Skript zur Vorlesung

Fortgeschrittene Programmierung

SS 2013

Prof. Dr. Michael Hanus
Priv.Doz. Dr. Frank Huch

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Version vom 19. Juli 2013

Vorwort

In dieser Vorlesung werden forgeschrittene Programmierkonzepte, die über die in den ersten Studiensemestern erlernte Programmierung hinausgehen, vorgestellt. Dabei wird anhand verschiedener Programmiersprachen der Umgang mit den Konzepten der wichtigsten Programmierparadigmen vermittelt. Moderne funktionale Programmierungstechniken werden am Beispiel der Sprache Haskell gezeigt. Logische und Constraint-orientierte Programmierung wird in der Sprache Prolog vermittelt. Konzepte zur nebenläufigen und verteilten Programmierung werden mit der Sprache Java vorgestellt und geübt.

Dieses Skript basiert auf der Vorlesung vom SS 2012 und ist eine überarbeitete Fassung einer Mitschrift, die ursprünglich von Nick Prühs im SS 2009 in LaTeX gesetzt wurde. Ich danke Nick Prühs für die erste LaTeX-Vorlage und Björn Peemöller und Lars Noelle für Korrekturhinweise.

Noch eine wichtige Anmerkung: Dieses Skript soll nur einen Überblick über das geben, was in der Vorlesung gemacht wird. Es ersetzt nicht die Teilnahme an der Vorlesung, die zum Verständnis der Konzepte und Techniken der fortgeschrittenen Programmierung wichtig ist. Ebenso wird für ein vertieftes Selbststudium empfohlen, sich die in der Vorlesung angegebenen Lehrbücher und Verweise anzuschauen.

Kiel, Juni 2013 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per E-Mail mitgeteilt werden.

Inhaltsverzeichnis

1 Funktionale Programmierung

Die funktionale Programmierung bietet im Vergleich zu klassischer imperativer Programmierung eine Reihe von Vorteilen:

- hohes Abstraktionsniveau, keine Manipulation von Speicherzellen
- keine Seiteneffekte, deshalb leichtere Codeoptimierung und bessere Verständlichkeit
- Programmierung über Eigenschaften, nicht über zeitlichen Ablauf
- implizite Speicherverwaltung
- einfachere Korrektheitsbeweise, Verifikation
- kompakte Quellprogramme, deshalb kürzere Entwicklungszeit, lesbarere Programme, bessere Wartbarkeit
- modularer Programmaufbau, Polymorphismus, Funktionen höherer Ordnung, Wiederverwendbarkeit von Code

In funktionalen Programmen stellt eine *Variable* einen unbekannten Wert dar. Ein *Programm* ist Menge von Funktionsdefinitionen. Der Speicher ist nicht explizit verwendbar, sondern wird automatisch verwaltet und aufgeräumt. Ein *Programmablauf* besteht aus der Reduktion von Ausdrücken. Dies geht auf die mathematische Theorie des λ-Kalküls von Church [1] zurück. Im folgenden führen wir die rein funktionale Programmierung anhand der Programmiersprache Haskell [7] ein.

1.1 Funktions- und Typdefinitionen

In der Mathematik steht eine Variable für unbekannte (beliebige) Werte, so dass wir dort oft mit Ausdrücken wie

$$x^2 - 4x + 4 = 0 \Leftrightarrow x = 2$$

arbeiten.

In imperativen Sprachen sehen wir hingegen häufig Ausdrücke wie

$$x = x + 1$$

welche einen Widerspruch zur Mathematik darstellen. In der funktionalen Programmierung werden, wie in der Mathematik, Variablen als unbekannte Werte (und nicht als Namen für Speicherzellen) interpretiert!

Während Funktionen in der Mathematik zur Berechnung dienen, verwenden wir Prozeduren oder Funktionen in Programmiersprachen zur Strukturierung. Dort ergibt sich aber

wegen Seiteneffekten kein wirklicher Zusammenhang. In funktionalen Programmiersprachen gibt es jedoch keine Seiteneffekte, somit liefert jeder Funktionsaufruf mit gleichen Argumenten das gleiche Ergebnis.

Funktionen können in Haskell folgendermaßen definiert werden:

```
f x1 ... xn = e
```

Dabei ist `f` der Funktionsname, `x1` bis `xn` sind formale Parameter bzw. Variablen, und `e` ist der Rumpf, ein Ausdruck über `x1` bis `xn`.

