2.3 Pattern Matching

Wie wir nun schon an verschiedenen Beispielen gesehen haben, wird in deklarativen Programmiersprachen der folgende Programmierstil bevorzugt:

> Definition der Operationen durch Muster (pattern) und mehrere Gleichungen

Auf der anderen Seite werden Ausdrücke durch

> Auswahl und Anwenden einer passenden Gleichung

ausgewertet, d.h. ersetze „Gleiches durch Gleiches“. Diese Auswahl einer passenden Gleichung wird dabei auch als “pattern matching“ bezeichnet.

Vergleiche: Definition der Konkatenation von Listen:

Musterorientierte Definition:

```
append []     ys = ys
append (x:xs) ys = x : (append xs ys)
```

Definition ohne Muster (wie in klassischen Programmiersprachen):

```
append xs ys = if xs == []
                 then ys
                 else (head xs): (append (tail xs) ys)
```

wobei `head`, `tail` vordefinierte Funktionen auf Listen sind:

`head` liefert das Kopfelement der Liste
`tail` liefert den Rest der Liste

d.h. es gilt immer:

```
head (x:xs) == x
tail (x:xs) == xs
```

bzw.

```
(head xs) : (tail xs) == xs
```

Andere Sprechweise:

- “`:`” ist ein **Konstruktor** für den Datentyp Liste
- `head`, `tail` sind **Selektoren** für den Datentyp Liste

Kernideen des **musterorientierten Stils**:

- Schreibe Konstruktoren in Argumente von linken Regelseiten
- Konstruktoren in linken Seiten haben die Wirkung von Selektoren

- Linke Seiten beschreiben somit einen „Prototyp“ für den Funktionsaufruf, bei dem diese Regel anwendbar ist, dar.

Beispiel:

```
append [] ys = ys    -- (*)
```

ist anwendbar, falls 1. Argument die Form einer leeren Liste hat und das 2. Argument beliebig ist.

Wann passt ein Muster?

> Ersetze Variablen in Muster (“binde Variablen“) so durch andere Ausdrücke, dass das ersetzte Muster syntaktisch identisch zum gegebenen Ausdruck ist. Ersetze in diesem Fall den Ausdruck durch rechte Regelseite (mit den entsprechenden Bindungen!).

Beispiel: Regel `(*)` und Ausdruck

- `append [] [1,2,3]`: Binde `ys` an `[1,2,3]`
  ⇒ linke Seite passt ⇒ Ersetze Ausdruck durch “`[1,2,3]`”
- `append [1] [2,3]`: Muster in `(*)` passt nicht, da Konstruktor `[]` immer verschieden vom Konstruktor “`:`” ist (beachte: `[1] == (:) 1 []`)

Was sind mögliche Muster in Haskell (es gibt aber noch mehr...)?

Muster	Das Muster passt, falls...
`x`	Variable passt auf jeden Ausdruck und wird an diesen gebunden
`[] True 'a'`	Konstanten passen nur auf gleichen Wert
`(x:xs) (Node` t_1 t_2`)`	Konstruktoren passen nur auf gleichen Konstruktor. Zusätzlich müssen die Argumente jeweils passen
(t_1, t_2, t_3)	Tupel passt auf Tupel gleicher Stelligkeit (Argumentzahl), zusätzlich müssen die Argumente jeweils passen

`v@pat`	Passt, falls Muster `pat` “passt“. Zusätzlich wird `v` an gesamten Wert gebunden
	Beispiel:
	`last [x] = x`
	`last (_ : xs@(_:_)) = last xs`
	Effekt: Vermeidung des Aufbaus der nichtleeren Liste `xs` in der rechten Seite

`_`	Joker, “wildcard pattern“: Passt auf jeden Ausdruck, keine Bindung

Ein potenzielles Problem kann sich ergeben, wenn wir Muster mit mehrfachem Vorkommen derselben Variablen zulassen:

```
equ x x = True
```

Woran wird die Variable `x` gebunden, d.h. an das erste oder zweite aktuelle Argument? Da dies unklar ist, sind solche Muster unzulässig!

In Mustern (linken Regelseiten) kommt jede Variable höchstens einmal vor.

Ausnahme: anonyme Variable "`_`": Jedes Vorkommen steht für eine andere Variable und kann somit an einen anderen Ausdruck gebunden werden.

