XML–Modelle und Programmierung Vorlesung, Wintersemester 2005

Johannes Waldmann, HTWK Leipzig

2. Februar 2006

Einleitung

Baumstrukturierte Dokumente

- Baumstruktur drückt Hierarchie aus (= Schichten von Abstraktionen)
- nur durch solche Strukturen kann man Überblick über Daten behalten
- gerichteter, geordneter Baum:

  jeder Knoten besitzt Liste von Kind-Knoten und evtl. weitere Attribute
- jedes XML-Dokument repräsentiert einen Baum

XML-Dokument (Beispiel)

(Roger L. Costello, XML Scheme Tutorial, GPL)

```
<?xml version="1.0"?>
<BookStore xmlns="http://www.books.org"
           xmlns:xsi="http://www.w3.org/2001/XM
           xsi:schemaLocation=
                          "http://www.books.org
                           BookStore.xsd">
        <Book>
                <Title>My Life and Times</Title
                <Author>Paul McCartney</Author>
                <Date>1998</Date>
                <ISBN>1-56592-235-2</ISBN>
                <Publisher>McMillin Publishing<
```

```
        </Book>
        <Book>
                <Title>Illusions The Adventures
                <Author>Richard Bach</Author>
                <Date>1977</Date>
                <ISBN>0-440-34319-4</ISBN>
                <Publisher>Dell Publishing Co.<
        </Book>
        <Book>
                <Title>The First and Last Freed
                <Author>J. Krishnamurti</Author
                <Date>1954</Date>
                <ISBN>0-06-064831-7</ISBN>
                <Publisher>Harper &amp; Row</Pu
```

```
        </Book>
</BookStore>
```

Form von XML-Dokumenten

jeder Knoten des Baumes heißt *Element* und besitzt:

- genau einen Namen
- kein, ein, mehrere *Kinder*, das sind Elemente.

textuelle Darstellung entspricht Baumdurchquerung:

- Teilbaum betreten: Element beginnt, begin-tag `<foo>`
- Teilbaum verlassen: Element endet, end-tag `</foo>`

Jedes XML-Dokument mit korrekter Klammerung der begin/end-tags heißt *wohlgeformt*.

Typen für Bäume

- ... wohlgeformt heißt noch nicht *sinnvoll*, deswegen:
- *XML-Schema* deklariert erwartete *Struktur* von Elementen (Name, Anzahl der Kind-Elemente)
- ein wohlgeformtes Dokument D, das zu einem Schema S paßt, heißt *gültig* (valid) bzg. S.

  (Das Feststellen der Gültigkeit heißt Validieren.)

Schema-Beispiel

(Roger L. Costello, XML Scheme Tutorial, GPL)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/X
            targetNamespace="http://www.books.o
            xmlns="http://www.books.org"
            elementFormDefault="qualified">
    <xsd:element name="BookStore">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Book" minOccu
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
```

```
    <xsd:element name="Book">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Title" minOcc
                <xsd:element ref="Author" minOc
                <xsd:element ref="Date" minOccu
                <xsd:element ref="ISBN" minOccu
                <xsd:element ref="Publisher" mi
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Title" type="xsd:string"
    <xsd:element name="Author" type="xsd:string
    <xsd:element name="Date" type="xsd:string"/
```

```
    <xsd:element name="ISBN" type="xsd:string"/
    <xsd:element name="Publisher" type="xsd:str
</xsd:schema>
```

Zusammenfassung

in Wirklichkeit passiert hier folgendes:

- ein XML-Dokument ist ein Baum
- ein XML-Schema definiert eine Menge von Bäumen

vergleiche:

- eine Grammatik definiert eine Sprache = Menge von Zeichenketten

ein XML-Schema ist eine reguläre Baumgrammatik, die eine *reguläre Baumsprache* definiert.
(alles seit spätestens 1980 gut bekannt, Baumsprachen, algebraische Datentypen)

XML-Benutzung

- wie werden Dokumente bzw. Teile davon spezifiziert? (Schema, XQuery)
- wie werden Dokumente verarbeitet?

  durch Werkzeug (XSLT) / durch Programm (mit XML-API)
- wie werden Dokumente im Programm repräsentiert?
  - gar nicht (SAX – stream processing)
  - als Baum (DOM)
  - als interne Datenstruktur (benötigt Data Binding)

Programmierbeispiel

benutzt JAXP-1.3,
Beispiel aus M. SeebÃ¶rger-Weichselbaum:
BrÃ¼ckenschlag, in XML-Magazin 2/05

```
DocumentBuilderFactory factory = DocumentBuilde
DocumentBuilder builder = factory.newDocumentBu
document = builder.parse( new File(param[0]) );
SchemaFactory schemafactory =
    SchemaFactory.newInstance(XMLConstants.W3C_
Source schemaFile = new StreamSource(new File(p
Schema schema = schemafactory.newSchema(schemaF
Validator validator = schema.newValidator();
validator.validate(new DOMSource(document));
```

Übungsaufgaben

Installation prüfen/nachholen: Eclipse-3.1.1, JDK-1.5 (enth. JAXP-1.3)

- Eclipse/Java (für bash – wer andere Shell nimmt, ist selbst dran schuld)

  `export PATH=/home/waldmann/built/bin:/home/wa`
- API-Dokumentation:
  `http://java.sun.com/j2se/1.5.0/docs/api/`
- Beispiel-Programmtexte:
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws05/xml/folien/eins/` (C) *.java von M. Seeberger-Weichselbaum aus XML-Magazin 2/05

`http://www.xmlmagazin.de/` (C) *.xml, *.xsd von Roger K. Costello, XML Scheme tutorial
`http://www.xfront.com/xml-schema.html`

Programmierbeispiel

- mit Beispielprogramm `parsen.java` das Dokument `BookStore.xml` gegen das Schema `BookStore.xsd` validieren.
- im Quelltext sowohl Dokument als auch Schema von URL (statt Datei) lesen
- eine Methode schreiben, die den DOM-Baum durchläuft und für jedes Element den Typ ausgibt (und dann alle Kinder besucht)