Ausdrücke können in Haskell wie folgt gebildet werden:

1. Zahlen: `3`, `3.14159`
2. Basisoperationen: `3 + 4`, `5 * 7`
3. Funktionsanwendungen: `(f e1 ... en)`. Die Außenklammerung kann entfallen, falls dies aus dem Zusammenhang klar ist.
4. Bedingte Ausdrücke: `(if b then e1 else e2)`

Nehmen wir an, die Quadratfunktion

```
square x = x * x
```

sei in einer Datei `square.hs` gespeichert. Dann kann man die Interpreter `hugs` oder `ghci`, welche sich im Verzeichnis `~haskell/bin/` befinden, verwenden wie folgt:

```
> ~haskell/bin/ghci
GHCi, version 7.4.2: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l square
[1 of 1] Compiling Main                ( square.hs, interpreted )
Ok, modules loaded: Main.
*Main> square 3
9
*Main> square (3 + 1)
16
*Main> :q
Leaving GHCi.
```

Eine Funktion zur Berechnung des Minimums zweier Zahlen kann in Haskell so aussehen:

```
min x y = if x <= y then x else y
```

Als nächstes betrachten wir die Fakultätsfunktion. Diese ist mathematisch wie folgt definiert:

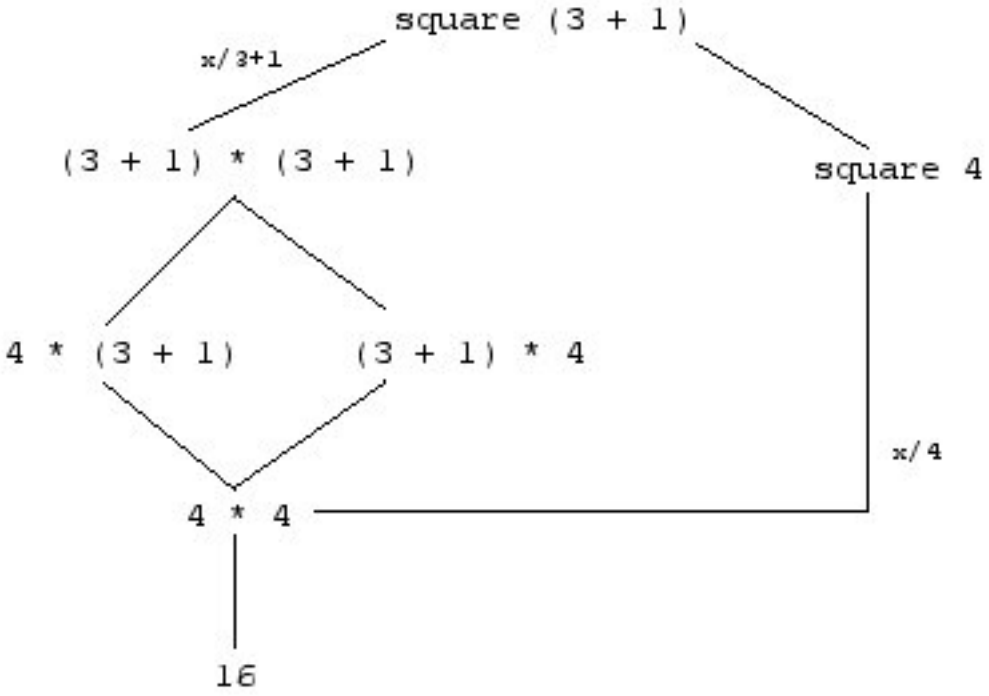


Abbildung 1.1: Mögliche Auswertungen von Funktionen

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot (n-1)!, & \text{sonst} \end{cases}$$

In Haskell setzen wir diese Funktion so um:

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

1.1.1 Auswertung

Die Auswertung von Funktionsdefinitionen in Haskell erfolgt durch orientierte Berechnung von links nach rechts: Zuerst werden die aktuellen Parameter gebunden, also die formalen Parameter durch die aktuellen ersetzt. Dann wird die linke durch die rechte Seite ersetzt.

Abbildung 1.1 zeigt, auf welche Art und Weise Funktionen ausgewertet werden können. Der rechte Ast zeigt, wie eine Funktion in Java auswertet wird, der linke ähnelt der Auswertung in Haskell, wenn man doppelte Berechnungen wie die von `3 + 1` weglässt.

Als weiteres Beispiel folgt die Auswertung eines Aufrufs unserer Funktion `fac`:

```
fac 2  =  if 2 == 0 then 1 else 2 * fac (2 - 1)                (1.1)
       =  if False then 1 else 2 * fac (2 - 1)                 (1.2)
       =  2 * fac (2 - 1)                                      (1.3)
       =  2 * fac 1                                            (1.4)
       =  2 * (if 1 == 0 then 1 else 1 * fac (1 - 1))          (1.5)
       =  2 * (if False then 1 else 1 * fac (1 - 1))           (1.6)
       =  2 * 1 * fac (1 - 1)                                  (1.7)
       =  2 * 1 * fac 0                                        (1.8)
       =  2 * 1 * (if 0 == 0 then 1 else 0 * fac (0 - 1))      (1.9)
       =  2 * 1 * (if True then 1 else 0 * fac (0 - 1))        (1.10)
       =  2 * 1 * 1                                            (1.11)
       =  2 * 1                                                (1.12)
       =  2                                                    (1.13)
```

Wir möchten nun eine effiziente Funktion zur Berechnung von Fibonacci-Zahlen entwickeln. Unsere erste Variante ist direkt durch die mathematische Definition motiviert:

```
fib1 n = if n == 0
         then 0
         else if n == 1
              then 1
              else fib1 (n - 1) + fib1 (n - 2)
```

Diese Variante ist aber äußerst ineffizient: Ihre Laufzeit liegt in $O(2^n)$.