Ein weiteres Problem kann sich ergeben, wenn auswertbare Ausdrücke an Positionen vorkommen, wo man den Wert wissen muss.

Beispiel:

```
fac 0 = 1
```

Betrachten wir nun den Aufruf: `fac (0+0)`
Das aktuelle Argument `(0+0)` passt erst nach Auswertung zu `0`, daher:

> Falls beim Pattern Matching der Wert relevant ist (d.h. bei allen Mustern außer Variablen), wird der entsprechende aktuelle Parameter ausgewertet.
>
> **Die Form der Muster beeinflusst das Auswertungsverhalten und damit auch das Terminationsverhalten!**

Beispiel: Betrachte folgende Definition der Konjunktion: ("Wahrheitstafel"):

```
and   False   False   =  False
and   False   True    =  False
and   True    False   =  False
and   True    True    =  True
```

Effekt der Muster: bei `(and` t_1 t_2`)` müssen beide Argumente ausgewertet werden, um eine Regel anzuwenden.

Betrachten wir nun folgende nichtterminierende Funktion:

```
loop :: Bool
loop = loop   -- Auswertung von loop terminiert nicht
```

Dann terminiert die Auswertung des Ausdrucks

```
(and False loop)
```

nicht!

Verbesserte Definition durch Zusammenfassen von Fällen:

```
and1 False x = False
and1 True  x = x
```

Effekt: 2. Argument muss nicht ausgewertet werden:

```
(and1 False loop)  ⇝ False
```

Weiteres mögliches Problem: Reihenfolge der Abarbeitung von Mustern:

Betrachten wir hierzu das logische Oder ("parallel or"):

```
or True    x    = True       (1)
or  x    True   = True       (2)
or False False  = False      (3)
```

Gegeben sei der Ausdruck

```
(or loop True)
```

Hierzu passt die Regel (2) (mit der Bindung `x` $\mapsto$ `loop`), sodass wir als Ergebnis `True` erhalten.

Aber die Regel (1) könnte auch passen: Auswertung des 1. Argumentes $\rightsquigarrow$ Endlosschleife

Dies zeigt, dass die Musterabarbeitung in einer Programmiersprache genau festgelegt werden muss. In Haskell ist diese informell wie folgt definiert:

> Alle Regeln werden von oben nach unten im Programm ausprobiert. Bei jeder Regel werden die Muster links nach rechts abgearbeitet. Dabei wird ein Argument ausgewertet, falls ein Muster in einer Regel dies verlangt.

Nachfolgend wollen wir diese Strategie präzisieren. Wir verwenden dazu `case`-Ausdrücke, die die folgende Form haben (Haskell erlaubt auch allgemeinere Formen):

`case` e `of`

$$\begin{array}{l} C_1\ x_{11} \ldots x_{1n_1} \to e_1 \\ \quad\vdots \qquad\qquad\quad \vdots \\ \quad\vdots \qquad\qquad\quad \vdots \\ C_k\ x_{k1} \ldots x_{kn_k} \to e_k \end{array}$$

wobei $C_1, \ldots, C_k$ Konstruktoren des Datentyps von e sind.

Beispiel: Mittels eines `case`-Ausdrucks kann die obige `append`-Operation auch wie folgt definiert werden:

```
append xs ys = case xs of
                 []    → ys
                 x:zs  → x : append zs ys
```

Bedeutung des `case`-Ausdrucks:

1. Werte e aus (bis ein Konstruktor oben steht)

2. Falls die Auswertung $C_i\ a_1 \ldots a_{n_i}$ ergibt (sonst: Fehler), binde x_{ij} an a_j $(j = 1, \ldots, n_i)$ und ersetze den gesamten `case`-Ausdruck durch e_i

Das obige Beispiel zeigt, dass Definitionen mit Pattern Matching in `case`-Ausdrücke übersetzbar sind. Im Folgenden werden wir dies präzisieren, indem wir die Semantik von Pattern Matching durch die Übersetzung in `case`-Ausdrücke definieren. Prinzipiell kann man dies mit unterschiedliche Verfahren machen. Wir orientieren uns hier an der in [Wadler 87] dargestellten Methode.

