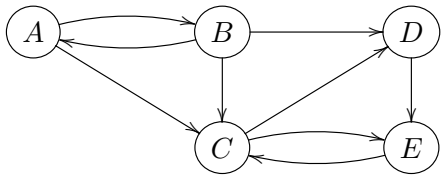
Eine Motivation zur eingekapselten Suche war der Wunsch, innerhalb eines Programms verschiedene Lösungen zu vergleichen, um z.B. die beste auszuwählen. Wir wollen nun an einem Beispiel zeigen, wie wir dies mit Mengenfunktionen realisieren können. Als Beispiel betrachten wir das **Suchen eines kürzesten Weges** in einem Graphen. Unser Beispielgraph hat die folgende Struktur:



Um diese Struktur in einem Programm zu modellieren, definieren wir zunächst einen Datentyp zur Darstellung der Knoten:

```
data Node = A | B | C | D | E
 deriving (Eq,Show)
```

Zur Darstellung der Kanten gibt es verschiedene Möglichkeiten. In einer logisch-funktionalen Sprache bietet sich die Darstellung als nichtdeterministische Operation an, d.h. wir definieren eine Operation `succ`, die einen Nachfolger zu einem Knoten liefert:

```
succ :: Node → Node
succ A = B
succ A = C
succ B = A
succ B = C
succ B = D
succ C = D
succ C = E
succ D = E
succ E = C
```

Nun definieren wir eine Operation, die uns (nichtdeterministisch) einen Pfad zwischen zwei Knoten liefert. Dies kann z.B. dadurch erfolgen, dass wir einen Pfad vom Ausgangsknoten schrittweise erweitern, bis wir den Zielknoten erreicht haben:

```
pathFromTo :: Node → Node → [Node]
pathFromTo n1 n2 = extendPath n2 [n1]

extendPath :: Node → [Node] → [Node]
extendPath target (n:p) =
  if target==n then reverse (n:p)
               else extendPath target ((succ n) : (n:p))
```

Dieses Programm hat allerdings das Problem, dass es bei zyklischen Graphen (so wie in unserem Beispiel) in eine Endlosschleife gerät. Wir können dies vermeiden, in dem

wir nur zyklenfreie Pfade erzeugen. Hierzu gibt es eine Programmiertechnik, die auch als **Constrained Constructor** bezeichnet wird [Antoy/Hanus 02]. Die Idee ist hierbei, einen Datenkonstruktor mit mehr „Intelligenz“ auszustatten, indem dieser überprüft, ob bestimmte Konstruktionsbedingungen erfüllt sind und, falls diese nicht erfüllt sind, einfach fehlschlägt anstatt die Daten zu konstruieren. Diese Programmiertechnik wird durch die funktional-logische Programmierung ermöglicht, weil hier mit Fehlschlägen gerechnet werden kann.
In unserem konkreten Beispiel tauschen wir den Listenkonstruktor “`:`” einfach gegen einen Konstruktor `consNoCycle` aus, der nur zyklenfreie Pfade konstruiert und ansonsten fehlschlägt:

```
extendPath :: Node → [Node] → [Node]
extendPath target (n:p) =
  if target == n then reverse (n:p)
                 else extendPath target (consNoCycle (succ n) (n:p))

consNoCycle :: Node → [Node] → [Node]
consNoCycle n p | all (n /=) p
                = n:p
```

Mittels Mengenfunktionen können wir nun einfach einen kürzesten Pfad konstruieren:

```
shortestPath :: Node → Node → [Node]
shortestPath n1 n2 = minValueBy shorter ((set2 pathFromTo) n1 n2)
 where
  shorter p1 p2 = compare (length p1) (length p2)
```

Damit liefert der Aufruf “`shortestPath A E`” den Wert `[A,C,E]`.

Als weiteres Beispiel betrachten wir das **8-Damen-Problem**. Es sollen 8 Damen auf einem Schachbrett so angeordnet werden, dass sie sich gegenseitig nicht schlagen können. Da eine Dame horizontal und vertikal schlagen kann, ist es klar, dass alle horizontalen und alle vertikalen Positionen der Damen verschieden sein müssen. Somit können wir annehmen, dass die n. Dame in Zeile n steht und uns auf die Frage konzentrieren, in welcher Spalte die n. Dame steht. Da alle Spalten der Damen verschieden sein müssen, müssen also die Spaltenpositionen eine Permutation der Reihenpositionen sein. Die Definition einer Permutation übernehmen wir von früher:

```
perm []     = []
perm (x:xs) = insert x (perm xs)
 where
  insert z ys     = z : ys
  insert z (y:ys) = y : insert z ys
```

Eine Permutation ist allerdings nur eine Lösung, wenn jede Dame eine andere nicht diagonal schlagen kann. Dies können wir einfach dadurch ausdrücken, indem wir definieren,

was eine „unsichere“ Positionspermutation ist: dies ist eine, bei der die Spaltenpositionen den gleichen Abstand wie die Zeilenpositionen haben:

```
unsafe xs | xs =:= _++[x]++ys++[z]++_
          = abs (x-z) =:= length ys + 1
  where x,ys,z free
```

Hierbei bestimmt also die Länge des „Zwischenraums“ `ys` den Abstand der Damen. Damit ist also eine Permutation eine Lösung, wenn sich keine Damen schlagen können, d.h. wenn es keine Lösung für `unsafe` gibt. Diese Negation (in der logischen Programmierung auch als “negation as failure” bekannt) kann mittels Mengenfunktionen und einem Leerheitstest formuliert werden:

```
queens n | isEmpty ((set1 unsafe) p) = p
 where p = perm [1..n]
```

