen (`Thing`) von der Superklasse aller RDF-Klassen abgeleitet. Die Klasse `Nothing` repräsentiert die leere Klasse. Es gelten die folgenden Beziehungen

$$\begin{aligned} Thing &= Nothing \cap \overline{Nothing} \\ Nothing &= \overline{Thing} = \overline{Nothing \cap \overline{Nothing}} = \overline{Nothing} \cup \overline{\overline{Nothing}} = \emptyset. \end{aligned}$$

Es werden noch weitere Klassen und Eigenschaften definiert, wie zum Beispiel `EquivalentCLass` und `EquivalentProperty` als Subeigenschaft von `suClassOf` bzw `subPropertyOf`. Die Disjunktheit von zwei Klassen wird als Eigenschaft `disjointWith` definiert, mit der `rdfs:domain rdf:resource="#Class"` und `rdfs:range rdf:resource="#Class"`. Eine Aussage über Gleichheit und Ungleichheit kann zwischen beliebigen Dingen `Thing` getroffen werden und in OWL Full kann solch eine Aussage auch auf Klassen angewendet werden. Als Eigenschaften gibt es

dann `sameIndividualAs, sameAs` und `differentFrom`. Die Vereinigung und der Schnitt von Klassen ergeben neue Klassen

```
<rdf:Property rdf:ID="unionOf">
        <rdfs:domain rdf:resource="#Class"/>
        <rdfs:range rdf:resource="&rdf;List"/>
</rdf:Property>
```

Die Klasse `Restriction` ist als Subklasse von `Class` definiert und dient der Beschreibung von Klassen, die bestimmten Bedingungen genügen. Innerhalb einer `Restriction` können dann die Eigenschaften `onProperty`, `allValuesFrom`, `minCardinality`,... auftreten.

## 7.6 Zukünftige Erweiterungen

Modules and Imports
OWL erlaubt nur den Import ganzer Ontoloigen und nicht Teile davon. Die Definition eines geeigneten Modularisierungsmechanismusses ist eine offene Frage für die Ontologiesprachen. Hier gilt es Aspekte wie das ,,Information Hiding" zu berücksichtigen.

Defaults
Viele praktische Informationssysteme erlauben in Subklassen das Überschreiben von geerbten Werten durch neue, präzisere Werte. Diese sollten dann als Defaultwerte angesehen werden. Allerdings gibt es nocht keinen Konsens über einen richtigen Formalismus zur Beschreibung dieses Problems.

Closed World Assumption
OWL folgt momentan der ,,open-world assumption", dass heisst: Eine Aussage kann nicht als wahr angenommen werden, wenn sie nicht bewiesen werden kann. Dies trifft im riesigen Netz, das nur teilweise bekannt sein kann auch zu. Im Gegensatz dazu steht die ,,closed-world assumption", dass heisst: Eine Aussage ist wahr, wenn ihre Negation nicht bewiesen werden kann.

Unique Names Assumption
Anders als in der Datenbankwelt geht OWL nicht davon aus, dass Dinge mit verschiedenen Namen auch wirklich verschiedene Instanzen sind. OWL folgt dem üblichen Logikparadigma, wo das nicht der Fall ist (Es gibt durchaus verschiedene Dinge, die den gleichen Namen haben) was im WWW ja auc hsinnvol ist.

Procedural Attachements
Ein gebräuchlches Konzept in Wissensrepräsentation ist es, die Bedeutung eines Terms durch anhängen eines erklärenden Quelltextstücks zu definieren. Allerdings nicht durch explizite Definitionen der Sprache. ,,Although widely used, this concept does not lend itself very well to integration in a system with a formal semantics, and it has not been included in OWL "

Rules of Property Chaining
Aus Gründen der Entscheidbarkeit erlaubt OWL die Komposition von Eigenschaften nicht. Allerdings ist dies in vielen Anwendungen eine nützliche Operation. Die Definition von Eigenschaften als generelle Regeln für andere Eigenschaften (z.B. als Hornklauseln) ist sicherlich wünschenswert und die Integration von regelbasierter Wissensrepräsentation oder der der Deskriptiven Logik ist Inhalt der momentanen Forschung.

Summary
OWL ist der vorgeschlagene Standard für Ontologien. Er baut auf RDF und RDF Schema auf. Er hat ist somit eine xml-basierte Syntax und Instanzen werden durch RDF-Deskriptionen definiert. Die meisten RDFS-Primitiva werden benutzt. Die formale Semantik und eine Folgerungsunterstützung (reasoning support) werd durch Abbildung auf die Logik erreicht. Hierbei kommt die Prädikatenlodik und die Deskriptive Logik zum Einsatz. Während OWL derzet ausreichend für die Praxis ist, werden Erweiterungen entwickelt, die mehr logische Fähigkeiten inklusive Regeln bringen werden.