Gegeben: Funktionsdefinition

$$
\begin{array}{l}
f\ p_{11} \ldots p_{1n}\ \texttt{=}\ e_1 \\
\vdots \\
f\ p_{m1} \ldots p_{mn}\ \texttt{=}\ e_m
\end{array}
$$

Wir übersetzen diese Definition in `case`-Ausdrücke mittels einer Transformation `match`, die wir nachfolgend genauer definieren:

$$
\begin{array}{l}
\texttt{f}\ x_1 \ldots x_n\ \texttt{=} \\
\quad [\![\texttt{match [}x_1\texttt{, ...,}\ x_n\texttt{] [([}p_{11}\texttt{, ...,}\ p_{1n}\texttt{],}\ e_1\texttt{),}
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{([}p_{m1} \ldots\texttt{,}\ p_{mn}\texttt{],}\ e_m\texttt{)]} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{ERROR}]\!]
\end{array}
$$

Die Klammern $[\![\ldots]\!]$ bedeuten, dass der darin enthaltene Ausdruck (zur Übersetzungszeit) zu einem Programmausdruck ausgewertet wird.

Um die Funktion der Transformation `match` besser zu verstehen, betrachten wir die allgemeine Form

```
match xs eqs E
```

Hier ist

`xs`: Liste von Argumentvariablen

`eqs`: Liste von Pattern-Rumpf-Paaren

`E`: Fehlerfall, falls kein Pattern aus `eqs` passt

Wir definieren `match` durch vier Transformationsregeln:

1. **Alle Muster fangen mit einer Variablen an:**

$$
\begin{aligned}
&[\![\texttt{match (x:xs) [(}v_1 : ps_1,\ \ e_1\texttt{),} \\
&\qquad\qquad\qquad\vdots \\
&\qquad\qquad\qquad\texttt{(}v_m : ps_m,\ \ e_m\texttt{)] E}]\!] \\
&= [\![\texttt{match xs [(}ps_1,\ \ e_1[v_1 \mapsto \texttt{x}]\texttt{),} \\
&\qquad\qquad\qquad\vdots \\
&\qquad\qquad\qquad\texttt{(}ps_m,\ \ e_m[v_m \mapsto \texttt{x}]\texttt{)] E}]\!]
\end{aligned}
$$

   Hierbei steht $e[v \mapsto t]$ für die Ersetzung aller Vorkommen der Variablen v in e durch t.

   Somit werden bei Variablenmustern keine Argumente ausgewertet.

   Anmerkung: diese Transformationsregel ist auch anwendbar falls $m = 0$ (leere Gleichungsliste).

2. **Alle Muster fangen mit einem Konstruktor an:**
Dann Argument auswerten + Fallunterscheidung über Konstruktoren:

$[\![$`match (x:xs) eqs E`$]\!]$ `-- wobei eqs`$\neq$`[]`
 = $[\![$`match (x:xs) (eqs`$_1$ `++` $\cdots$ `++ eqs`$_k$`) E`$]\!]$

Hierbei ist `(eqs`$_1$ `++` $\cdots$ `++ eqs`$_k$`)` eine Gruppierung von `eqs`, sodass alle Muster in `eqs`$_i$ mit dem Konstruktor C_i beginnen, wobei $C_1, \ldots, C_k$ alle Konstruktoren vom Typ von `x` sind, d.h.

$$\texttt{eqs}_i \; = \; \texttt{[}((C_i \;\; p_{i,1,1} \ldots p_{i,1,n_i}) : ps_{i,1}, \;\; e_{i,1})\texttt{,} \\ \vdots \\ ((C_i \;\; p_{i,m_i,1} \ldots p_{i,m_i,n_i}) : ps_{i,m_i}, \;\; e_{i,m_i})\texttt{]}$$

Beachte: falls in `eqs` der Konstruktor C_j nicht vorkommt, dann ist `eqs`$_j$ `= []`. Außerdem haben in jedem `eqs`$_i$ die Regeln die gleiche Reihenfolge wie in `eqs`.