```
public static void walk (Node n, int depth) {
```

Ausgabe soll sinnvoll eingerückt sein (soviele Leerzeichen, wie das zweite Argument angibt)

- die Ausgabe interpretieren

Später:

- ein Schema für 2/3-Bäume schreiben (d. h. jeder Knoten ist leer oder hat einen Schlüssel und 2 oder 3 Kinder), ein Beispiel-Dokument und validieren.
- ein Programm, das zusätzlich prüft, ob alle leeren Knoten gleich tief sind (geht das mit einem Schema?)

Modelle

Bäume, Terme

- *Graphentheorie*: Baum ist kreisfreier zusammenhängender Graph

  (d. h. ungerichtet, ungeordnet)

- *Algebra*: ein *Term* ist ein Baum mit
  - markierten Knoten (Funktionssymbole)
  - gerichteten Kanten (von der Wurzel weg)
  - geordneten Kindern (nicht Menge, sondern Liste)

Wenn f ein Symbol und $k \geq 0$ und $t_1, \ldots, t_k$ Terme, dann ist auch $f(t_1, \ldots, t_k)$ ein Term.

Positionen in Termen

Position eines Knotens (Teilterm) beschreiben durch Pfad von der Wurzel:

$\mathrm{Pos}(t) \subseteq \mathbb{N}^*$ ist Menge der Positionen von Term t:
$\mathrm{Pos}(f(t_1, \ldots, t_k)) = \{\epsilon\} \cup \bigcup_{1 \leq i \leq k} \{i\} \cdot \mathrm{Pos}(t_i)$

Übung: gibt es einen Term t mit $\mathrm{Pos}(t) = \{\epsilon, 1, 21, 113\}$? Welche Eigenschaften erfüllt jede Menge $\mathrm{Pos}(t)$?

Teilterme

$t(p)$ ist Symbol an Position $p \in \mathrm{Pos}(t)$:
$(f(\ldots))(\epsilon) = f, (f(t_1, \ldots, t_k))(iw) = t_i(w)$

$t|_p$ ist Teilterm von t an Position $p \in \mathrm{Pos}(t)$:
$t|_\epsilon = t, (f(t_1, \ldots, t_k))(iw) = t_i|_w$

Baum-Automaten (I)

Literatur: Gecseg/Steinby: Tree Automata (1984), Comon et al.: TATA (Tree automata Techniques and Applications) (2000–),
`http://www.grappa.univ-lille3.fr/tata/`
Definition: Ein Baumautomat $A = (\Sigma, Q, F, R)$ besteht aus

- Alphabet A, • Zustandsmenge Q
- Menge der akzeptierenden Zustände $F \subseteq Q$
- Regelmenge R mit Regeln der Form $f(q_1, \ldots, q_k) \to q$ mit $f \in \Sigma, k \geq 0, q_1, \ldots, q_k \in Q, q \in Q$.

Beispiel: $\Sigma = \{F, T, \vee, \wedge\}, Q = \{0, 1\}, F = \{1\}$,
$R = \{F \to 0, T \to 1, 0 \vee 0 \to 0, 0 \vee 1 \to 1, 1 \vee 0 \to 1, 1 \vee 1 \to 1, 0 \wedge 0 \to 0, 0 \wedge 1 \to 0, 1 \wedge 0 \to 0, 1 \wedge 1 \to 1\}$.

Baum-Automaten (II)

Eine *Rechnung* von A mit Eingabeterm t ist eine Folge von Regelanwendungen.
Beispiel: Eingabe $t = F \vee (T \wedge F)$, Rechnung
$F \vee (T \wedge F) \to 0 \vee (T \wedge F) \to 0 \vee (1 \wedge F) \to 0 \vee (1 \wedge 0) \to 0 \vee 0 \to 0$.
Eine *akzeptierende* Rechnung endet mit einem $q \in F$. Wir schreiben $t \to_R^* q \in F$.
Die Sprache des Automaten ist
$L(A) = \{t \mid \exists q \in F : t \to_R^* q\}$.
Die Sprache des Beispiel-Automaten sind genau die Formeln mit Wahrheitswert 1.
Übung: warum gibt es in Baum-Automaten keine Startzustände?
Übung: geben Sie einen Baumautomaten für die Menge aller allgemeingültigen aussagenlogischen Formeln über dem Alphabet $\{x, T, F, \neg, \vee, \wedge\}$ mit einer Variablen x an (Beispiel: $x \vee \neg x$ gehört dazu.) Hinweis: der Automat hat 4 Zustände.

Beispiele für reguläre Baumsprachen

- Alle Bäume über $\{r^2, b^2, a^0\}$ (d. h. r und b zweistellig, a nullstellig), bei denen kein r-Knoten ein r-Kind besitzt (vgl. Rot-Schwarz-Bäume)
- Alle Bäume über $\{f^2, g^2, a^0, b^0\}$, die wenigstens ein f und darunter wenigstens ein g enthalten

Anwendungen von Baum-Automaten

- Automaten beschreiben Muster für (erwünschte, verbotene) Bäume und Teilbäume

  (entspr. reguläre Ausdrücke für Wörter)

- Diese Muster (Automaten) könne auf eine effiziente (deterministische) Form gebracht werden

  (entspr. sog. *Kompilieren* von regulären Ausdrücken)

- Damit können Anfragen an XML-Dokumente effizient bearbeitet werden

Top-down oder Bottom-up

- Bottom-up: Eingabe verarbeiten (akzeptieren): $F \to 0, 0 \vee 1 \to 1, \ldots,$
- Top-down: Ausgabe erzeugen: $1 \to 0 \vee 1, 0 \to F$
- beide Sichtweisen sind äquivalent (Pfeile herumdrehen)
- … für nichtdeterministische Automaten
- was heißt *Determinismus*?

Ein Bottom-Up-Automat (Σ, Q, F, R) heißt deterministisch, wenn in R alle linken Regelseiten verschieden sind.
Ein Top-Down-Automat (Σ, Q, F, R) heißt deterministisch, wenn?

Es gibt Baumsprachen, die einen deterministischen bottom-up-, aber keine deterministischen top-down-Automaten besitzen.

Operationen mit Baumautomaten

Wenn A_1, A_2 endliche Baumautomaten über Signatur Σ, dann kann man jeweils einen Baumautomaten konstruieren für

- Vereinigung $L(A_1) \cup L(A_2)$
- Durchschnitt $L(A_1) \cap L(A_2)$
- Komplement $\mathrm{Term}(\Sigma) \setminus L(A_1)$

Kreuzproduktkonstruktion

- gegeben $A_1 = (\Sigma, Q_1, F_1, R_1), A_2 = (\Sigma, Q_2, F_2, R_2)$,
- konstruiere $A = (\Sigma, Q_1 \times Q_2, F, R)$
- mit passendem F und R
- ermöglicht Konstrukttion von Vereinigung und Durchschnitt von regulären Baumsprachen

Potenzmengenkonstruktion

- gegeben $A_1 = (\Sigma, Q_1, F_1, R_1)$
- konstruiere $A = (\Sigma, 2^Q, F, R)$
- mit passendem F, R
- so daß A deterministisch und vollständig ist und $L(A) = L(A_1)$.
- ermöglicht Konstruktion von Komplementen

Einzelheiten zu XML

Beispiele aus: Michael Fitzgerald: *XML Hacks*, O'Reilly 2004, ISBN: 0-596-00711-6
`http://www.oreilly.com/catalog/xmlhks/`,
`http://examples.oreilly.com/xmlhks/`

Dokumente

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- a time instant -->
<time timezone="PST">The time is:
 <hour>11</hour> <minute>59</minute>
 <second>59</second> <meridiem>p.m.</meridiem>
 <atomic signal="true" symbol="&#x25D1;"/>
</time>
```

enthält:

- XML-Deklaration (1. Zeile)
- Kommentar (2. Zeile)
- Element `time`

Element

```
<time timezone="PST">The time is:
 <hour>11</hour> <minute>59</minute>
 <second>59</second> <meridiem>p.m.</meridiem>
 <atomic signal="true" symbol="&#x25D1;"/>
</time>
```

Element besitzt Namen und kann enthalten:

- Attribute (`timezone="PST"`)
- Kind-Elemente (`second`,…)
- Text (parsed character data)

wenn Element sowohl Text als auch Kinder besitzt, hat es *mixed content*

Sonderzeichen

einige Zeichen haben syntaktische Bedeutung, diese kann ausgeschaltet werden durch

- *character references*

  `ü` (hex), `ü` (dez)

- *entity references*

  predefined: `< > & ' "`

- *CDATA*

  `<![CDATA[ foo & bar ]]>`

Namensräume

Zuordnung von Element-Namen zu Namensräumen

default-Deklaration (gilt für Element und alle Kinder)

```
<time timezone="PST"
      xmlns="http://www.wyeast.net/time">
 <hour>11</hour> <minute>59</minute> ...
</time>
```

prefix-Deklaration:

```
<tz:time timezone="PST"
    xmlns:tz="http://www.wyeast.net/time">
 <tz:hour>11</tz:hour> <tz:minute>59</tz:minute
</tz:time>
```