# 8 Einführung in die Technik der Ontologieentwicklung

Dieses Kapitel beschreibt die Möglichkeiten und Wege, Ontologien zu entwickeln.

Es scheint sinnvoll. den folgenden dogmatischen Regeln bei der Entwicklung von Ontologien zu folgen

1. Es gibt keinen *korrekten* Weg, eine Domaine zu modellieren. Es gibt immer mehrere Alternativen und die bester Lösung hängt auch immer von der zu entwickelnden Anwendung und den denkbaren Erweiterungen für diese Anwendung ab.

2. Die Entwicklung von Ontologien ist notwendigerweise ein iterativer Prozess.

3. Die Konzepte einer Ontologie sollten sich so nah wie möglich an den beschriebenen (logischen oder physischen) Objekten und den Beziehungen unter ihnen orientieren. Meistens erkennt man diese aus den Nomen (Objekte) und Verben (Relationen) in den Sätzen, die die Domaine beschreiben.

## 8.1 Schritte zu einer Ontologie

Es folgt eine Folge von Schritten, die man als Weg zu einer Ontologie befolgen kann

### 8.1.1 Bestimmung des Bereichs der Ontologie

In diesem Schritt stelle man sich eine Reihe von Fragen, um sich über die Domaine und die Reichweite der zu erstellenden Ontologie bewusst zu werden (*Was ist die Domaine der Ontologie? Wofür soll die Ontologie benutzt werden? Für welche Art von Fragen soll die Ontologie Antworten geben? Wer wird die Ontoligie benutzen und warten?*). Ein Weg, den Zuständigkeitsbereich einer Ontologie zu bestimmen ist es, Kompetenzfragen zu stellen. Diese Fragen können ausserdem später genutzt werden, ob zu überprüfen, ob die Ontologie genug Informationen enthält, um diese Fragen zu beantworten.

Ein Beipiel: Die Weindomaine

- Welche Charakteristika von Wein soll ich berücksichtigen, wenn ich einen Wein aussuche?

- Ist Bordeaux ein Weiß- oder Rotwein?

- Kann man einen *Cabernet Sauvignon* zu Fisch trinken?

- Welchen Wein trinkt man am besten zu gegrilltem Fleisch?

### 8.1.2 Wiederbenutzung von existierenden Ontologien

In diesem Schritt sucht man das Web nach Ontologien, die den eigenen Anforderungen gerecht werden könnten (*Code — where ever you can steal it.*). Mögliche Anlaufstellen sind zum Beispiel:
Ontolingua ontology library (http://www.ksl.stanford.edu/software/ontolingua)
Open Directory Project (www.dmoz.org)
...

### 8.1.3 Aufzählung von wichtigen Termen der Ontologie

In diesem Schritt schreibt man sich am besten eine Liste der Terme, die in der zu beschreibenden Domaine vorkommen auf. Dies sind all die Dinge, über die man später Aussagen machen will. In unserem Beipiel:
`wine, grape, winery, location, flavor,... a wine's color, types of food, subtypes of wine,...`

### 8.1.4 Defninition der Klassen und der Klassenhierarchie

In diesem Schritt gibt es zwei Ansätze, einen top-down- und einen bottom-up-Ansatz.
Der top-down-Ansatz startet mit den generellen Konzepten einer Domaine und verfeinert diese anschliessend (`wine, food` $\leadsto$ `wine(white, red)` $\leadsto$ `redwine(bordeaux, chianti, ...)`.
Der bottom-up-Ansatz startet mit der spzialisiertesten Klassen und generalisiert dann durch Gruppierung („Paulliac und Margaux sind Subklassen von Bordeaux, ...").

In der Realität wird man eine Mischung aus beiden Ansätzen wählen oder beide durchspielen und dann eine Synthese aus beiden erstellen.