Wie können wir unsere erste Variante verbessern? Wir berechnen die Fibonacci-Zahlen von unten: Die Zahlen werden von 0 an aufgezählt, bis die n-te Zahl erreicht ist: 0 1 1 2 3 ... $fib(n)$.

Diese Programmiertechnik ist bekannt unter dem Namen *Akkumulatortechnik*. Hierzu müssen wir die beiden vorigen Zahlen stets als Parameter mitführen:

```
fib2' fibn fibnp1 n = if n == 0
                     then fibn
                     else fib2' fibnp1 (fibn + fibnp1) (n - 1)

fib2 n = fib2' 0 1 n
```

Dabei ist `fibn` ist die n-te Fibonacci-Zahl, `fibnp1` die $(n+1)$-te. Dadurch erreichen wir eine lineare Laufzeit.

Aus softwaretechnischer Sicht ist unsere zweite Variante aber unschön: Wir wollen natürlich andere Aufrufe von `fib2'` von außen vermeiden. Wie das funktioniert ist im nächsten

Abschnitt beschrieben.

1.1.2 Lokale Definitionen

Haskell bietet mehrere Möglichkeiten, Funktionen lokal zu definieren. Eine Möglichkeit stellt das Schlüsselwort `where` dar:

```
fib2 n = fib2' 0 1 n
  where fib2' fibn fibnp1 n =
          if n == 0
          then fibn
          else fib2' fibnp1 (fibn + fibnp1) (n - 1)
```

`where`-Definitionen sind in der vorhergehenden Gleichung sichtbar, außerhalb sind sie unsichtbar.
Alternativ können wir auch das Schlüsselwort `let` verwenden:

```
fib2 n =
  let fib2' fibn fibnp1 n =
        if n == 0
        then fibn
        else fib2' fibnp1 (fibn + fibnp1) (n - 1)
  in fib2' 0 1 n
```

Dabei ist `let` im Gegensatz zu `where` ein *Ausdruck*. Das durch `let` definierte `fib2'` ist nur innerhalb des `let`-Ausdrucks sichtbar.
`let` ... `in` ... kann als beliebiger Ausdruck auftreten: So wird

```
(let x = 3
     y = 1
 in x + y) + 2
```

ausgewertet zu 6.
Die Syntax von Haskell verlangt bei der Definition solcher Blöcke keine Klammerung und keine Trennung der einzelnen Definitionen (z.B. durch ein Semikolon). In Haskell gilt die *Layout-Regel (off-side rule)*: Das nächste Symbol hinter `where` oder `let`, das kein Whitespace ist, definiert einen *Block*:

- Beginnt nun die Folgezeile rechts vom Block, so gehört sie zur gleichen Definition.
- Beginnt die Folgezeile am Blockrand, so beginnt hier eine neue Definition im Block.
- Beginnt die Folgezeile aber links vom Block, so wird der Block davor hier beendet.

Lokale Definitionen bieten eine Reihe von Vorteilen:

- Namenskonflikte können vermieden werden
- falsche Benutzung von Hilfsfunktionen kann vermieden werden

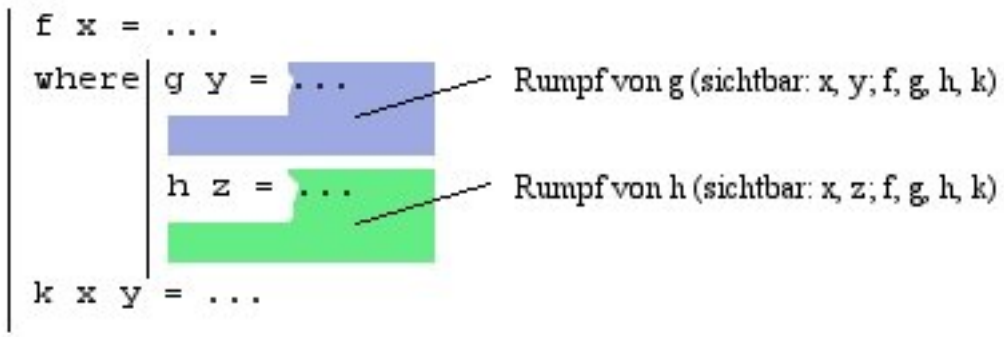


Abbildung 1.2: Layout-Regel in Haskell

- bessere Lesbarkeit
- Mehrfachberechnungen können vermieden werden
- weniger Parameter bei Hilfsfunktionen

Betrachten wir ein Beispiel zur Vermeidung von Mehrfachberechnungen: Statt der unübersichtlichen Funktionsdefinition

```
f x y = y * (1 - y) + (1 + x * y) * (1 - y) + x * y
```

schreiben wir besser

```
f x y = let a = 1 - y
            b = x * y
        in y * a + (1 + b) * a + b
```

Durch lokale Deklarationen mittels `let` und `where` ist es auch möglich, Parameter bei Hilfsfunktionen zu sparen. Dies wird durch folgendes Beispiel deutlich:

Das Prädikat `isPrim` soll überprüfen, ob es sich bei der übergebenen Zahl um eine Primzahl handelt. In Haskell können wir dies wie folgt ausdrücken:

```
isPrim n = n /= 1 && checkDiv (div n 2)
  where checkDiv m =
          m == 1 || mod n m /= 0 && checkDiv (m - 1)
```

Hier drückt `/=` Ungleichheit aus, `&&` steht für eine Konjunktion, `||` für das logische Oder und `div` für die ganzzahlige Division.