Namespace selbst ist nur URI

XML-Schema

Schema

siehe `http://www.w3.org/TR/xmlschema-0/`

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="time" type="Time"/>
<xs:complexType name="Time">
  <xs:sequence>
   <xs:element ref="hour"/>
   <xs:element ref="minute"/>
  </xs:sequence>
   <xs:attribute name="timezone" type="xs:string" use="r
</xs:complexType>
<xs:element name="hour" type="Digits"/>
<xs:simpleType name="Digits">
 <xs:restriction base="xs:string">
```

```
  <xs:pattern value="\d\d"/>
 </xs:restriction>
</xs:simpleType>
```

Schema, Definition

- Schema ist Baumgrammatiken (Top-Down-Automaten), Grammatik-Variablen = Automaten-Zustände = `xs:complexType` (kann Kindelemente, Attribute haben) oder `xs:simpleType` (kann nicht)
- Funktionssymbole = `xs:element` oder `xs:attribute`
- Grammatik ist selbst ein XML-Dokument, benutzt Namensraum:

  `<xs:schema xmlns:xs="http://www.w3.org/2001/X`
- enthält Deklarationen für zusammengesetzte Typen, einfache Typen, Elemente

Zusammengesetzte Typen (I)

- mit einfachem Inhalt, mit Attributen:

```
<xs:complexType>
  <xs:simpleContent>
    <xs:extension base="xs:decimal">
      <xs:attribute name="currency" type="xs:s
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

- ...

Zusammengesetzte Typen (II)

- Reihung (Bestandteile müssen alle in genau der Reihenfolge vorkommen)

```
<xs:sequence>
 <xs:element ref="hour"/>
 <xs:element ref="minute"/>
</xs:sequence>
```

- Auswahl (genau einer der Bestandteile muß vorkommen)

```
<xsd:choice>
  <xsd:group   ref="shipAndBill"/>
  <xsd:element name="singleUSAddress" type="US
</xsd:choice>
```

Einfache Typen

- vordefinierte: `string,integer,boolean, date,duration,...` siehe `http://www.w3.org/TR/xmlschema-0/#simpleTypesTable`
- selbst definierte, durch:
  - Einschränkung `http://www.w3.org/TR/xmlschema-0/#facetsTable1`

    ```
    <xs:simpleType name="myInteger">
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="10000"/>
        <xs:maxInclusive value="99999"/>
      </xs:restriction>
    </xs:simpleType>
    ```
  - Reihung `<xs:list itemType="myInteger"/>`
  - Auswahl `<xs:union memberTypes="foo bar"/>`

Anonyme oder benannte Typen

Anonym:

```
<xs:element name="time">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="hour" type="xs:string"/>
            <xs:element name="minute" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

benannt:

```
<xs:element name="time" type="Time"/>
<xs:complexType name="Time">
  <xs:sequence>
   <xs:element ref="hour"/> ...
<xs:element name="hour" type="Digits"/>
```

Typen, Einschätzung

- die Idee, überhaupt Typen (Grammatiken) zu verwenden, ist sehr richtig
- für zusammengesetzte Typen:

  Lage wird verkompliziert durch implizite Reihungen (Listen) ohne direkt umschließendes Element

  (minOccurs, maxOccurs, Gruppen)

- Unterscheidung zwischen Element und Attribut ist zweifelhaft.

  oft sagt man: Elemente sind Daten, Attribute sind Meta-Daten. aber warum sollen Metadaten nicht strukturiert sein?
- …führt zu übelsten String-Hackereien in Attributen:
  - reguläre Ausdrücke (simulieren Struktur/Grammatik),
  - union (simuliert choice),
  - list (simuliert sequence)

Übungen zu XML-Schema

- 2/3-Bäume (jeder innere Knoten 2 oder 3 Kinder)
- Rot-Schwarz-Bäume (jeder Knoten rot oder schwarz, keine zwei roten direkt untereinander)

Namespaces und Schemas

- Verweis von Dokument auf Schema (für unqualif. Namen):

```
<foobar
   xmlns:xsi="http://www.w3.org/2001/XMLSchema
   xsi:noNameSpaceSchemaLocation="my-schema.xs
> ... </foobar>
```

- Verweise auf Schema mit Namensraum:

```
<foobar
   xmlns:xsi="http://www.w3.org/2001/XMLSchema
   xsi:SchemaLocation="http://my-namespace.org
   xmlns="http://my-namespace.org/"
```

```
> ... </foobar>
```

- Namensraum im Schema angeben:

```
<xs:schema
    targetNamespace="http://my-namespace.org/"
    xmlns:adr="http://my-namespace.org/"
> ... </xs:schema>
```

Import von Schemas

- Ein Schema kann in ein anderes eingefügt werden:

```
<xs:include schemaLocation="anderes.xsd"/>
```

- dabei kann man auch noch umdefinieren:

```
<xs:redefine schemaLocation="anderes.xsd">
  <xs:complexType name="foo"> ... </ >
</xs:redefine>
```

  …das ist natürlich ganz furchtbar.

- (grundsätzlich ist Präfix `xs` so definiert:)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XM
```

Ableitung: Erweiterung und Einschränkung

- Typen um neue Elemente und Attribute erweitern:

```
<xs:complexType name="bar">
  <xs:extension base="foo">
    <xs:sequence>...
```

- Wertebereiche einschränken (dabei Deklarationen wiederholen)

abgeleitete Typen dürfen in Instanzdokumenten anstelle der ursprünglichen benutzt werden.
es gibt Begriffe *abstract, final* …
das ist alles ganz nett, aber es fehlt (natürlich wieder mal) generischen Polymorphie (Bsp: RS−Baum<E>)

Data mapping (Diskussion)

grundsätzliche Aufgabe:
möglichst enge Beziehung herstellen
zwischen Baumstruktur des Datenmodells der Anwendung
und der Baumstruktur XML-Repräsentation …

in den Fällen, wo das geht. — Wann geht es *nicht*? wenn Anwendungsdaten keine Bäume sind, sondern andere Graphen (Relationen).

Data mapping für algebraische Datentypen

Mathematik (= Haskell):

```
data Tree a
  = Leaf
  | Node { key :: a, children :: [ Tree a ] }
```

Java (naiv)

```
class Node<A> {
  A key ; Node<A> [] children ;
}
```

Java ("richtig")

```
interface Tree<A> { void insert (A x); ... }
class Leaf<A> implements Tree<A> { }
class Node<A> implements Tree<A> { }
```

Data mapping (II)

Es gibt diese Begriffe/Beziehungen:

Baum, Knoten, Schlüssel, Blatt, (linkes/rechtes) Kind.

genauer z. B.:

ein *Knoten* ist ein nichtleerer *Baum*, der zwei Bäume *besitzt*, die er *Kinder* nennt.

Das muß man auf XML-Struktur abbilden
(Diskussion Übungsaufgabe)
Es gibt (m. E., wie immer) eine ausführliche Lösung und verkürzende Hacks.
Verkürzungen sind aber gar nicht nötig, da das niemand von Hand machen sollte.

