Im Prinzip sind diese Elemente ausreichend, um XML-Dokumente zu verarbeiten, d.h. um Teilinformation zu extrahieren oder zu transformieren. Zum Beispiel können wir den Namen und die Telefonnummer aus einer `entry`-Struktur mittels Pattern Matching wie folgt extrahieren (man beachte, dass wir hier für eine besser lesbare Regel funktionale Muster verwenden):

```
getNamePhone
  (xml "entry"
       [xml "name"  [xtxt name],
        _,
        xml "phone" [xtxt phone]])  =  name ++ ": " ++ phone
```

Diese Lösung hat allerdings einige Nachteile:

- Die exakte Struktur der XML-Dokumente muss vollständig bekannt sein. So kann `getNamePhone` nur auf Einträge mit genau drei Komponenten angewendet werden, sodass sie auf beide obigen `entry`-Strukturen nicht erfolgreich anwendbar ist.
- In großen XML-Dokumenten sind viele Teile für die jeweilige Anwendung irrelevant. Trotzdem muss man Muster für das komplette Dokument angeben.
- Wenn sich die Struktur des Gesamtdokuments ändert, muss man viele Muster entsprechend anpassen, was leicht zu Fehlern führen kann.

Diese Probleme können mittels mächtigerer funktionaler Muster vermieden werden. Nachfolgend geben wir einige Muster hierzu an.

Partielle Muster

Mittels funktionaler Muster können wir z.B. an Stelle der exakten Liste aller Teildokumente nur die relevanten angeben und die übrigen weglassen:

```
getNamePhone
  (xml "entry"
    (with [xml "name"  [xtxt name],
           xml "phone" [xtxt phone]]))  =  name ++ ": " ++ phone
```

Die Bedeutung des Operators "`with`" ist, dass die angegebenen Teildokumente vorkommen müssen, aber dazwischen beliebig viele andere Elemente stehen dürfen. Diesen können wir in Curry wie folgt definieren:

```
with :: Data a => [a]  →  [a]
with []     = _
with (x:xs) = _ ++ x : with xs
```

Somit wird z.B. der Ausdruck "`with [1,2]`" zu jeder Liste der Form

$$x_1 : \ldots : x_m : 1 : y_1 : \ldots : y_n : 2 : zs$$

ausgewertet, wobei x_i, y_j, zs neue logische Variablen sind. Damit passt die Definition von `getNamePhone` auf jede `entry`-Struktur, die die `name`- und `phone`-Strukturen als Kinder enthält. Solch ein **partielles Muster** ist also robust gegen weitere Teilstrukturen, die vielleicht in zukünftigen XML-Dokumenten hinzugefügt werden.

Ein Nachteil dieser Definition von `getNamePhone` ist, dass diese nur auf XML-Strukturen mit leerer Attributliste passt. Falls wir an XML-Strukturen mit beliebigen Attributen interessiert sind, könnten wir die Operation

```
xml' :: String → [XmlExp] → XmlExp
xml' t xs = XElem t _ xs
```

definieren, die ein XML-Dokument mit beliebigen Attributen beschreibt. Damit könnten wir z.B. die Operation `getName` definieren:

```
getName (xml' "entry" (with [xml' "name" [xtxt n]])) = n
```

Diese liefert den Namen in einer `entry`-Struktur unabhängig davon, ob die Strukturen Attribute enthalten.

Ungeordnete Muster

Wenn die Struktur der Daten bei der Entwicklung eines Softwaresystems verändert wird, könnte sich auch die Reihenfolge der Elemente in einem XML-Dokument ändern. Um unsere Programme dagegen robust zu machen, können wir festlegen, dass die Teildokumente in einer beliebigen Reihenfolge auftreten dürfen, indem wir den Operator "`anyorder`" davor setzen:

```
getNamePhone
  (xml "entry"
    (with
      (anyorder [xml "phone" [xtxt phone],
                 xml "name"  [xtxt name]]))) = name ++ ": " ++ phone
```

Hierbei berechnet `anyorder` eine beliebige Permutation der Argumentliste, sodass wir den Code hierfür schon kennen:

```
anyorder :: [a] → [a]
anyorder []     = []
anyorder (x:xs) = insert (anyorder xs)
 where
  insert ys     = x : ys
  insert (y:ys) = y : insert ys
```

Die letzte Definition von `getNamePhone` passt also auf beide `entry`-Strukturen unseres Beispieldokumentes.