### 8.1.5 Definition der Eigenschaften der Klassen

In diesem Schritt werden die Eigenschaften der Klassen aus der Beschreibung herausgearbeitet. Die Terme, die Klassen darstellen wurden ja schon verarbeitet, die meisten verbleibenden Terme sollten nun Eigenschaften der Klassen sein (a wine's color, body, flavor, sugarcontent,...). Zusätzlich lassen sich nun Typen von Objakteigenschaften definieren

- intrinsische (innewohnende) Eigenschaften, wie `flavor` (Aroma)
- extrinsische (von aussen kommende) Eigenschaften, wie der Name des Weins
- Teile, wie die Gänge eines Gerichtes
- Beziehungen zu anderen Individuen, wie der Hersteller des Weins, oder die Traubensorte des Weins

### 8.1.6 Definition von Facetten (Kardinalitäten, Typen)

Bis zu diesem Schritt würde die Expressivität von RDF / RDF Schema ausreichen. Nun sollen allerdings Kardinalitäten (Wein hat genau eine Farbe), Typen (Preis ist eine Nummer), Domaine und Bereich (ein Weingut befindet sich in der Domaine `produces`), sowie Beziehungen (Symmetrie, Transitivität, inverse Eigenschaften,..)definiert werden.
Als grundsätzliche Regel gilt hier, dass man bei der Definition von `domain` oder `range` die Klasse finden sollte, die am allgemeinsten ist oder die Domaine oder doe range ausfüllen kann. Auf der anderen Seite sollte man auch nicht eine zu allgemeine Klasse wählen, da diese dann ihre Rolle nicht richtg ausfüllen würde.
Zu prüfen ist dann im Folgenden

- enthält die Menge von Klassen, die Gültigkeitsbereiche einer Eigenschaft definieren eine Klasse und ihre Subklasse, dann sollte die Subklasse entfernt werden.
- sind hingegen alle Subklassen einer Klasse $K$ enthalten, $K$ selbst aber nicht, dann sollten alle Subklassen durch $K$ ersetzt werden.

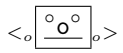
- enthält die Menge beinahe alle Subklassen von $K$, kann dies ein Indiz dafür sein, dass $K$ einen größeren Geltungsbereich haben sollte.

### 8.1.7 Erschaffung von Instanzen

In diesem letzten Schritt bei der Erstellung von Ontologien werden Instanzen erstellt. Man sucht sich eine Klasse, kreiert eine individuelle Instanz und füllt sie mit Werten. Zum Beispiel einen Chateau-Morgan-Beaujolais, um einen bestimmten Typ von Beaujolais zu repräsentieren.

### 8.1.8 Test auf Anomalien

In diesem Schritt wird de erstellte Ontologie auf Anomalien überprüft. Hier gilt es zu prüfen, ob die Klassenhierarchie korrekt ist (`is-a` Beziehungen, Zyklen,...). Weiterhin findet ein Analyse der Nachbarn in der Hierarchie statt (die Nachbarn sollten alle auf der gleichen Abstraktionsebene sein). Inkompatible `domain` und `range` Definitionen für transitive, symmetrische oder inverse Eigenschaften sollten aufgedeckt werden. Weiterhin sind Kardinalitätseigenschaften eine häufige Fehlerquelle.



# 9 The Logic Layer of the Semantic Web

Dieses Kapitel baut auf dem Kapitel ,,Logic and Inference: Rules" in dem Buch ,,A Semantic Web Primer" von G. Antoniou ud F.v. Harmelen auf.

## 9.1 Logical languages

Bisher ging es hauptsächlich um die Präsentation von Wissen, also von Wissen über den Inhalt von, im Web befindlichen, Ressourcen und über die Konzepte einer speziellen Domaine (Ontologie). Schon Aristoteles, der als Vater der Logik angesehen wird, beschäftigte sich mit diesem Problem. So lässt sich Wissensrepräsentation über das WWW, die Künstliche Intelligenz und die Philosophie bis hin zu den alten Griechen zurückverfolgen. Logik ist immer noch die Grundlage zur Wissensrepräsentation und insbesondere tritt hier die *Prädikatenlogik* (auch *First Order Logik (FOL)*) in Erscheinung.
Die wesentlichen Vorteile der Logik sind:

- Sie liefert eine Hochsprache von hoher Expressivität, mit der Wissen in einer transparenten Art und Weise ausgedrückt werden kann.
- Sie hat eine gut verstandene Semantik, mit der logischen Aussagen ihre Bedeutung unmissverständlich zugeordnet werden kann.
- Es gibt eine präzise Vorstellung von logischer Konsequenz.
- Es existieren Beweissysteme, die aussagekräftig/fundiert und vollständig sein sollten
  - aussagekräftig/fundiert — Alle abgeleiteten Aussagen folgen aus den Voraussetzungen.
  - vollständig — Alle logischen Konsequenzen der Voraussertzungen können von de m Logiksystem abgeleitet werden.
- Die existierenden Beweissysteme erlauben die automatische Ableitung von Aussagen aus einer Menge von Voraussetzungen (syntaktisch)
- Die Bedeutung einer logischen Sprache liegt in der Wahrheit eines Satzes.
- **Für FOL/Prädikatenlogik gibt es aussagekräftige und vollstandige Beweissysteme**; dies macht sie so Wertvoll.