Beispiel: Haskell2Xml

```
data Tree a = Leaf
            | Node { key :: a
                   , colour :: Colour
                   , left :: Tree a
                   , right :: Tree a
                   }
{-! for Tree derive: Haskell2Xml !-}
```

Präprozessor `DrIFT` erzeugt daraus Quelltext (Instanz für Interface)

```
class Haskell2Xml a where
    toXml   :: a -> Document
    fromXml :: Document -> a
```

Haskell2Xml (II)

```
Node { key = 4, colour = Black
   , left = Node { key = 1, colour = Black
                 , left = Leaf, right = Leaf }
   , right = Leaf }
```

Objekt (oben), Repräsentation (unten)

```
<Tree-int-XML>
 <Node-int>
   <int value="4" /> <Black />
   <Node-int >
     <int value="1" /> <Black /><Leaf /><Leaf/>
   </Node-int>
   <Leaf/>
  </Node-int>
</Tree-int-XML>
```

Data Binding/Diskussion

grundsätzliches Problem:

```
<foo>
   <bar> ... </bar>
</foo>
```

kann zweierlei bedeuten:

- bar ist Attribut-Name des Objektes foo
- bar ist Attribut-Konstruktor …

hier *Attribut* im OO-Sinne (nicht XML-Sinne) des Wortes
Haskell2Xml: ignoriert Attribut-Namen. — Übung: Castor?

JAXB: von Schema zu Java

Grundlegendes

- Java Web Services

  enthalten u. a. JAXB (Java Architecture for Xml Binding)

- Tutorial: `http://java.sun.com/webservices/docs/1.6/tutorial/doc/index.html`

Konvertierung in beide Richtungen möglich

- XML-Schema → Java-Klassen:

  mit Compiler `xjc`

- Java-Klassen → XML-Schema:

  mit Annotationen/Reflection

Von Schema zu Java

- Binary in `$JWSDP/jaxb/bin/xjc`, einfachste Benutzung (ausprobieren):

  ```
  xjc BookStore.xsd
  ```

- für eigenes Schema probieren (binäre Bäume)
- erzeugte Klassen/Methoden benutzen (unmarshal/marshal)

Castor

Castor is an Open Source data binding framework for Java[tm]. It's the shortest path between Java objects, XML documents and relational tables. Castor provides Java-to-XML binding, Java-to-SQL persistence, and more.
`http://castor.codehaus.org/`
Idee: Verbindung zwischen XML-Schema und Java-Klassen (für beide Richtungen) durch *Java-Introspektion* bestimmen (evtl. mithilfe einer Mapping-Datei beschreiben)

Castor-Beispiel

(binäre Bäume, Varianten ausprobieren)

Introspektion

- ... bezeichnet Ermitteln und Ausnutzen von Typinformationen zur Laufzeit (z. B. Klassen-Signaturen)
- in Java kann diese Information aus class-files bzw. class-Objekten gewonnen werden (Quelltext ist nicht notwendig)
- steht in Widerspruch zu statischer (Compile-Zeit) Typprüfung? deren Ziel ist ja, daß man gar keine Laufzeit-Typinformation braucht.

Introspektion (Beispiel)

```
import java.lang.reflect.*;

Class c = baum.Baum.class;

for (Field f : c.getFields()) {
System.out.println (f);
}

for (Method m : c.getMethods()) {
System.out.println (m);
}
```

Zum Vergleich: Introspektion in Haskell

```
data Rose a =
     Rose { key :: a
          , children :: [ Rose a ]
          }
     deriving ( Typeable, Data )
*Main> dataTypeOf $ make 2
DataType {tycon = "Main.Rose", datarep = AlgRep [Rose]}
*Main> dataTypeConstrs $ dataTypeOf $ make 2
[Rose]
*Main> map constrFields $ dataTypeConstrs $ dataTypeOf $
[["key","children"]]
```

Zum Vergleich: Präprozessoren

- Introspektion: Typinformation von Compilezeit zu Laufzeit retten
- Präprozessoren (CPP, Template Haskell): bereits zur Compilezeit echten Code ausführen (der dann Quelltext produziert, der kompiliert wird).

Zusammenfassung Data Binding

Datenformate:

- XML-Dokument (Text,Datei), XML-Schema
- DOM-Objekt, generische Klasse Node usw.
- XML-RPC-Daten
- Java-Objekt, anwendungsspezifische Klassen

Arbeitsgänge: Marshalling, Unmarshalling
Werkzeuge

- Compilezeit: Schema → Quelltexte mit xjc
- Laufzeit: Schema/Instanz ↔ Objekt mit Introspektion (Castor, JAXB)

XML-RPC

Idee

- RPC: remote procedure call,

  d. h. Client (caller) und Server (callee)

  – auf verschiedenen Rechnern
  – in verschiedenen Sprachen

- Datentransport (Argumente/Resultat) in sprachunabhängiger Form (als XML-Dokument)

  → benötigt Data Binding

RPC-Server/Client

- Server ist z. B. CGI-Programm, vom Webserver (bei jeder Anfrage neu) gestartet
- oder alleinstehender minimaler Web-Server (`org.apache.xmlrpc.WebServer`)

Client (Beispiel):

```
import org.apache.xmlrpc.*;
XmlRpcClientLite c =
  new XmlRpcClientLite
    ("http://dfa.imn.htwk-leipzig.de/cgi-bin/si
Object s = c.execute("examples.add",
  new Vector<Integer>
    (Arrays.asList (new Integer[] { 3, 4 })));
```

HTTP-Protokoll

Überprüfen mit `netcat`:

- Welche Anfrage schickt ein Web-Browser an einen Web-Server?

```
netcat -l -p 9876
konqueror http://localhost:9876/foo/bar.html
```

- Welche Antwort schickt der Web-Server?

  Browser-Anfrage (obere 3 Zeilen: GET, Host, Leerzeile) hier eingeben:

```
netcat www.imn.htwk-leipzig.de 80
```

Einfacher RPC-Server

Eine Klasse mit zu exportierenden Methoden:

```
class Numbers {
    public int add (int x, int y)
        { return x+y; }
}
```

Eine Server-Klasse mit main:

```
WebServer w = new WebServer(9876);
w.addHandler("Numbers", new Numbers());
w.start ();
```

Der von außen sichtbare RPC-Methodenname ist dann `Numbers.add`.

RPC-Aufgaben

- HTTP-Protokoll ausprobieren (netcat)
- RPC-Client ausprobieren (Server: `dfa.imn.htwk-leipzig.de/cgi-bin/simple_server`)
- RPC-Server ausprobieren
- komplexe Typen ausprobieren (Client) (RPC-Befehl listPeople auf Server `dfa.imn.htwk-leipzig.de/cgi-bin/person_server`)
- komplexe Typen (Server + Client):

  Methode `Numbers.divMod`, soll Record mit Quotient und Rest liefern

WSDL - Web Service Descrition (Language)

WSDL

Eigenschaften (Einschränkungen) von XML-RPC:

- Procedure Call (Frage/Antwort)
- vorgegebenes, eingeschränktes Data Binding (primitive Typen, struct = HashTable, array = Vector)

wünschenswerte Erweiterungen:

- andere Ablaufmodelle (z. B. Frage jetzt, Anwort später)
- anderes Data Binding (z. B. nach eigenem XML-Schema)
- Beschreibung von Services (= durch WSDL-Dokument)

Literatur

- Sameer Tyagi: Patterns and Strategies for Building Document-Based Web Services, `http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns/index.html`, `http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns2/index.html`
- WSDL Tutorial, `http://www.w3schools.com/wsdl/wsdl_intro.asp`
- Dennis Sosnoski: Apache Axis SOAP for Java `http://www.sosnoski.com/presents/java-xml/axis/`

XML-Transformationen / Term-Ersetzungs-Systeme

Idee

beschreibe *Transformationen* von Bäumen (Termen, Dokumenten) durch *Regeln*
Regel ist Paar von Termen mit Variablen: $(f(x), g(x, x))$,
Regelanwendung: wenn linke Seite *paßt*, dann *ersetze* durch entsprechende rechte Seite.
Beispiel $h(1, f(f(2))) \to_R h(1, g(f(2), f(2)))$.
Regelsystem in XSLT hinschreiben, mit Prozessor (z. B. Saxon, Xalan) auf XML-Dokument anwenden.
einfachster Fall: XML nach HTML (formatierte Ausgabe)

Variablen, Substitutionen

- Terme und Variablen: $\mathrm{Term}(\Sigma, V)$: Menge der Terme mit Symbolen aus Σ und Variablen aus V.
- Vereinbarung: Variablen am Ende des Alphabets ($x, y, \ldots$), Funktionssymbole am Anfang ($f, g, a, b, \ldots$)
- Menge der in einem Term t vorkommende Variablen: $\mathrm{Var}(t)$
- Substitution σ ist Abbildung $V \to \mathrm{Term}(\Sigma, W)$
- eine Substitution σ angewendet auf einen Term t erzeugt Term $t\sigma$

  Beispiel $t = f(x)$, $\mathrm{Var}(t) = \{x\}$, $\sigma : x \mapsto g(2,2)$, $t\sigma = f(g(2,2))$.