In der letzten Zeile brauchen wir keine weitere Klammerung, da `&&` stärker bindet als `||`. Das `n` ist in `checkDiv` sichtbar, weil letzteres lokal definiert ist.

1.2 Datentypen

1.2.1 Basisdatentypen

Ganze Zahlen

Einen Basisdatentyp von Haskell haben wir oben bereits verwendet: ganze Zahlen. Tatsächlich werden in Haskell zwei Arten ganzer Zahlen unterschieden:

`Int`: Werte -2^{31} ... $2^{31} - 1$
`Integer`: beliebig groß (nur durch Speicher begrenzt)

Operationen: `+ - * div mod`
Vergleiche: `< > <= >= == /=`

Boolesche Werte

Ein weiterer Basisdatentyp sind die booleschen Werte:

`Bool`: `True False`

Operationen: `&& || not == /=`

Hier steht == für äquivalent, /= steht für das ausschließende Oder (XOR).

Gleitkommazahlen

Gleitkommazahlen sind ebenfalls ein Basisdatentyp:

`Float`: `0.3 -1.5e-2`

Operationen: wie `Int`, aber `/` statt `div`, kein `mod`

Zeichen

Und auch Unicode-Zeichen sind ein Haskell-Basisdatentyp:

`Char`: `'a' '\n' '\NUL' '\214'`

Operationen (diese stehen in der Bibliothek `Data.Char`, die man mittels der Angabe von "`import Data.Char`" zu Beginn des Haskell-Programms hinzuladen kann):

```
chr :: Int  → Char
ord :: Char  → Int
```

Mit "`::`" werden in Haskell optionale Typannotationen beschrieben, wie nachfolgend weiter erläutert wird.

1.2.2 Typannotationen

Haskell ist eine streng typisierte Programmiersprache, d.h. alle Werte und Ausdrücke in Haskell haben einen Typ, welcher auch annotiert werden kann:

```
3 :: Int
```

In diesem Beispiel ist `3` ein Wert bzw. Ausdruck, und `Int` ist ein Typausdruck. Weitere Beispiele für Typannotationen sind:

```
3 :: Integer
(3 == 4) || True :: Bool
(3 == (4 :: Int)) || True :: Bool
```

Wir können auch Typannotationen für Funktionen angeben. Diese werden in eine separate Zeile geschrieben:

```
square :: Int  → Int
square x = x * x
```

Aber was ist der Typ von `min` (siehe Abschnitt 1.1), der zwei Argumente hat? Dieser wird folgendermaßen aufgeschrieben:

```
min :: Int  → Int  → Int
```

Auch zwischen den Argumenttypen wird also ein Funktionspfeil geschrieben. Später werden wir noch sehen, warum dies so ist.

1.2.3 Algebraische Datenstrukturen

Eigene Datenstrukturen können als neue Datentypen definiert werden. Werte werden mittels *Konstruktoren* aufgebaut. Konstruktoren sind frei interpretierte Funktionen, und damit nicht reduzierbar.

Definition eines algebraischen Datentyps

Algebraische Datentypen werden in Haskell wie folgt definiert:

`data` τ `=` $C_1\ \tau_{11}\ \ldots\ \tau_{1n_1}$ `|` $\ldots$ `|` $C_k\ \tau_{k1}\ \ldots\ \tau_{kn_k}$

wobei

- τ der neu definierte Typ ist

- $C_1, ..., C_k$ die definierten Konstruktoren sind und
- $\tau_{i1}, ..., \tau_{in_i}$ die Argumenttypen des Konstruktors C_i sind, also

  $C_i \ :: \ \tau_{i1} \ \rightarrow \ \cdots \ \rightarrow \tau_{in_i} \ \rightarrow \tau$

  gilt.

Beachte: Sowohl Typen als auch Konstruktoren müssen in Haskell mit Großbuchstaben beginnen!

Beispiele

1. Aufzählungstyp (nur 0-stellige Konstruktoren):

   ```
   data Color = Red | Blue | Yellow
   ```

   definiert genau die drei Werte des Typs `Color`. `Bool` ist auch ein Aufzählungstyp, der durch

   ```
   data Bool = False | True
   ```

   vordefiniert ist.
2. Verbundtyp (nur ein Konstruktor):

   ```
   data Complex = Complex Float Float
   Complex 3.0 4.0 :: Complex
   ```

   Dies ist erlaubt, da Haskell mit getrennten Namensräumen für Typen und Konstruktoren arbeitet.
   Wie selektieren wir nun einzelne Komponenten? In Haskell verwenden wir *pattern matching* statt expliziter Selektionsfunktionen (diese Möglichkeit wird später noch genauer erläutert):

   ```
   addC :: Complex  → Complex  → Complex
   addC (Complex r1 i1) (Complex r2 i2) = Complex (r1+r2) (i1+i2)
   ```