Die Prädikatenlogik ist einzigartig in dem Sinne, dass für sie Beweissysteme existieren. Diese Tatsache gilt für Sprachen höherer Expressivtät nicht (RDF und OWL(DL+Lite) können als Spezialisierung der Prädikatenlogik angesehen werden.).

### 9.1.1 Boolesche Logik

Die BNF(Backus Naur Form)-Grammatik

$$\begin{aligned}
Sentence &\rightarrow AtomicSentence \mid ComplexSentence \\
AtomicSentence &\rightarrow True \mid False \mid Symbol \\
Symbol &\rightarrow P \mid Q \mid R \mid ... \\
ComplexSentence &\rightarrow \neg\, Sentence \\
&\quad \mid (Sentence \ \wedge \ Sentence) \\
&\quad \mid (Sentence \ \vee \ Sentence) \\
&\quad \mid (Sentence \ \Rightarrow \ Sentence) \\
&\quad \mid (Sentence \ \Leftrightarrow \ Sentence)
\end{aligned}$$

und die Wahrheitstabelle für Boolesche Logik

| $P$ | $Q$ | $\neg P$ | $P \wedge Q$ | $P \vee Q$ | $P \Rightarrow Q$ | $P \Leftrightarrow Q$ |
|---|---|---|---|---|---|---|
| false | false | true | false | false | true | true |
| false | true | true | false | true | true | false |
| true | false | false | false | true | false | false |
| true | true | false | true | true | true | true |

Tabelle 1: Wahrheitstabelle für Boolesche Logik

Mit diesem Handwerkzeug lässt sich nun die Prädikatenlogik beschreiben

### 9.1.2 FOL – First Order Logik , Prädikaten Logik

$$\begin{aligned}
Sentence &\rightarrow AtomicSentence \\
&\quad \mid (Sentence\ Connective\ Sentence) \\
&\quad \mid Quantifier\ Veriable, ...\ Sentence \\
&\quad \mid \neg Sentence \\
AtomicSentence &\rightarrow Predicate(Term, ...) \mid Term = Term \\
Term &\rightarrow Function(Term, ...) \\
&\quad \mid Constant \\
&\quad \mid Variable \\
Connective &\rightarrow \Rightarrow \ \mid \ \wedge \ \mid \ \vee \ \mid \ \Leftrightarrow \\
Quantifier &\rightarrow \forall \mid \exists \\
Constant &\rightarrow y \mid x \mid john \mid ... \\
Variable &\rightarrow Y \mid X \mid S \mid ... \\
Predicate &\rightarrow Before \mid HasColor \mid Raining \mid ... \\
Function &\rightarrow Mother \mid LeftLeg \mid ...
\end{aligned}$$

### 9.1.3 DL – Deskriptive Logik

Die Deskriptive Logik ist eine Teilmenge der Prädikatenlogik (FOL). Ihre Notation ist so entwickelt worden, dass es einfacher wird, *Definitionen* und *Eigenschaften* von Kategorien zu beschreiben. die Aufgaben der Schlussfolgerung sind

- **Zusammenfassung** von Kategorien und Unterkategorien durch vergleichen ihrer Definitionen
- **Klassifikation** Testen, ob ein Objekt zu einer Kategorie gehört
- manchmal auch **Konsistenz** Test, ob die Definition der Zugehörigkeit zu einer Kategorie logisch befriedigend ist

Normalerweise fehlt der Deskriptiven Logik die Negation und die Disjunktion ganz, oder diese Begriffe sind nur in einem limitierten Form erlaubt. Dafür gibt es aber effiziente Beweissysteme für die Deskriptive Logik.

In der Logiksprache CLASSIC lassen sich beispielsweise Definitionen der Form $Bachelor = AND(Unmarried, Adult, Male)$. machen. Dies wäre äquivalent zu dem folgenden Ausdruck der Prädikatnelogik $Bachelor(x) \Leftrightarrow (Unmarried(x) \wedge Adult(x) \wedge Male(x))$

### 9.1.4 Rule Systems – Hornlogik Logikprogramme

**Definition 5** *(Hornklauseln (Rules))*
*Eine Regel ist von der Form $A_1, ..., A_n \rightarrow B$, wobei $A_i$, $B$ atomare Formeln sind. $B$ ist der Kopf der Formel und $A_1$, $A_2$, ... sind die durch Konjunktion (und) verbundenen Voraussetzungen. Weiter werden alle Variablen, die in einer Regel auftauchen nach vorne gezogen und so umgeformt, dass nur noch der Allquantor ($\forall$) vorkommt.*