Positionen, Teilterme (Wiederholung)

- Term t, Position $p \in \mathrm{Pos}(t)$.
- $t(p)$ Symbol an Position p in t
- $t\mid_p$ Teilterm an Position p in t
- $t[p := s]$ in t an Position p den Term s einsetzen

Beispiele:

- Term $t = h(1, f(f(2)))$,
- Positionen $\mathrm{Pos}(t) = \{\epsilon, 1, 2, 21, 211\}$,
- Symbol $t(2) = f$,
- Teilterm $t|_2 = f(f(2))$,
- Teilterm ersetzen $t[2 := a] = h(1, a)$.

Regeln

- Regel (l, r), Schreibweise $(l \to r)$, mit $l, r \in \mathrm{Term}(\Sigma, V)$
- Regel $(l \to r)$ an Position p in $t \in \mathrm{Term}(\Sigma)$ anwenden, um s zu erhalten, Schreibweise $t \to_{(l\to r),p} s$

  $\exists \sigma : V \to \mathrm{Term}(\Sigma) : t|_p = l\sigma \wedge s = t[p := r\sigma]$

Beispiel: Regel $(l, r) = (f(x), g(x, x))$, Term $t = h(1, f(f(2)))$.
Position $p = [2] \in \mathrm{Pos}(t)$, Teilterm $t|_p = f(f(2))$,
Substitution $\sigma : x \mapsto f(2)$ mit $t|_p = l\sigma$,
auf r anwenden: $r\sigma = g(f(2), f(2))$,
in t einsetzen:
$s = t[p := r\sigma] = h(1, r\sigma) = h(1, g(f(2), f(2)))$.

Regelsysteme

- R eine Menge von Regeln, definiert Relation (einmalige Anwendung irgendeiner Regel irgendwo):

  $\to_R := \{(t, s) \mid \exists (l, r) \in R, p \in \mathrm{Pos}(t) : t \to_{(l,r),p} s\}$
- transitive Hülle $\to_R^+$ (mehrmalige Regelanwendung)

liefert nichtdeterministisches Berechnungsmodell (vgl. Grammatiken/Wortersetzungssysteme)
Fragen:

- jede Rechnung endet? (Termination) nach welcher Zeit?
- jede Rechnung endet mit genau einem Resultat?

dabei: Resultat = Term, auf den keine Regel anwendbar ist = Normalform.

Beispiele

Signatur (mit Stelligkeiten): $\Sigma = \{e^0, s^1, p^2, m^2, h^2\}$

$$R = \{p(e, y) \to y, p(s(x), y) \to s(p(x, y))\}$$

Ableitungen von $p(p(e, s(e)), s(e))$?

Wort- und Term-Ersetzung

Man kann aus jedem Wort (= Folge von Buchstaben) einen Term konstruieren,
Beispiel: $abaab \approx a(b(a(a(b(\epsilon)))))$.

Damit ist sind Wortersetzungssysteme ein Spezialfall von Termersetzungssystemen.
Beispiel:
$aabb \to bbbaaa \approx a(a(b(b(x)))) \to b(b(b(a(a(a(x))))))$

Alle Aussagen über TES gelten auch für WES.

einige Aussagen über WES sind spezieller, da es für Wörter eine assoziative Verknüpfung gibt, aber für Terme nicht.

Termination und Normalisierung

Für eine zweistellige Relation ρ auf M
(zum Beispiel eine Ersetzungsrelation $\to_R$ auf $\mathrm{Term}(\Sigma)$)

- ρ ist *terminierend (stark normalisierend)*, falls es keine unendliche lange ρ-Folge gibt

  d. h. $\rho(x_0, x_1), \rho(x_1, x_2), \rho(x_2, x_3), \ldots$
- ρ is schwach normalisierend, falls zu jedem $x_0 \in M$ eine ρ-Folge zu einer ρ-Normalform führt

Beachte: es gibt ρ, die schwach, aber nicht stark normalisieren.
Beispiel (Wortersetzung): $fgfg \to gfgffg$ (Alfons Geser, 2000)

Aufgaben

Signatur (mit Stelligkeiten): $\Sigma = \{e^0, s^1, p^2, m^2, h^2\}$

- $R_p = \{p(e, y) \to y, p(s(x), y) \to s(p(x, y))\}$
- $R_m = R_p \cup \{m(e, y) \to e, m(s(x), y) \to p(m(x, y), y)\}$
- $R_h = R_m \cup \{h(x, e) \to s(e), h(x, s(y)) \to m(h(x, y), x)\}$

Fragen (leicht):

- Menge der R_h-Normalformen?
- Normalformen (Existenz? Eindeutigkeit?) von: $p(s(p(s(e), s(e))), s(e))$, $m(s(s(e)), s(s(e)))$, $h(s(s(e)), s(s(s(e))))$.

(schwerer) $\Sigma = \{a^2, d^0\}, R = \{a(a(d, x), y) \to a(x, a(x, y))\}$: Normalformen? Ableitungslängen?

Normalformen

Für eine zweistellige Relation ρ auf M
(zum Beispiel eine Ersetzungsrelation $\to_R$ auf $\mathrm{Term}(\Sigma)$)
heißt $x \in M$ eine ρ-Normalform, falls $\neg\exists y \in M : \rho(x, y)$.

Beachte: für passende ρ und x kann vorkommen:

- p hat mehrere Normalformen
- p hat keine Normalform

Termersetzung/Anwendungen

Termersetzung ist

- Turing-vollständiges Berechnungsmodell
- Grundlage für funktionale Programmierung
- Grundlage für XML-Transformationen mit XSLT

Für Anwendungen wichtig sind

- Termination (keine unendlich langen Rechnungen)
- Konfluenz (eindeutige Ergebnisse von Rechnungen)

Term-Ersetzung und Computeralgebra

Regeln (Term-Ersetzungs-System) für das Differenzieren:

$$D_x(x) = 1, \text{ wenn } x \notin A : D_x(A) = 0$$
$$D_x(A + B) = D_x(A) + D_x(B)$$
$$D_x(A \cdot B) = \ldots, D_x(A/B) = \ldots$$
$$D_x(\log x) = 1/x, D_x(\sin x) = \cos x, \ldots$$

Dazu braucht man aber noch Vereinfachungsregeln.
Wie drückt man die Kettenregel $D_x(f(g(x)) = \ldots$ aus?
Hier sind f und g Variablen, aber zweiter Ordnung
(bezeichnen Funktionen).
Regeln für das Integrieren?
Existenz und Eindeutigkeit von Normalformen?