3. Listen (gemischte Typen): Hier betrachten wir zunächst nur Listenelemente vom Typ `Int`:

   ```
   data List = Nil | Cons Int List
   ```

   Hier steht `Nil` für die leere Liste. Die Funktion `append` erlaubt das Zusammenhängen von Listen:

   ```
   append :: List  → List  → List
   append Nil         ys = ys
   append (Cons x xs) ys = Cons x (append xs ys)
   ```

Dies ist eine Definition mit Hilfe mehrerer Gleichungen, wobei die erste passende gewählt wird. Ein Funktionsaufruf könnte zum Beispiel so reduziert werden:

```
   append (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)
 = Cons 1 (append (Cons 2 Nil) (Cons 3 Nil))
 = Cons 1 (Cons 2 (append Nil (Cons 3 Nil)))
 = Cons 1 (Cons 2 (Cons 3 Nil))
```

In Haskell sind Listen vordefiniert mit:

```
data [Int] = [] | Int:[Int]
```

`[]` entspricht `Nil`, "`:`" entspricht `Cons` und `[Int]` entspricht `List`.
Dabei ist der Operator "`:`" rechtsassoziativ, also es gilt:

```
1:(2:(3:[]))
```

entspricht

```
1:2:3:[]
```

was auch noch wie folgt kompakt aufgeschrieben werden kann:

```
[1,2,3]
```

Außerdem bietet uns Haskell den in der `Prelude` vordefinierte Operator `++` statt `append`:

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

Operatoren sind zweistellige Funktionen, die infix geschrieben werden und mit Sonderzeichen beginnen. Durch Klammerung werden sie zu normalen Funktionen:

```
(++) :: [Int] -> [Int] -> [Int]
```

`[1] ++ [2]` entspricht `(++) [1] [2]`.

Umgekehrt können zweistellige Funktionen durch einfache Gegenanführungszeichen `‘...‘` infix verwendet werden:

```
div 4 2
```

kann auch wie folgt aufgeschrieben werden:

```
4 ‘div‘ 2
```

Für selbstdefinierte Datentypen besteht nicht automatisch die Möglichkeit, diese vergleichen oder ausgeben zu können. Hierzu kann man das Schlüsselwort `deriving` hinter der Datentypdefinition verwenden:

```
data MyType = ... deriving (Eq, Show, Ord)
```

1.3 Polymorphismus

Zur Definition universell verwendbarer Datenstrukturen und Operationen unterstützt Haskell *Typpolymorphismus*. Um dies genauer zu erläutern, wollen wir als Beispiel die Länge einer Liste bestimmen:

```
length :: [Int]  → Int
length []     = 0
length (_:xs) = 1 + length xs
```

Diese Definition funktioniert natürlich auch für andere Listen, beispielsweise vom Typ `[Char]`, `[Bool]` oder `[[Int]]`.
Allgemein könnte man also sagen:

```
length :: ∀ Type τ . [τ]  →Int
```

was in Haskell durch Typvariablen ausgedrückt wird:

```
length :: [a]  → Int
```

Was ist der Typ von `(++)`?

```
(++) :: [a]  → [a]  → [a]
```

Es können also nur Listen mit gleichem Argumenttyp konkateniert werden. Wir betrachten ein weiteres Beispiel:

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

Dies funktioniert, da `[a]` ein Listentyp ist, und `[x]` eine einelementige Liste (entspricht `x:[]`). Die beiden Regeln dürfen wir aber nicht vertauschen, sonst terminiert kein Aufruf der Funktion!
Wie können wir nun selber polymorphe Datentypen definieren? Hierzu gibt es in Haskell *Typkonstruktoren* zum Aufbau von Typen:

```
data K  a1 ... am  = C1  τ11  ...  τ1n1  |  ...  |  Ck  τk1  ...  τknk
```

Diese Datentypdefinitionen sehen also ähnlich aus wie zuvor, aber hier gilt:

- K ist ein Typkonstruktor (kein Typ)
- $a_1, \ldots, a_m$ sind Typvariablen
- τ_{ik} sind Typausdrücke, welche Basistypen, Typvariablen oder die Anwendung eines Typkonstruktors auf Typausdrücke sind.

Funktionen und Konstruktoren werden auf Werte bzw. Ausdrücke angewandt und erzeugen Werte. Analog werden Typkonstruktoren auf Typen angewandt und erzeugen Typen.