Man beachte hierbei, dass die Definition der Permutation mittels `where` wichtig ist, denn es soll ja die ausgegebene Permutation genau die sein, die auch als sicher getestet wurde. Ebenso ist die schwache Einkapselung hier unbedingt notwendig, denn die verschiedenen Permutationen sollen gerade nicht in `unsafe` eingekapselt werden. Damit erhalten wir z.B. die folgenden Ergebnisse:

```
> queens 4
[3,1,4,2]
[2,4,1,3]
> queens 8
[8,4,1,3,6,2,7,5]
[8,3,1,6,2,5,7,4]
...
```

4.7.3 Funktionale Muster

Die Konzepte logisch-funktionaler Programmiersprachen ermöglichen eine interessante Erweiterung des üblichen Pattern Matching, die zu mächtigen Programmregeln führt. Dies wollen wir in diesem Kapitel erläutern.

Betrachten wir zunächst noch einmal die logisch-funktionale Definition des letzten Elementes einer Liste:

```
last :: [a] → a
last xs | _++[x] =:= xs
        = x
   where x free
```

Hierbei steht “`=:=`” für *strikte* Gleichheit, d.h. die Bedingung ist erfüllt, wenn beide Seiten zu identischen Datentermen ausgewertet werden können. Dies bedeutet, dass das Argu-

ment `xs` *vollständig* ausgewertet wird, obwohl wir ja eigentlich nur das letzte Element der Liste berechnen wollen. Dies kann nicht nur zu Effizienzproblemen führen, sondern es kann auch dazu führen, dass das letzte Element nicht berechnet werden kann, falls die Berechnung anderer Listenelemente fehlschlägt. Zum Beispiel kann mit dieser Definition der Ausdruck

```
last [failed,2]
```

nicht zu `2` ausgewertet werden.

Intuitiv wertet diese Version von `last` das Argument weiter aus als eigentlich notwendig ist. Wir könnten aber `last` auch als rein funktionales Programm definieren:

```
last :: [a]  → a
last [x]      = x
last (x:y:ys) = last (y:ys)
```

Dieses Programm hat die obigen Probleme nicht und ist auch recht kurz, aber die Tatsache, dass hierdurch wirklich immer das letzte Element einer Liste berechnet wird, ist nicht so offensichtlich. Man müsste hierzu noch beweisen, dass für alle Listen `xs` und Werte `x` die Eigenschaft

```
last (xs++[x]) →* x
```

gilt. Anstatt nun diese Eigenschaft zu beweisen, kann man mittels *funktionaler Muster* dies als Programmregel zur Definition von `last` benutzen:

```
last :: [a]  → a
last (xs++[x]) = x
```

Man beachte, dass in allgemeinen Termersetzungssystemen eine solche Regel erlaubt ist, allerdings nicht in üblichen funktionalen Programmen, da das Argument der linken Seite eine definierte Funktion enthält (die Definition ist also nicht konstruktorbasiert). Ein **funktionales Muster** (**functional pattern**) ist somit ein Muster, das neben Variablen und Konstruktoren auch definierte Funktionen enthält.[11]

Funktionale Muster führen nicht nur zu gut lesbaren Programmen (es sind eher „ausführbare Spezifikationen“) sondern auch zu einem besseren Programmverhalten (sie sind weniger strikt), wie wir noch sehen werden. Was bedeuten aber diese Muster und wie werden diese abgearbeitet?

Zur Vermeidung der Definition einer neuen Programmlogik für logisch-funktionale Programme mit funktionalen Muster wird in [Antoy/Hanus 05] vorgeschlagen, solche Programme in Standardprogramme ohne funktionale Muster zu transformieren. Hierzu wird ein funktionales Muster interpretiert als Abkürzung für die Menge aller Konstruktorterme, die aus diesem Muster ableitbar sind (wobei „ableitbar“ Auswertung für logisch-

[11] Funktionale Muster werden in den Curry-Implementierungen PAKCS und KiCS2 unterstützt.

funktionale Programme bedeutet). Zum Beispiel kann man mittels Narrowing das Muster `xs++[x]` wie folgt ableiten:

```
xs++[x]  ⇝{xs↦[]}       [x]
xs++[x]  ⇝{xs↦[x1]}     [x1,x]
xs++[x]  ⇝{xs↦[x1,x2]}  [x1,x2,x]
...
```

Somit steht das Muster `xs++[x]` also für die unendlich vielen Muster `[x]`, `[x1,x]`, `[x1,x2,x]`,... und die Regel

```
last (xs++[x]) = x
```

steht damit für die unendlich vielen Regeln

```
last [x] = x
last [x1,x] = x
last [x1,x2,x] = x
...
```

Wir können zwar die Transformation nicht explizit, d.h. zur Übersetzungszeit durchführen, da wir ein unendlich großes Programm erhalten würden, allerdings kann diese Transformation auch zur Laufzeit des Programms realisiert werden, indem die Muster dann transformiert werden, wenn es notwendig ist.

Ein Vorteil dieser Transformation ist nun offensichtlich: Das Argument von `last` muss nun nicht mehr vollständig ausgewertet werden, sondern `xs` und `x` repräsentieren Mustervariablen, die auch an unausgewertete Ausdrücke gebunden werden können (im Gegensatz zu logischen Variablen!). So kann nun der Ausdruck

```
last [failed,2]
```

mittels der zweiten transformierten Regel zu `2` ausgewertet werden.