Konfluenz

Eine zweistellige Relation ρ heißt *konfluent*, wenn

$$\forall x, y_1, y_2 : \rho^*(x, y_1) \wedge \rho^*(x, y_2) \Rightarrow \exists z : \rho^*(y_1, z) \wedge \rho^*(y_2, z)$$

(Bild ist einfacher zu merken)

Satz: wenn ρ auf M konfluent ist, dann besitzt jedes $x \in M$ höchstens eine ρ-Normalform.

Beachte: es wird nicht behauptet, daß x *überhaupt eine* Normalform besitzt.

Falls ρ jedoch terminiert, dann läßt sich Konfluenz charakterisieren und entscheiden durch einen Hilfsbegriff (lokale Konfluenz, später)

Lokale Konfluenz

Eine zweistellige Relation ρ heißt *lokal konfluent*, wenn

$$\forall x, y_1, y_2 : \rho(x, y_1) \wedge \rho(x, y_2) \Rightarrow \exists z : \rho^*(y_1, z) \wedge \rho^*(y_2, z)$$

Beachte: es gibt Relationen ρ, die lokal konfluent sind, aber nicht konfluent. Satz: wenn ρ terminiert und lokal konfluent ist, dann ist ρ konfluent.

Kritische Paare

Für ein Termersetzungssystem R über Σ: falls

- $(l_1 \to r_1)$ und $(l_2 \to r_2)$ sind Regeln in R ohne gemeinsame Variable (ggf. vorher umbenennen!)
- es gibt $p \in \mathrm{Pos}(l_1)$ so daß $l_1[p]$ und l_2 *unifizierbar* sind

  d. h., es existiert *mgu* σ mit $l_1[p]\sigma = l_2\sigma$
- falls $(l_1 \to r_1) = (l_2 \to r_2)$ (vor Umbenennung), dann $p \neq \epsilon$
- dann heißt $(r_1\sigma, (l_1\sigma)[p := r_2\sigma])$ *kritisches Paar* von R.

Ein kritisches Paar (s, t) heißt *zusammenführbar*, falls
$\exists u : s \to_R^* u \wedge t \to_R^* u$.
Satz: ein Termersetzungssystem ist genau dann lokal konfluent, wenn alle kritische Paare zusammenführbar sind.

Unifikation

- ein *Unifikator* von zwei Termen $s, t \in \mathrm{Term}(\Sigma, V)$

  ist eine Substitution $\sigma : V \to \mathrm{Term}(\Sigma, V)$ mit $t\sigma = s\sigma$
- zwei Terme s, t können keinen, einen oder mehrere Unifikatoren haben
- Substitutionen kann man ordnen durch $\sigma_1 \leq \sigma_2$ (σ_1 ist allgemeiner als σ_2) falls $\exists\tau : \sigma_2 = \sigma_1 \circ \tau$
- Satz: Wenn s, t unifizierbar sind, dann gibt es einen allgemeinsten Unifikator (most general unifier, mgu) σ von s und t:

  für jeden Unifikator σ_2 von s und t gilt $\sigma \leq \sigma_2$.

Bestimmung des mgu

(vergleiche Logische Programmierung)

- falls $s = t$, dann return identische Abbildung
- falls s Variable
  - falls s in t vorkommt, dann fail
  - sonst return $(s \mapsto t)$
- falls t Variable entsprechend
- falls $s = f(x_1, \ldots)$ und $t = g(y_1, \ldots)$, dann
  - falls $f \neq g$, dann fail, sonst …
  - bestimme $\sigma = \mathrm{mgu}(x_1, y_1)$
  - return $\sigma \cup \mathrm{mgu}((x_2\sigma, \ldots), (y_2\sigma, \ldots))$

Dieser Algorithmus ist korrekt, aber nicht effizient.

Orthogonale Systeme

- ein TES R über Σ heißt *nichtüberlappend*, wenn R keine kritischen Paare besitzt.
- ein TES R heißt linkslinear, wenn in keiner linken Regelseite eine Variable mehrfach vorkommt.
- ein TES R heißt *orthogonal*, falls es linkslinear und nichtüberlappend ist.

Satz: jedes orthogonale System ist konfluent.

Funktionale Programmierung $\approx$ System R ist orthogonal und ... (nächste Folie)

Konstruktor-Systeme

- Für TES R über Σ: ein Symbol $f \in \Sigma$ heißt *definiert*, wenn es als Wurzel einer linken Regelseite in R vorkommt.
- Alle anderen Symbole heißten *Konstruktoren*.
- R heißt Konstruktor-System, falls in jeder linken Seite nur ein definiertes Symbol vorkommt (und zwar in der Wurzel).

Beispiele: $\{P(E,y) \to \dots, P(S(x),y) \to \dots, M(E,y) \to \dots, M(S(x),y) \to \dots\}$ ist Konstruktor-System, definierte Symbole sind $\{P, M\}$, Konstruktoren sind $\{S, E\}$ aber $\{A(A(D,x),y) \to A(x, A(x,y))\}$ nicht.

XSLT

- (extensible style sheet language transformations)
- ist eine (seltsame) Art, ein Term-Ersetzungssystem für XML-Dokumente hinzuschreiben.
- besteht aus Ersetzungsregeln (Templates) und Ersetzungsstrategie
- Regeln haben Form $f(x_1, \dots, x_n) \to \dots$ für Elemente f, evtl. mit zusätzlichen Bedingungen
- ungefähr outermost rewriting (bei der Wurzel beginnend), behandelt aber jeden Knoten (normalerweise) höchstens einmal.

Ersetzungssysteme und Programmanalyse

- R ein Ersetzungssystem, L eine Sprache (Menge) von Bäumen,
  $$R^*(L) = \{t \mid \exists s \in L : s \to_R^* t\}$$
  alle Mehr-Schritt-Nachfolger von L.
- modelliert: R ein Programm, L eine Menge von (möglichen) Eingaben, dann $R^*(L)$ Menge von erreichbaren Zuständen (Zwischenergebnissen)
- E eine Menge von verbotenen Zuständen. Kein verbotener Zustand erreichbar: $R^*(L) \cap E = \emptyset$
- Beispiel: durch XSLT-Transformation soll immer Schema-konformes Dokument entstehen

Ersetzung und Automaten (Regularität)

- Normalformen
  - Normalformen für $\{fgfg \to \dots\}$?
  - Normalformen für $\{A(A(D,x),y) \to \dots\}$?
  - Ist Menge der Normalformen immer regulär? (Nein.)
- Nachfolgermengen
  - L und E durch endliche (Baum-)Automaten beschrieben. Wie kann man $R^*(L) \cap E = \emptyset$ entscheiden?
  - im Allgemeinen gar nicht, wg. Halteproblem
  - in Einzelfällen doch
    $\Sigma = \{a, b\}, R = \{aa \to aba\}, L = a^*, E = \Sigma^* bb \Sigma^*$

XSLT

Plan

- Paul Grosse, Norman Walsh: *XSL Concepts and Practical Use* `http://nwalsh.com/docs/tutorials/xsl/xsl/`
- Michael Kay (saxonica.com): Saxon-Dokumentation und -Beispiele `http://saxon.sourceforge.net/`
- Apache XML Project: Xalan Java-Dokumentation und -Beispiele `http://xml.apache.org/xalan-j/`

XSLT-Beispielaufgaben

- Benutzung von Saxon so:

```
java -jar /home/waldmann/built/saxonb8-6-1/sa
     /home/waldmann/built/saxonb8-6-1/samples,
     othello.xsl
     > othello.html
```

- XSLT-Erläuterungen siehe `http://nwalsh.com/docs/tutorials/xsl/xsl/slides.html`

Aufgaben:

- leere Transformation:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1
          version="1.0">
</xsl:transform>
```

(wendet Default-Templates an)

- LINE als `<P> .. </P>` formatieren
- SPEECH: Name des Sprechers als Überschrift setzen

XSLT-Definitionen

- ein `xsl:transform`-Dokument (Transform, Stylesheet)
- enthält `xsl:template`-Knoten (Templates):

```
<xsl:template match="Knoten-Auswahl">
   Ausgabe-Vorschrift
</xsl:template>
```

- Templates sind (im Prinzip) Term-Ersetzungs-Regeln, die nach einer bestimmten Strategie angewendet werden.

XSLT-Auswertung

- Rechnung beginnt in der Wurzel
- Pro Knoten wird normalerweise nur eine Regel angewendet.
- Was ist, wenn mehrere Regeln passen?

  Regeln haben Prioritäten:

  – die speziellere überstimmt die allgemeinere,
  – die frühere überstimmt die spätere.
- Auf den erzeugten Baum werden normalerweise keine Regeln angewendet.

Datenmodell von XPATH

- Wurzel

  genaue eine pro Dokument
- Element

  hat Elter und Liste von Kindern
- Attribut

  hat Elter (ist aber kein Kind) und Wert
- Text

  hat Elter (aber keine Kinder) und Wert

Achsen zur Navigation

(wie in Verzeichnisbäumen)

- nach unten:

  child, descendant, descendant-or-self (//)
- nach oben:

  parent, ancestor (..) , ancestor-or-self
- Geschwister:

  following-sibling, preceding-sibling
- Cousins usw.:

  following, preceding
- attribute

Achsen als Relationen

Die Achsen beschreiben Relationen auf Positionen.
Wiederholung zu Relationen (ich hoffe jedenfalls):

- eine *Relation* R über Menge M ist Menge von geordneten Paaren. $R \subseteq M \times M$.
- das *Produkt* von zwei Relationen R, S ist $R \circ S = \{(x,z) \mid \exists y : (x,y) \in R \wedge (y,z) \in S\}$
- die *transitive Hülle* von R ist $R^+ = R^1 \cup R^2 \cup \ldots$.
- die *reflexive und transitive Hülle* von R ist $R^{=}R^0 \cup R^1 \cup R^2 \cup \ldots$,

  dabei ist R^0 die identische Relation $\{(x,x) \mid x \in M\}$
- die inverse Relation von R ist $R^- = \{(y,x) \mid (x,y) \in R\}$

Achsen als Relationen

bezeichne mit C die child-relation. Das ist die Relation auf $\mathbb{N}^*$ ($=$ Positionen)
$C = \{(w, wx) \mid w \in \mathbb{N}^*, x \in \mathbb{N}\}$
Beispiel $([1,4,0],[1,4,0,3]) \in C$.

- descendant $= C^+$
- descendant-or-self $= C^*$
- parent $P = C^-$
- ancestor $= P^+ = (C^-)^+$
- ancestor-or-self $= P^* = (C^-)^*$

Achsen als Relationen (II)

Die Relation following-sibling ist:
$r = \{(wx, wy) \mid w \in \mathbb{N}^*, x \in \mathbb{N}, y \in \mathbb{N}, x < y\}$.
Beispiel: $([2,5,1],[2,5,3]) \in r$.
(r wegen *rechts*, alle weiter rechts stehenden Kinder des gleichen Elters)

- preceding-sibling $= l = r^-$

Achsen als Relationen (III)

was ist *following* (R) ? (großes R, wegen $r \subseteq R$)
Definition: alle Knoten, die *nach* dem End-Tag des Kontextknotens *beginnen*
$R = (C^-)^* \circ r \circ C^*$
das ergibt vollständige Fallunterscheidung: für beliebige Knoten (Positionen) x, y gilt genau eines von:
$x = y, (x,y) \in C^+, (x,y) \in (C^-)^+, (x,y) \in R, (x,y) \in R^-$

Kontrollfragen zu Achsen

- sind diese Relationen gleich: $r \circ R$ und $R \circ r$?
- wo liegen die Knoten x mit der Eigenschaft $R(x) = \emptyset$?

  dabei bedeutet $R(x) = \{y \mid (x, y) \in R\}$.

Schritte und Knoten-Tests

Lokalisierungs-Schritt hat die Form
`achse::test`
(Achse wie vorhin)
Test:

- *name*
- node ()
- text ()
- `*` alles

`@`*name* ist Abkürzung für `attribute::`*name*

Prädikate

jeder Lokalisierungs-Schritt beschreibt Liste von Knoten.
jeder Schritt kann mit Prädikaten versehen sein:
`/person[1]/profession[.="physicist"][position()`

- Prädikat vom Typ Knoten → Zahl:

  wähle Knoten dieser Nummer

- Prädikat vom Typ Knoten → Boolean:

  wähle Knoten mit Wert True

XSLT-Rechnungen

Ausgabe-Vorschrift kann enthalten:

- literalen Text `<P>, </P>`
- Verweise auf Werte (Knoten) des Originals

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

- Auswahl von (Kind-)Knoten (und Templates) für weitere Rechnung

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

Default-Regeln

sind implizit in jedem XSLT-Dokument vorhanden:

- Regeln auf Kinder von (Element-)Knoten anwenden:

```
<xsl:template match="*">
  <xsl:apply-templates/>
</xsl:template>
```

- Inhalt von Text- und Attribut-Knoten ausgeben:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

Diese kann man nicht beseitigen, aber überschreiben (durch Regeln höherer Priorität)

Push und Pull

zwei wesentlich verschiedene Stile:

- Pull (die Transformation *zieht* die Information aus der Eingabe)

  nur ein Template, enthält die Konstruktion der gesamten Ausgabe

  Struktur der gewünschten Ausgabe bestimmt den Ablauf

- Push (?)

  viele (kleine) Templates

  Struktur der Eingabe bestimmt den Ablauf

für kleine Aufgaben reicht Push, für kompliziertere braucht man wohl Pull.

Steuerung der Auswertung

- xsl:apply-templates (alle), xsl:call-template (eines)
- . . . eventuelle Parameter mit xsl:with-param übergeben und xsl:param benutzen
- xsl:for-each mit select
- xsl:if, xsl:choose, xsl:when, xsl:otherwise

Funktionales Programmieren mit XSLT

```
<xsl:template match="*" name="fun">
  <xsl:param name="x"/>
  <xsl:choose>
    <xsl:when test="0&lt;$x">
      <S>
      <xsl:call-template name="fun">
        <xsl:with-param name="x" select="$x -1"
      </xsl:call-template>
      </S>
    </xsl:when>
    <xsl:otherwise> <Z/> </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Programmieren (Kommentar)

... das ist ja ganz nett gemeint (nur Funktionen/Konstanten, keine Prozeduren/Variablen/Zuweisungen), aber softwaretechnisch furchtbar realisiert:

- groteske Syntax, komplizierte Parameterübergabe, kein Typsystem, keine Datenstrukturen (außer Strings), kein Modulsystem.
- das ist der typische (abschreckende) Fall einer domainspezifischen Sprache (DSL), die *alleinstehend* ist
- ... anstatt *eingebettet* in eine Gastsprache, deren Typ- und Modulsystem sowie Bibliotheken sie nutzen könnte

Aufgaben zu XSLT

- nur Text für Othello ausgeben
- SPEECH-Blöcke numerieren
- nur die jeweils erste (und letzte) LINE jeder SPEECH ausgeben
- Inhaltsverzeichnis herstellen
- mit `call-template` und `with-argument` einen vollständigen Binärbaum gegebener Tiefe herstellen

Formatting Objects

Idee

- FO ist XML-Beschreibungssprache für Dokumente
- durch Prozessoren werden aus FO konkret formatierte Ausgaben berechnet

  (PDF, PCL, PS, SVG, AWT, TXT, ...)
- FO-Dokument wird durch Transformation (XSLT) aus semantischen Dokument erzeugt

Literatur, Implementierung:
`http://xmlgraphics.apache.org/fop/`
Beispiele:

```
fop.sh -fo example.fo -pdf example.pdf
fop.sh -fo example.fo -awt
```

Dokument- und Seitenbeschreibungssprachen

für Eingabe? Speicherung? Ausgabe?