Tiefe Muster

Wenn man Informationen in einem tief verschachtelten XML-Dokument sucht, dann ist es unschön, den kompletten Pfad von der Wurzel bis zum Teildokument anzugeben. Besser wäre es, nur die Teildokumente zu spezifizieren und ein Matching in beliebiger Tiefe zuzulassen. Hier nutzen wir die Operation "`deepXml`", die an Stelle von `xml` in einem Muster verwendet werden kann und passt, falls dies an einer beliebigen Stelle im gegebenen Dokument vorkommt. Wenn wir z.B. `getNamePhone` durch

```
getNamePhone
  (deepXml "entry"
     (with [xml "name"  [xtxt name],
            xml "phone" [xtxt phone]]))  =  name ++ ": " ++ phone
```

definieren und auf das komplette Beispieldokument anwenden, dann erhalten wir zwei berechnete Werte. Die Menge oder Liste aller berechneten Werte könnten wir dann mittels eingekapselter Suche erhalten.

Die Implementierung von `deepXml` ist ähnlich wie die von `with`, allerdings berechnen wir alle Strukturen, die das angegebene Element als Wurzel oder Nachfolger enthalten:

```
deepXml :: String  →  [XmlExp]  →  XmlExp
deepXml tag elems = xml' tag elems
deepXml tag elems = xml' _ (_ ++ [deepXml tag elems] ++ _)
```

Negierte Muster

Manchmal ist es von der Anwendung her sinnvoll, XML-Dokumente zu suchen, die bestimmte Muster nicht erfüllen. Zum Beispiel könnte man alle Einträge heraussuchen, die keine E-Mail-Adresse haben. Da die allgemeine Negation von Berechnungen in der logisch-funktionalen Programmierung schwierig ist, könnten wir stattdessen konstruktiv vorgehen und eine Operation "`withOthers`" definieren, die sich wie "`with`" verhält, allerdings in einem zweiten Argument die Teildokumente liefert, die *nicht* im Muster enthalten sind. Auf diesen kann man dann weitere Bedingungen formulieren. Wenn wir z.B. den Namen und Telefonnummer eines `entry`-Dokumentes wissen wollen, das keine E-Mail-Adresse hat, dann können wir dies wie folgt definieren:

```
getNamePhoneWithoutEmail
  (deepXml "entry"
    (withOthers [xml "name" [xtxt name], xml "phone" [xtxt phone]] others))
  | "email" `noTagOf` others  =  name ++ ": " ++ phone
```

Hierbei ist das Prädikat `noTagOf` erfüllt, falls das angegebene Element nicht vorkommt (die Operation `tagOf` liefert die Markierung eines XML-Dokumentes):

```
noTagOf :: String  →  [XmlExp]  →  Bool
```

```
noTagOf tag = all ((/= tag) . tagOf)
```

Somit liefert `getNamePhoneWithoutEmail` für unser Gesamtdokument genau einen Wert zurück.

Die Implementierung von `withOthers` ist etwas komplizierter als `with`, da wir die übrigen Element akkumulieren müssen:

```
withOthers :: Data a => [a] → [a] → [a]
withOthers ys zs = withAcc [] ys zs
 where -- Accumulate remaining elements:
   withAcc prevs [] others | others === prevs ++ suffix = suffix
                                                 where suffix free
   withAcc prevs (x:xs) others =
      prefix ++ x : withAcc (prevs ++ prefix) xs others
                                                 where prefix free
```

Somit wird der Ausdruck "`withOthers [1,2]` *os*" zu jeder Liste der Form

$x_1 : \ldots : x_m : 1 : y_1 : \ldots : y_n : 2 : zs$

ausgerechnet, wobei $os = x_1 : \ldots : x_m : y_1 : \ldots : y_n : zs$. Wenn wir diesen Ausdruck also als funktionales Muster verwenden, dann passt dieser auf jede Liste, die die Elemente `1` und `2` enthält, wobei die Variable *os* an die Liste der übrigen Dokumente gebunden wird.