Nun transformieren wir weiter:

$[\![$`match (x:xs) (eqs`$_1$ `++` $\cdots$ `++ eqs`$_k$`) E`$]\!]$ =
 `case x of`
  $C_1 \;\; x_{1,1} \ldots x_{1,n_1} \;\; \rightarrow [\![\texttt{match ([}x_{1,1}, \ldots, x_{1,n_1}\texttt{]++xs) } eqs'_1 \texttt{ E}]\!]$
  :
  :
  $C_k \;\; x_{k,1} \ldots x_{k,n_k} \;\; \rightarrow [\![\texttt{match ([}x_{k,1}, \ldots, x_{k,n_k}\texttt{]++xs) } eqs'_k \texttt{ E}]\!]$

wobei

$$eqs'_i \; = \; \texttt{[([}p_{i,1,1}, \ldots, p_{i,1,n_i}\texttt{] ++ } ps_{i,1}\texttt{, } e_{i,1}\texttt{),} \\ \vdots \\ \texttt{([}p_{i,m_i,1}, \ldots, p_{i,m_i,n_i}\texttt{] ++ } ps_{i,m_i}\texttt{, } e_{i,m_i}\texttt{)]}$$

und $x_{i,j}$ neue Variablen sind.

Beispiel: Übersetzung von `append`:

```
append x1 x2
 = ⟦match [x1, x2] [(([], ys], ys),
                    ([x:xs, ys], x: append xs ys)] ERROR⟧
 = case x1 of
    []     →  ⟦match [x2] [([ys], ys)] ERROR⟧
    x3:x4  →  ⟦match [x3, x4, x2] [([x,xs,ys], x: append xs ys)] ERROR⟧
```

Nun wenden wir die 1. Transformation auf den ersten `case`-Zweig an:

```
= case x1 of
    []     → ⟦match [] [([], x2)] ERROR⟧
    x3:x4  → ⟦match [x3, x4, x2] [([x,xs,ys], x: append xs ys)] ERROR⟧

= case x1 of
    []     → x2
    x3:x4  → ⟦match [x3, x4, x2] [([x,xs,ys], x: append xs ys)] ERROR⟧
```

Der letzte Transformationschritt ist sinnvoll, da die Musterliste in

```
⟦match [] [([], x2)] ERROR⟧
```

leer ist und somit das Pattern Matching erfolgreich war. Genau hierfür benötigen wir die nachfolgende Transformationsregel.

3. **Musterliste ist leer:**
   Hier können wir zwei Fälle unterscheiden:

   a) Genau eine Regel ist anwendbar:

   $[\![$`match [] [([], e)] E`$]\!] =$ `e`

   b) Keine Regel ist anwendbar: Benutze nun die Fehleralternative:

   $[\![$`match [] [] E`$]\!] = [\![$`E`$]\!]$

   Anmerkung: Theoretisch kann evtl. auch mehr als eine Regel übrigbleiben, z.B. bei

```
f True  = 0
f False = 1
f True  = 2
```

   In diesem Fall kann der Übersetzer z.B. eine Fehlermeldung machen oder einfach die erste Alternative wählen.

4. **Muster fangen mit Konstruktoren als auch Variablen an:**
   Gruppiere diese Fälle:

   $[\![$`match xs eqs E`$]\!] =$
   $[\![$`match xs eqs`$_1$ `(match xs eqs`$_2$ `(...(match xs eqs`$_n$ `E) ...))`$]\!]$

   wobei

   `eqs ==` $\texttt{eqs}_1$ `++` $\texttt{eqs}_2$ `++ ...++` $\texttt{eqs}_n$

   und in jedem $\texttt{eqs}_i$ `(`$\neq$`[])` beginnen alle Muster entweder mit Variablen oder mit Konstruktoren; falls in $\texttt{eqs}_i$ alle Muster mit Variablen anfangen, dann fangen in $\texttt{eqs}_{i+1}$ alle Muster mit Konstruktoren an (oder umgekehrt)

Dieser Algorithmus hat folgende Eigenschaften:

1. **Vollständigkeit**: Jede Funktionsdefinition wird in `case`-Ausdrücke übersetzt (dies ist klar wegen der vollständigen Fallunterscheidung aller Transformationsregeln).

2. **Terminierung**: Die Anwendung der Transformationen ist endlich.
   Definiere hierzu:

   - Größe eines Musters: Anzahl der Symbole in diesem
   - Größe einer Musterliste: Summe der Größen der einzelnen Muster

   Dann gilt: jeder `match`-Aufruf wird durch Aufrufe mit kleinerer Musterlistengröße in einer Transformation ersetzt, woraus die Terminierung des Verfahrens folgt.