- Eingabe: WYSIWIG, textuell
- Speicherung: plattform-unabhängig? menschenlesbar?
- Ausgabe: für Drucker/Bildschirm

Eingabe und Speicherformat: logische Gliederung, Ausgabe: optische Gliederung. FO ist irgendwo dazwischen.

Rendering

bei Ausgabe (rendering) werden Objekte auf Druckfläche(n) angeordnet:

- relative Plazierungen (der Objekte zueinander, zum Container-Objekt) in absolute Maße umrechnen
- bei mehreren Wahlmöglichkeiten die günstigste (schönste) auswählen:

  Wörter auf Zeile anordnen (Silbentrennung!), Absätze auf Seiten anordnen

Layout-Manager

- Objekte als Rechtecke mit teilw. flexiblen Abmessungen (minimale, bevorzugte, maximale Größe).
- Container sind selbst Objekte.
- elementare Objekte: (Java) Button, Label, ..., $\TeX$: Buchstabe

FO und XSLT, Beispiele

Material:

- Homepage (C) Apache, `http://xmlgraphics.apache.org/fop/`
- FOP ist hier installiert (Linux-Pool)

  `/home/waldmann/built/fop-0.20.5`
- Quelltexte siehe `http://svn.apache.org/viewcvs.cgi/xmlgraphics/fop/branches/fop-0_20_2-maintain/src/org/apache/fop/`
- Tutorial (C) Antenna House, `http://www.antennahouse.com/XSLsample/XSLsample.htm`

Benutzung/Aufgabe

- Beispiel kopieren:

```
cp /home/waldmann/built/fop-0.20.5/examples/fo
```

- formatieren

```
fop.sh -fo simple.fo -pdf simple.pdf
fop.sh -fo simple.fo -ps simple.ps
fop.sh -fo simple.fo -awt
```

- Schreiben Sie eine XSLT-Transformation `book.xsl`, die Othello schön formatiert:

```
cp /home/waldmann/built/saxonb8-6-1/samples/da
```

```
fop.sh -xsl book.xsl -xml othello.xml -pdf oth
```

Ausblick, Zusammenfassung

Verarbeitung von XPATH-Ausdrücken

XPATH-Ausdruck R beschreibt Relation auf Knoten.
ausgehend von aktuellem Knoten x, beschreibt Menge $R(x) = \{y \mid (x, y) \in R\}$.
für Implementierung:

- alle Elemente solcher Mengen effizient bestimmen
- entscheiden, ob Menge leer ist

Endliche Automaten

wenn der Ausdruck R nur Achsen-Schritte und einfache Knotentests enthält, dann kann man $R(x)$ durch endlichen Automaten darstellen.
...und damit die Fragen effizient beantworten.
(*einfach*: Vergleich mit vorgegebenen Werten) (nicht einfach: Rechnungen mit Zahlen usw.)

Wenn das zu transformierende Dokument groß ist, lohnt sich die vorherige Berechnung solcher Hilfsmittel.

Beispiele: gegeben $w = [3, 1, 2]$. und maximaler Knotengrad 3. Bestimme reguläre Ausdrücke für die Mengen $C^*(w), R(w)$.

Prüfung von Transformationen

Wenn für die Transformation auch die XML-Schema für Eingabe und Ausgabe bekannt sind,
dann kann (sollte) man prüfen, ob jedes Resultat einer Transformation tatsächlich ein Dokument erzeugt, das zum Ausgabescheme paßt
...am liebsten automatisch.
Die Schemas beschreiben reguläre Baumsprachen, die Transformation ist ein Term-Ersetzungs-System, also ist das die Frage:
für welche Ersetzungssysteme R gilt: wenn L eine reguläre Baumsprache, dann ist auch $R^*(L)$ regulär und kann effektiv bestimmt werden?

Regularitäts-Erhaltung

im Allgemeinen sind Ersetzungssysteme nicht REG-erhaltend, aber in Spezialfällen (besondere Form der Regeln) doch.

...ist aktuelles Forschungsgebiet innerhalb der Termersetzung, z. B. auf Konferenzen RTA (Rewriting Techniques and Applications).

Beispiele (Wortersetzung): alle linken Seiten Länge $= 1$ oder alle rechten Seiten Länge ≤ 1 (aber nicht beides gleichzeitig!)

Beispiel Für $S = \{100 \to 011, 001 \to 110\}$ (Solitär-Spiel) bestimme $L = S^*(0^*10^*)$ (abräumbare Anordnungen).
Diese Menge ist eine reguläre Sprache.

Wiederholung

- Terme = Bäume, Positionen = $\mathbb{N}^*$, Teilbäume, XPATH-Datenmodell, DOM, (un)marshalling, XML-RPC
- Relationen auf Positionen, Navigation (XPATH-Achsen)
- Beschreibung von Baumsprachen durch endliche Automaten $\approx$ XML-Schema
- Term-Ersetzungs-Systeme (Regeln, Substitutionen) $\approx$ XSLT, Anwendung: FO

XML sucks?

Aaron Crane, `http://xmlsucks.org/but_you_have_to_use_it_anyway/`

- XML is a giant step in no direction at all. (Eric Naggum)
- ist zunächst *nur* Syntax, hilft überhaupt nichts in Bezug auf Inhalt
- selbst die Syntax ist zweifelhaft (obscenely verbose, Attribute/Content?, Trickesereien (CDATA))

  und die Idee einer generischen Syntax für Bäume gibt es schon lange (LISP, 1960)
- zur Beschreibung/Verarbeitung des Inhalts benutzt man klassische Modelle (endliche Automaten, Typsysteme, Termersetzung, funktionale Programmierung)

XML sucks?

Die Probleme, die XML löst (Baum als Text darstellen), sind nicht schwer, und die Lösung ist nicht sehr gut, aber *alle machen mit*:

- XML zu benutzen, kostet Geld
- XML nicht zu benutzen, kostet eventuell mehr Geld.

man wird durch XML-Anbindung dazu angehalten, über externe Repräsentationen der Daten nachzudenken, die ein Programm verarbeitet. Das sollte man sowieso tun:

Programme sind *Werkzeuge* zur Datenverarbeitung, d. h. sie lesen und schreiben Daten. Die Daten sollen *unabhängig* von den Werkzeugen sein, damit man Werkzeuge kombinieren und austauschen kann.

autotool-Auswertung

- ```
  40 :      32457  Christian Fröhlich
  24 :      36189  Mathias Schilha
  18 :      36178  Stephan Krull
  ```
- Aufgabe TRS-Times:

```
TRS { variablen = [ x,y,a,b,t ] , regeln = [
f(x,y) -> g(x,y,Z),
g(Z,y,t) -> t, g(S(x),y,t) -> h(x,y,Z,Z,t),
h(x,S(y),a,b,t) -> h(x,y,S(a),S(b),t),
h(x,Z,a,b,t) -> g(x,a,add(b,t)),
add(S(b),t) -> add(b,S(t)), add(Z,t) -> t
] }
```
```