Es gilt also $\forall X_1 ... \forall X_k((B_1 \wedge ... \wedge B_n) \rightarrow A)$. Weiter definiert man

**Definition 6** *(Fakt, Logikprogramm, Ziel)*
*Ein Fakt ist eine atomare Formel (loyalUser(a2384574)).*
*Ein Logikprogramm ist eine endliche Menge von Fakten und Regeln.*
*Ein Ziel markiert eine Frage, die an ein Logikprogramm gestellt wird (Ein Ziel stellt eine Anfrage an ein Logikprogramm dar.).*

Um zu zeigen, dass ein Ziel positiv von einem Logikprogramm beantwortet werden kann, negiert man es und zeigt, dass dann ein Widerspruch aus dem Ziel folgt.

### 9.1.5 Monotonic rule systems

In einem Monotonen Regelsystem bleibt eine Konklusion (Schluss) gültig, wenn sie einmal gezeigt werden konnt, auch wenn neues Wissen hinzukommt.

Beispiel: Familienbeziehungen
Fakten:

| | |
|---|---|
| $mohter(X,Y)$ | $X$ is mother of $Y$ |
| $father(X,Y)$ | $X$ is father of $Y$ |
| $male(X)$ | $X$ is male |
| $female(X)$ | $X$ is female |

Regeln zur Ableitung weiterer Beziehungen wären dann

$$\begin{aligned} mother(X,Y) &\rightarrow parent(X,Y) \\ father(X,Y) &\rightarrow parent(X,Y) \end{aligned}$$

Es lassen sich nun Aussagen machen wie *„Ein Bruder ist eine männliche Person, die ein gleiches Elternteil hat"*, was als $male(X)$, $parent(P,X$, $parent(P,Y)$, $notSame(X,Y) \rightarrow brother(X,Y)$ dargestellt werden kann. So lassen sich allerlei Regeln konstruieren. Reicht eine Regel nicht aus, können auch mehrere Regeln einen Sachverhalt ausdrücken.

### 9.1.6 Non-monotonic rule systems

Soll neues Wissen die Ergebnisse beeinflussen können, kommt man zu nichtmonotonen Regelsystemen. Ist zum Beipiel das Geburtsdatum eines Kunden bekannt, kann man ihm Rabatt gewähren (monoton), ist hingegen das Datum *noch* nicht bekannt, will man Regeln definieren, die den Rabatt bei Bekanntmachung des Geburtsdatums gewähren definieren.
Hier ist es nützlich, anfechtbare Regeln zu haben. Hierzu kann eine Regel durch eine andere überstimmt werden. Um Konflikte zwischen Regeln zu erlauben, wird die Negation im Kopf und Rumpf der Regeln erlaubt.

$$\begin{aligned} p(X) &\Rightarrow q(X) \\ R(X) &\Rightarrow \neg q(X) \end{aligned}$$

Um nicht-monotone Regeln von monotonen Regeln zu unterscheiden, wird $\Rightarrow$ benutzt anstelle von $\rightarrow$. Nun scheint es noch sinnvoll ein Prioritätensystem einzuführen, mit externen Prioritäten wie

$$\begin{aligned} r_1 : p(X) &\Rightarrow q(X) \\ r_2 : R(X) &\Rightarrow \neg q(X) \\ \textit{und} \quad r_1 &> r_2. \end{aligned}$$

Die externen Prioritäten müssen azyklisch sein.

**Definition 7** *(Defeasible Rules –anfechtbare Regeln)*
*Eine anfechtbare Regel ist von der Form*

$$r : L_1, ..., L_n \Rightarrow L$$

*mit $r$ als Label, $L_1, ..., L_n$ als Rumpf bzw. Prämissen und $L$ als Kopf der Regel. Ausserdem sind $L, L_1, ..., L_n$ positive oder negative Literale (also atomare Formale oder deren Negation).*

**Definition 8** *(Defeasible Logic Programm)*
*Ein Tripel von der Form $(F, R, >)$ besthend aus einer Menge $F$ von Fakten, einer endlichen Menge $R$ von anfechtbaren Regeln und der azyklischen binären Relation $>$ auf $R$.*

### 9.1.7 Beispiel eines nicht monotonen Regelsystems

siehe 09_vorlesung_rule_21_06_2004.pdf F.19

# 10 Rule Markup and the uppper layers of the Semantic Web

Um die im Vorangegangenen besprochenen Regeln nun auch in XML definieren zu können ist ein ,,Markup" nötig.