Transformation von XML-Dokumenten

Da Transformationaufgaben eine der Stärken deklarativer Programmierung ist, ist natürlich auch die Transformation von XML-Dokumenten einfach nach dem Schema

transform pattern = *newdoc*

möglich. Zum Beispiel könnten wir ein `entry`-Dokument in eine andere XML-Struktur übersetzen, die die Telefonnummer und den vollen Namen der Person enthält:

```
transPhone (deepXml "entry" (with [xml "name"  [xtxt n],
                                   xml "first" [xtxt f],
                                   xml "phone" phone])) =
   xml "phonename" [xml "phone" phone, xml "fullname" [xtxt (f++' ':n)]]
```

Da die Anwendung von `transPhone` auf unser Beispieldokument nichtdeterministisch zwei neue XML-Dokumente liefert, möchte man diese zu einem neuen Dokument zusammenfassen. Mittels eingekapselter Suche ist dies einfach möglich. Beispielsweise liefert der Ausdruck

```
xml "table" (sortValues ((set1 transPhone) c))
```

eine vollständige Tabelle aller Paare von Telefonnummern und vollen Namen aus dem Dokument c.

Ebenso können wir auch Mengenfunktionen schachteln, um Zwischeninformationen zu akkumulieren. Als Beispiel könnten wir eine Liste aller Personen mit der Anzahl ihrer E-Mail-Adressen erstellen. Hierzu definieren wir eine Operation, die für ein `entry`-Dokument den Namen und die Anzahl der E-Mail-Adressen liefert:

```
getEmails (deepXml "entry" (withOthers [xml "name" [xtxt name]] os)) =
          (name, length (sortValues ((set1 emailOf) os)))

emailOf (with [xml "email" email]) = email
```

Die vollständige Liste dieser Einträge können wir aus dem Dokument c wie folgt berechnen:

```
sortValues ((set1 getEmails) c)
```

Für unser Beispieldokument wird damit `[("Hanus",2),("Smith",0)]` ausgerechnet.

Anwendung

Diese Art der XML-Verarbeitung zeigt noch einmal die generellen Vorteile der deklarativen Programmierung. Durch die Kombination mächtiger Konzepte wie musterorientierte Programmierung, Nichtdeterminismus und Suche erhält man einfache, lesbare und ausführbare Problemlösungen. Da die Lesbarkeit und Effizienz der Programmentwicklung im Vordergrund stehen, aber zunächst einmal nicht die Effizienz der Ausführung, hängt es immer von der konkreten Anwendung ab, ob die Laufzeiteffizienz ausreichend ist. Obwohl man nicht erwarten kann, dass man in den Laufzeiten mit spezialisierten XML-Implementierungen konkurrieren kann, kann dies trotzdem für viele Anwendungen ausreichend sein. Zum Beispiel ist es bei unserer Implementierung so, dass häufig die Zeit zum (deterministischen!) Einlesen eines XML-Dokumentes länger ist als die Zeit zu dessen (nichtdeterministischer) Verarbeitung. So wird diese einfache Implementierung auch eingesetzt, aus dem UnivIS der CAU Kiel[19] die Veranstaltungsdaten im XML-Format zu extrahieren und in die Moduldatenbank des Instituts[20] automatisch einzupflegen. In dieser Anwendung ist es sehr nützlich, tiefe und partielle XML-Muster zu spezifizieren, um die große Datenmenge der UnivIS-XML-Dokumente, von denen der Großteil ignoriert wird, einfach zu verarbeiten.

[19] `http://univis.uni-kiel.de/`
[20] `https://mdb.ps.informatik.uni-kiel.de/`