Beispiel: Übersetzung der logischen Oder-Operation:

```
or  True   x     = True    (1)
or  x      True  = True    (2)
or  False  False = False   (3)
```

Im ersten Schritt wird diese Operation wie folgt übersetzt:

```
or x y =   ⟦ match [x,y] [([True , x    ], True),
                          ([x    , True ], True),
                          ([False, False], False)] ERROR ⟧
```

Die Anwendung der 4. Transformation auf den `match`-Ausdruck ergibt:

```
⟦ match [x,y] [([True, x], True)]
    (match [x,y] [([x, True], True)]
      (match[x,y] [([False, False], False)] ERROR)) ⟧
```

Dieser Ausdruck wird wie folgt weiter übersetzt, wobei die Art der einzelnen Transformationsschritte links notiert ist:

```
2          case x of
1,3          True   → True
1,2          False  → case y of
3                       True  → True
2                       False → case x of
3                                  True  → ERROR
2                                  False → case y of
3                                             True  → ERROR
3                                             False → False
```

Nun ist klar, wie `(or loop True)` abgearbeitet wird: zunächst erfolgt eine Fallunterscheidung über das 1. Argument, was zu einer Endlosschleife führt.

Allerdings gibt es keine Endlosschleife, wenn die Regeln `(1)` und `(2)` in der Reihenfolge vertauscht werden!

Ursache: Regeln `(1)` und `(2)` "überlappen", d.h. für das Muster `(or True True)` sind beide Regeln anwendbar

Somit gilt:

> **Bei Funktionen mit überlappenden Mustern hat die Regelreihenfolge Einfluss auf den Erfolg des Pattern Matching.**

Als Konsequenz gilt die Empfehlung:

> **Vermeide überlappende linke Regelseiten!**

Ist dies allerdings ausreichend für Reihenfolgeunabhängigkeit?

Nein! Betrachte hierzu das folgende Beispiel:

```
diag x     True   False  =  1
diag False x      True   =  2
diag True  False  x      =  3
```

Diese Gleichungen sind nicht überlappend. Betrachten wir den folgenden Ausdruck:

```
diag loop True False
```

Dieser Ausdruck wird zu `1` ausgewertet. Wenn allerdings die Gleichungen in umgekehrter Reihenfolge umsortiert werden, führt die Auswertung dieses Ausdrucks zu einer Endlosschleife!

Die Ursache liegt in einem **prinzipiellem Problem:**

> Es existiert nicht für jede Funktionsdefinition eine sequentielle Auswertungsstrategie, die immer einen Wert berechnet.
> (vgl. dazu [Huet/Lévy 79, Huet/Lévy 91]).

Die Ursache für diese Probleme liegt in der Tranformation 4, bei der Auswertung und Nichtauswertung von Argumenten gemischt ist. Daher definieren wir die folgende Klasse von Funktionen:

> Eine Funktionsdefinition heißt **uniform**, falls die Transformation 4 bei der Erzeugung von `case`-Ausdrücken nicht benötigt wird.

Intuitiv bedeutet dies: Keine Vermischung von Variablen und Konstruktoren an gleichen Positionen.

Satz 2.1 *Bei uniformen Definitionen hat die Reihenfolge der Regeln keinen Einfluss auf das Ergebnis.*

Beispiel:

```
xor False x     = x
xor True  False = True
xor True  True  = False
```

ist uniform (trotz Vermischung im 2. Argument) wegen der „links-nach-rechts“-Strategie beim Pattern-Matching.

Dagegen ist

```
xor' x     False = x
xor' False True  = True
xor' True  True  = False
```

nicht uniform. Es wäre aber uniform bei Pattern matching von rechts nach links.

Es sind auch andere Pattern-Matching-Strategien denkbar, aber viele funktionale Sprachen basieren auf links→rechts Pattern Matching.

Ausweg: Nicht-uniforme Definitionen verbieten? Manchmal sind sie aber ganz praktisch:

Beispiel: Umkehrung einer 2-elementigen Liste (und nur diese!):

```
rev2 [x,y] = [y,x]
rev2 xs    = xs
```

Als uniforme Definition:

```
rev2  []         = []
rev2  [x]        = [x]
rev2  [x,y]      = [y,x]
rev2  (x:y:z:xs) = x:y:z:xs
```

Andere Alternativen: nicht-sequentielle Auswertung, d.h. statt leftmost outermost könnte man auch eine „parallel outermost“-Strategie verwenden. Diese wäre zwar vollständig (d.h. läuft nicht in Endlosschleifen, obwohl Werte existieren), sie ist aber aufwändig zu implementieren.