## 10.1 Rule Markup

Terme werden durch die Tags `<term>`, `<function>`, `<var>` und `<const>` realisiert. So kann man zum Beispiel folgenden Term $f(X, a, g(b, Y))$ darstellen durch

```
<term>
    <function>f</function>
    <term>
        <var>X</var>
    </term>
    <term>
        <const>a</const>
    </term>
    <term>
        <function>g</function>
    </term>
    ...
</term>
```

Die zusätzlichen Tags `<atom>` und `predicate` erlauben dann Atomare Formeln. $f(X, a, g(b, Y))$:

```
<atom>
    <predicate>f</predicate>
    <term>
        <var>X</var>
    </term>
    ...
</atom>
```

Einen Fakt stellt man nun durch das Tag `<fact>` dar:

```
<fact>
    <atom>
        <predicate>p</predicate>
        <term>
            <const>a</const>
        </term>
    </atom>
</fact>
```

$\Rightarrow p(a)$.

Regeln bestehen ja aus einem Kopf und einem Rumpf, was durch die Tags `<head>` und `<body>` eingefasst durch `<rule>` ausgedrückt wird. Schliesslich werden Anfragen duch das `<query>`-Tag ausgedrückt, mit dem Inhalt eines Rumpfes einer Regel.

Monotone Regeln sind in einer DTD beschrieben:
Ein Programm besteht aus Regeln und Fakten:

```
<!ELEMENT program ( (rule | fact)*)>
```
Ein Fakt besteht aus atomaren Formeln:
```
<!ELEMENT fact (atom)>
```
Eine Regel besteht aus einem Kopf und einem Rumpf:
```
<!ELEMENT rule (head | body)>
```
Ein Kopf besteht aus atomaren Formeln:
```
<!ELEMENT head (atom)>
```
Ein Rumpf besteht aus einer Liste von atomaren Formeln:
```
<!ELEMENT body (atom*)>
```
Eine atomare Formel besteht aus einem Prädikat gefolgt von einer Reihe von Termen:
```
<!ELEMENT atom (predicat, term*)>
```
Eine Term ist eine Konstante, eine Variable oder ein Kompositum aus einem Funktionensymbol gefolgt von einer Reihe von Termen:
```
<!ELEMENT term (const | var | (function, term*))>
```
Prädikate, Funktionensymbole, Konstanten und Variablen sind atomare Formeln:
```
<!ELEMENT predicate (#PCDATA)>
<!ELEMENT function (#PCDATA)>
<!ELEMENT const (#PCDATA)>
<!ELEMENT var (#PCDATA)>
```
Eine Anfrage ist eine Liste von atomaren Formeln:
```
<!ELEMENT query (atom*)>
```

Für nicht monotone Regeln gelten folgende syntaktische Unterschiede

- negierte atomare Formeln dürfen in Kopf und Rumpf von Regeln auftreten
- Jede Regel hat ein Label
- Neben Regeln und Fakten enthält ein Logikprogramm noch Aussagen zur Priorität

...Folien 11-16

## 10.2 Trust

## 10.3 Proof

## 10.4 Web Services

## 10.5 Übung 6

# 11 User Modeling, Personalization of Hypermedia: Adaptive Hypermedia

Die Wurzeln der Personalisierung des /semantischen) Webs sind Hypermedien (sog. Adaptive Hypermedien) und Die Analyse von Nutzungsdaten (sog. ,,Recommender Systems").
Adaptive Hypermedia stammt aus Hypermedia Systemen, sowie intelligenten Tutorsystemen und findet seinen Anfang in den frühen 90er Jahren (1992/93). Ein Meilenstein war 1996 *,,Methods and technics of adaptive hypermedia" – P. Brusilovsky.* ,,Recommender Systems" stammen aus dem Bereich des Datamining.

Zunächst folgen die Definitionen zu *,,Hypertext"* und *,,Hypermedia"*:

**Definition 9** *(Hypertext)*
*Ein Hypertext ist eine Menge von Knoten, die aus Text bestehen, die durch Links miteinander verbunden sind. Jeder Knoten enthält dabei eine gewisse Menge an Informationen (Text) und eine Anzahl von Links zu andern Knoten.*

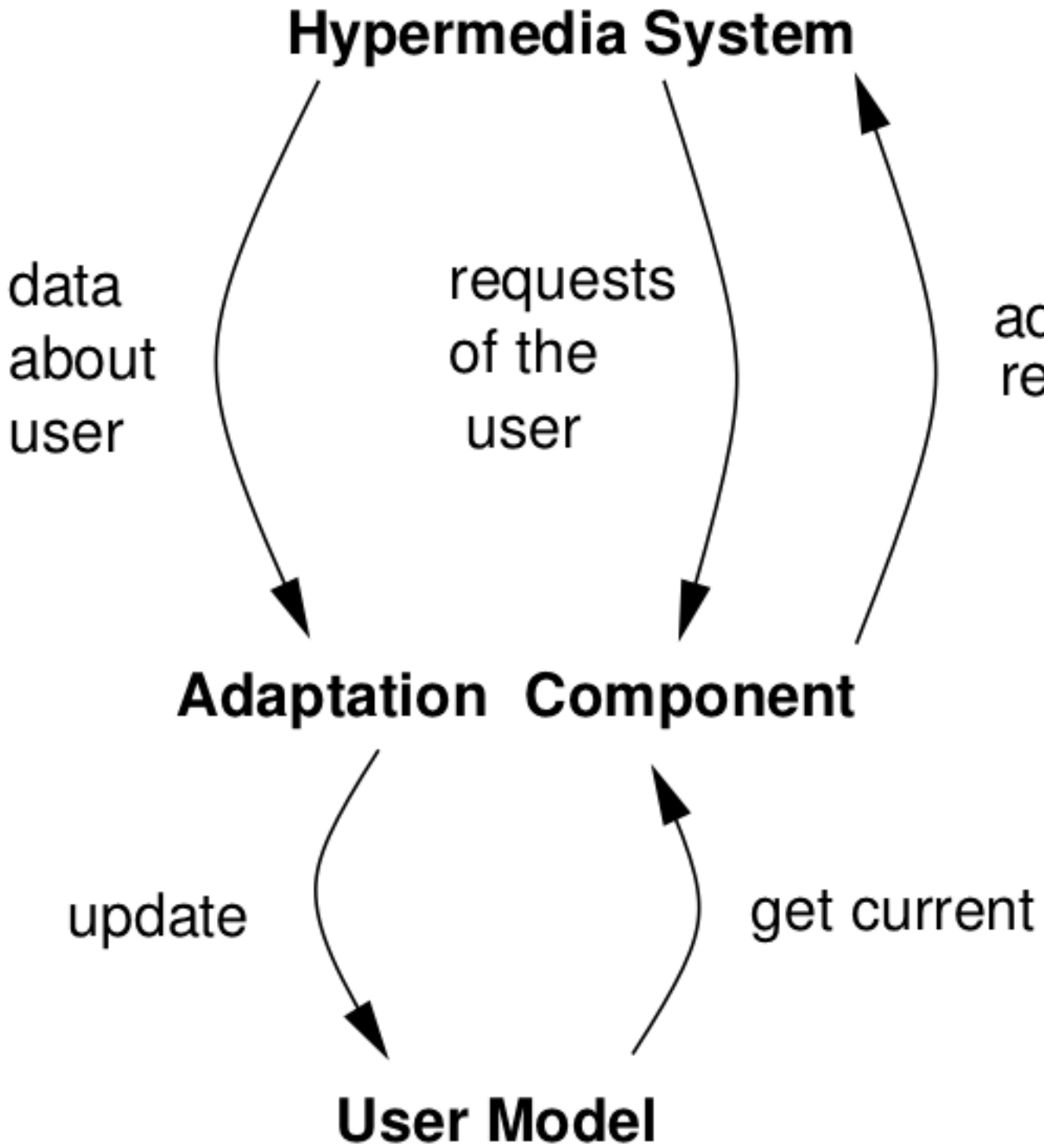
**Definition 10** *(Hypermedia)*
*Ist eine Erweiterung von Hypertext, die sich verschiedener Formen von Medien (Text, Audio, Video, ...) bedient.*

Hypermedia wird zu Addaptiver Hypermedia, wenn es sich um personalisierte Systeme für individuelle Benutzer handelt. So erhält jeder Nutzer eine individuelle Sicht und eigene Navigationsmöglichkeiten, mit dem Hypermediensystem zu arbeiten. Ein *,,Adaptive Hypermadia System"* wird dann wie folgt definiert

**Definition 11** *(Adaptive Hypermedia System)*
*Mit einem adaptiven Hypermedialen System sind alle Hypertext und Hypermedialen Systeme gemeint, die einen gewissen Teil der Eigenschaften des Nutzers in einem Nutzermodell halten und diese in visuellen Aspekten des Systems wiederspiegeln.*

Anwendung finden solche System in vielen Bereichen

- in der Ausbildung, der Fokus leigt hier im Bereich Wissen / Lernen
- Online-Informationssystemen, zum Beispiel: elektronische Enzyklopädien, Dokumentensammlungen, Reise-, Online-Hilfs-, Organisationsinformationssysteme, Wissensmanagement, ...; Der Fokus liegt hier auf de mKontext (schnelle Referenzierung, Vorbereitung der Präsentation, Wissenswieder auffrischung, einen Überblick erlange, ...)
- Personalisierte Sichten (personalized views), hier gibt es statische (Abfrage einer Datenbank) und dynamische (individuelle Sichten)
- E-Commerce; optimiert die Produktpräsentation, adaptiert auf das Benutzerverhalten
- Führung und Orientation (lokal und global) bieten




## 11.1 User Modeling

Beim erstellen eines Usermodells für ein adaptives System kann man entweder einen individuellen Nutzer modellieren oder Stereotypen definieren. Der kritische Punkt bei der Modellierung von individuellen Nutzern ist die Initialisation. Beim Modellieren von Gruppen von Nutzern bzw von Stereotypen (Vorkenntnisse, Inter-

essen, Demographische Gruppen, ...) ist der kritische Punkt, die Balance zu halten zwischen einem zu generellen Modell, das keinen Effekt erziehlt und einem zu speziellen, was keinen Stereotyp liefert. Beispiele für individuelle Modelle sind e-Learning, task-/processsupport, knowledege management, recruitement. Zum zweiten Ansatz würde man imme dann greifen, wenn man Gruppensharakteristika erkennen kann, zum Beipiel bei einem Krankenhaus-Informationssystem (Doktoren, Krankenschwestern, Besucher, Kranke, ...).

In einem Nutzermodell können Charakteristika eines Nutzers abgebildet werden, wie zum Beipsiel

- Geräteinformationen (Anzeigeeigenschaften, Internetverbindung, Werkzeuge, ...)
- Vorlieben ( Farben, html/txt/pdf/...)
- Ziele, momentante Aufgaben
- Benötigte Informationen
- vorhandenes Wissen, Erfahrung, schon gesehene Informationen
- ...

**Definition 12** *(User Model – Nutzermodell)*
*Ein Nutzermodell ist eine Anwendung, die Charakteristika von Nutzer unterhält und aus diesen schlossfolgert.*

Ein Nutzermodell speichert also nicht nur Daten eines Nutzers, sondern zieht auch Schlüsse daraus und versucht neue Annahmen zu erstellen.

Kritisch hierbei ist immer die Privatssphäre des Nutzers zu beurteilen. Ausserdem ist zu beachten, dass sich die Daten mit der Zeit ändern, manche schneller und manche langsamer.

Im Hypertext kann man sich folgende Adaption vorstellen

- Hypertext → Graph
- Knoten → Adaption auf inhaltlicher Ebene
- Kanten → Adaption auf Navigationsebene

ansich ist ein Graph, die Knoten

Auf iunhaltlicher Ebene können weitere Erklärungen eingeblendet werden, Vergleiche oder Varianten angezeigt werden oder auch die Reihenfolge der Präsentation verändert werden. Dies kann über bedingten oder gestreckten Text, Seitenfragmente oder -varianten oder aber auch Frames geschehen.

Auf der Ebene der Navigation kann man den Nutzer direkt führen (nächste Schritte, Pfade,...), die Sortierung adaptieren (Gleiches gruppierne, auf Grund von Vorkenntnissen sortieren, ..) oder Information verstecken, die noch nicht relevent ist. Auch das Konzept der sogenannten *„Traffic Lights"*, auch in Kombination mit Informationsunterdrückung, bei dem man zum Beispiel sehen kann, was man schon gelesen hat und was nicht, ist sehr beliebt. Hierfür gibt es viele verschiedene Systeme und Techniken (siehe logikbasierte Definition von Adaptiver Hypermedialer Systeme)

## 11.2 Adaptive Hypermedia

## 11.3 Übung 7

# 12 Definition of Adaptive Hypermedia Systems, Examples of AHS, Recommender Systems

## 12.1 Adaptive Hypermedia

## 12.2 Recommender Systems

# 13 Summary and Outlook

## 13.1 User Modeling

## 13.2 Adaptive Hypermedia

# A Begriffe

**Syntax** Muster und Regeln, nach denen Wörter zu größeren funktionalen Einheiten zusammengefasst werden.

**Semantik** Meint die Bedeutung der Wörter (Wörter ohne Bedeutung sind nur Wörter, mit Bedeutung hingegen werden sie zu Worten)

**Prämisse** Voraussetzung (*engl.:* premise)

# B Abbildungen und Tabellen

## Abbildungsverzeichnis



## Tabellenverzeichnis