Als Beispiel für einen polymorphen Datentyp wollen wir partielle Werte in Haskell modellieren:

```
data Maybe a = Nothing | Just a
```

Dann ist "`Maybe Int`" und auch "`Maybe (Maybe Int)`" ein zulässiger Typ.
Falls in einem Typ ein Typkonstruktor auf Typvariablen angewandt wird, dann nennt man den sich ergebenden Typ auch *polymorph*, wie in dem folgenden Beispiel.

```
isNothing :: Maybe a  → Bool
isNothing Nothing  = True
isNothing (Just _) = False
```

Ein weiteres gutes Beispiel für einen polymorphen Datentyp ist der Binärbaum:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

height :: Tree a  → Int
height (Leaf _)     = 1
height (Node tl tr) = 1 + max (height tl) (height tr)
```

In Haskell sind auch polymorphe Listen als syntaktischer Zucker vordefiniert wie:

```
data [a] = [] | a : [a]
```

Diese Definition ist zwar kein syntaktisch zulässiges Haskell-Programm, sie kann aber wie folgt verstanden werden: Die eckigen Klammern um `[a]` sind der Typkonstruktor für Listen, `a` ist der Elementtyp, `[]` ist die leere Liste und `:` ist der Listenkonstruktor.
Aus diesem Grund sind die folgenden Ausdrücke alle gleich:

```
(:) 1 ((:) 2 ((:) 3 []))
1 : (2 : (3 : []))
1 : 2 : 3 : []
[1,2,3]
```

Nach obiger Definition sind also beliebig komplexe Listentypen wie `[Int]`, `[Maybe Bool]` oder `[[Int]]` möglich.
Einige Funktionen auf Listen sind beispielsweise `head`, `tail`, `last`, `concat` und `(!!)`:

```
head :: [a]  → a
head (x:_) = x

tail :: [a]  → [a]
tail (_:xs) = xs

last :: [a]  → a
last [x]    = x
last (_:xs) = last xs

concat :: [[a]]  → [a]
concat []     = []
concat (l:ls) = l ++ concat ls

(!!) :: [a]  → Int  → a
(x:xs) !! n = if n == 0 then x
                        else xs !! (n - 1)
```

Für die letzte Funktion ist auch folgende Definition möglich:

```
(x:_ ) !! 0 = x
(_:xs) !! n = xs !! (n - 1)
```

Zeichenketten sind in Haskell als Listen von Zeichen definiert:

```
type String = [Char]
```

Hierbei leitet "`type`" die Definition eines *Typsynonyms* ein, d.h. einen neuen Namen (`String`) für einen anderen Typausdruck (`[Char]`).
Damit entspricht die Zeichenkette `"Hallo"` der Liste `'H':'a':'l':'l':'o':[]`. Aus diesem Grund funktionieren alle Listenfunktionen auch für Strings: der Ausdruck

```
length ("Hallo" ++ " Leute!")
```

wird zu `11` ausgewertet.
Weitere in Haskell vordefinierte Typkonstruktoren sind:

- Vereinigung zweier Typen

  ```
  data Either a b = Left a | Right b
  ```

  Damit können zum Beispiel Werte „unterschiedlicher Typen“ in eine Liste geschrie-

ben werden:

```
[Left 42, Right "Hallo"] :: [Either Int String]
```

Vereinigungstypen können z.B. mittels Mustern verarbeitet werden, wie z.B.

```
valOrLen :: Either Int String  → Int
valOrLen (Left  v) = v
valOrLen (Right s) = length s
```

- Tupel

```
data (,) a b = (,) a b
data (,,) a b c = (,,) a b c
```

Auch hierauf sind bereits einige Funktionen definiert:

```
(3,True) :: (Int,Bool)

fst :: (a, b)  → a
fst (x, _) = x

snd :: (a, b)  → b
snd (_, y) = y

zip :: [a]  → [b]  → [(a, b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys


unzip :: [(a, b)]  → ([a], [b])
unzip []           = ([], [])
unzip ((x ,y):xys) = let (xs, ys) = unzip xys
                     in (x:xs, y:ys)
```

Im Prinzip ist `unzip` die Umkehrung von `zip`, d.h. wenn "`unzip zs`" zum Ergebnis `(xs,ys)` auswertet, dann wertet "`zip xs ys`" wiederum zu `zs` aus. Allerdings wertet umgekehrt `unzip (zip xs ys)` nicht immer zu `(xs,ys)` aus!

1.4 Pattern Matching

Wie wir schon gesehen haben, können Funktionen durch mehrere Gleichungen mittels *pattern matching* definiert werden:

`f` $pat_{11} \ldots pat_{1n}$ `=` e_1

$\vdots$

`f` $pat_{k1} \ldots pat_{kn}$ `=` e_k

Dies führt zu sehr übersichtlichen Programmen, da man die rechten Seiten nur für die speziellen Fälle, die durch die Muster spezifiziert werden, definieren muss. Bei mehreren Regeln wählt Haskell die textuell erste Regel mit passender linker Seite und wendet diese an. Überlappende Regeln sind prinzipiell erlaubt, aber sie können zu nicht klar lesbaren Programmen führen und sollten daher besser vermieden werden.

1.4.1 Aufbau der Pattern

Im Prinzip sind die Muster Datenterme (d.h. sie enthalten keine definierten Funktionen) mit Variablen. Genauer sind folgende Muster möglich:

- `x` *(Variable)*: passt immer, Variable wird an aktuellen Wert gebunden.
- `_` (*Wildcard*): passt immer, keine Bindung.
- `C` $pat_1 ... pat_k$ wobei `C` k-stelliger Konstruktor: passt, falls gleicher Konstruktor und Argumente passen auf $pat_1, ..., pat_k$
- `x@pat` (*as pattern*): passt, falls `pat` passt; zusätzlich wird `x` an den gesamten passenden Wert gebunden

  Damit können auch überlappende Pattern vermieden werden:

```
last [x]          = x
last (x:xs@(_:_)) = last xs
```

Außerdem gibt es noch sogenannte `(n+`k`)`-Pattern als Muster für positive ganze Zahlen, welche wir aber nicht erläutern, da diese oft nicht unterstützt werden und auch nicht so wichtig sind.

Pattern können auch bei `let` und `where` verwendet werden:

```
unzip ((x,y) : xys) = (x:xs, y:ys)
 where
  (xs,ys) = unzip xys
```

1.4.2 Case-Ausdrücke

Manchmal ist es auch praktisch, mittels pattern matching in Ausdrücke zu verzweigen: So definiert

`case` e `of` $pat_1 \rightarrow e_1$
$\vdots$
$pat_n \rightarrow e_n$

einen Ausdruck mit Typ von $e_1, ..., e_n$, welche alle den gleichen Typ haben müssen. $e, pat_1, ..., pat_n$ müssen ebenfalls den gleichen Typ haben. Nach dem Schlüsselwort `of` gilt auch die Layout-Regel, d.h. die Muster $pat_1, ..., pat_n$ müssen alle in der gleichen Spalte beginnen.

Das Ergebnis von `e` wird hier der Reihe nach gegen pat_1 bis pat_n gematcht. Falls ein Pattern passt, wird der ganze `case`-Ausdruck durch das zugehörige e_i ersetzt.

Als Beispiel betrachten wir das Extrahieren der Zeilen eines Strings:

```
lines :: String  → [String]
lines ""        = []
lines ('\n':cs) = "" : lines cs
lines (c:cs)    = case lines cs of
                    []     → [[c]]
                    (l:ls) → (c : l) : ls
```

1.4.3 Guards

Jedes pattern matching kann eine zusätzliche boolesche Bedingung bekommen, welche *Guard* genannt wird:

```
fac n | n == 0    = 1
      | otherwise = n * fac (n - 1)
```

Durch Kombination von Guards and case-Ausdrücke kann man beispielsweise die ersten n Elemente einer Liste extrahieren:

```
take :: Int  → [a]  → [a]
take n xs | n <= 0    = []
          | otherwise = case xs of
                          []     → []
                          (x:xs) → x : take (n-1) xs
```

Guards sind auch bei `let`, `where` und `case` erlaubt.

1.5 Funktionen höherer Ordnung

Funktionen sind Bürger erster Klasse: Sie können wie alle anderen Werte verwendet werden. Anwendungen davon sind:

- generische Programmierung
- Programmschemata (Kontrollstrukturen)

Wir erreichen damit eine bessere Wiederverwendbarkeit und eine höhere Modularität des Codes.

1.5.1 Beispiel: Ableitungsfunktion

Die Ableitungsfunktion ist eine Funktion, die zu einer Funktion eine neue Funktion liefert. Die numerische Berechnung sieht so aus:

$$f'(x) = \lim_{dx \to 0} \frac{f(x+dx)-f(x)}{dx}$$

Eine Implementierung mit kleinem dx könnte in Haskell so aussehen:

```
dx = 0.0001

derive :: (Float → Float) → (Float → Float)
derive f = f'
 where f' :: Float → Float
       f' x = (f (x + dx) - f x) / dx
```

Nun wird `(derive sin) 0.0` ausgewertet zu `1.0`, `(derive square) 1.0` wird ausgewertet zu `2.00033`.

1.5.2 Anonyme Funktionen (Lambda-Abstraktionen)

Manchmal möchte man nicht jeder Funktion mit einem Namen definieren (wie `square`), sondern diese auch direkt da schreiben, wo man sie benötigt, wie z.B.

```
derive (\x → x * x)
```

Das Argument entspricht der Funktion $x \mapsto 2x$. Eine solche Funktion ohne Namen wird auch als *Lambda-Abstraktion* oder *anonyme Funktion* bezeichnet. Hierbei steht \ für λ, `x` ist ein Parameter und `x * x` ein Ausdruck (der Rumpf der Funktion).
Allgemein formuliert man anonyme Funktionen in Haskell wie folgt:

\ p_1 ... p_n → e

wobei $p_1, \ldots, p_n$ Pattern sind und e ein Ausdruck ist.
Dann können wir auch schreiben:

```
derive f = \x → (f (x + dx) - f x) / dx
```

Aber `derive` ist nicht die einzige Funktion mit funktionalem Ergebnis: So kann man zum Beispiel die Funktion `add` auf drei verschiedene Weisen definieren:

```
add :: Int → Int → Int
add x y = x + y
```

oder

```
add = \x y  → x + y
```

oder

```
add x = \y  → x + y
```

Also kann `add` auch als Konstante gesehen werden, die eine Funktion als Ergebnis liefert, oder als Funktion, die einen `Int` nimmt und eine Funktion liefert, die einen weiteren `Int` nimmt und erst dann einen `Int` liefert.

Somit müssen die Typen `Int -> Int -> Int` und `Int -> (Int -> Int)` identisch sein. Die Klammerung ergibt sich durch rechtsassoziative Bindung des Typkonstruktors `(->)`. Beachte aber, dass `(a -> b) -> c` *nicht* das gleiche ist wie `a -> b -> c` oder `a -> (b -> c)`!

Es wäre also unabhängig von der Definition von `add` sinnvoll, folgendes zu schreiben:

```
derive (add 2)
```

Wird eine Funktion auf „zu wenige“ Argumente appliziert, nennt man dies *partielle Applikation*Applikation, partielle. Die partielle Applikation wird syntaktisch einfach möglich durch Currying. *Currying* geht auf *Haskell B. Curry* und *Schönfinkel* zurück, die in den 40er Jahren unabhängig voneinander folgende Isomorphie festgestellt haben:

$$[A \times B \to C] \simeq [A \to (B \to C)]$$

Mit Hilfe von partieller Applikation lassen sich nun eine Reihe von Funktionen definieren:

- `take 42 :: [a] -> [a]` liefert die bis zu 42 ersten Elemente einer Liste.
- `(+) 1 :: Int -> Int` ist die Inkrementfunktion.

Bei Operatoren bieten die sog. *Sections* eine zusätzliche, verkürzte Schreibweise:

- `(1+)` steht für die Inkrementfunktion.
- `(2-)` steht für `\x -> 2-x`.
- `(/2)` steht für `\x -> x/2`.
- `(-2)` steht aber *nicht* für `\x -> x-2`, da der Compiler hier das Minus-Zeichen nicht vom unären Minus unterscheiden kann.

Somit ist bei Operatoren auch eine partielle Applikation auf das zweite Argument möglich:

```
(/b) a = (a/) b = a / b
```

Die Reihenfolge der Argumente ist wegen partieller Applikation also eine Designentscheidung, aber mit λ-Abstraktion und der Funktion `flip` bei der partiellen Anwendung in anderer Reihenfolge noch veränderbar.

1.5.3 Generische Programmierung

Wir betrachten die folgenden Funktionen `incList` und `codeStr`:

```
incList :: [Int]  → [Int]
incList []     = []
incList (x:xs) = (x + 1) : incList xs

code :: Char  → Char
code c | c == 'Z'  = 'A'
       | c == 'z'  = 'a'
       | otherwise = chr (ord c + 1)

codeStr :: String  → String
codeStr ""     = ""
codeStr (c:cs) = code c : codeStr cs
```

Dann wird `codeStr "Informatik"` ausgewertet zu `"Jogpsnbujl"`. Wir können beobachten, dass es sich bei `incList` und `codeStr` um fast identische Definitionen handelt, die sich nur in der Funktion unterscheiden, die auf die Listenelemente angewandt wird.
Die Verallgemeinerung ist die Funktion `map`:

```
map :: (a  → b)  → [a]  → [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Damit lassen sich `incList` und `codeStr` viel einfacher ausdrücken:

```
incList = map (+1)
codeStr = map code
```

Wir betrachten zwei weitere Beispiele: Eine Funktion, die die Summe aller Zahlen in einer Liste liefert, und eine Funktion, die zur Eingabekontrolle die Summe der Unicode-Werte einer Zeichenkette berechnet:

```
sum :: [Int]  → Int
sum []     = 0
sum (x:xs) = x + sum xs

checkSum :: String  → Int
checkSum ""     = 1
checkSum (c:cs) = ord c + checkSum cs
```

Findet sich hier ebenfalls ein gemeinsames Muster? Ja! Beide Funktionen lassen sich viel einfacher mit der mächtigen Funktion `foldr` ausdrücken:

```
foldr :: (a → b → b) → b → [a] → b
foldr _ e []     = e
foldr f e (x:xs) = f x (foldr f e xs)

sum = foldr (+) 0
checkSum = foldr (\c res → ord c + res) 1
```

Zum Verständnis von `foldr` hilft die folgende Sichtweise: Die an `foldr` übergebene Funktion vom Typ `(a -> b -> b)` wird als Ersatz für den Listenkonstruktor `(:)` in der Liste eingesetzt, und das übergebene Element vom Typ `b` als Ersatz für die leere Liste `[]`. Man beachte: Der Typ der übergebenen Funktionen passt zum Typ von `(:)`.

So entsprechen sich also die folgenden Ausdrücke:

```
  foldr f e [1,2,3]
= foldr f e ((:) 1 ((:) 2 ((:) 3 [])))
=           (f   1 (f   2 (f   3 e )))
```

Das allgemeine Vorgehen beim Entwerfen solcher Funktionen ist das Folgende: Suche ein allgemeines Schema und realisiere es durch funktionale Parameter.

Ein weiteres Schema kennen wir ist die Funktion `filter`, die Elemente mit einer bestimmten Eigenschaft aus einer Liste filtert:

```
filter :: (a → Bool) → [a] → [a]
filter _ []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

Diese können wir zum Beispiel verwenden, um eine Liste in eine Menge umzuwandeln, also um alle doppelten Einträge zu entfernen:

```
nub :: [Int] → [Int]
nub []     = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Mit Hilfe von `filter` können wir sogar Listen mittels *Quicksort* sortieren:

```
qsort :: [Int] → [Int]
qsort []     = []
qsort (x:xs) =
  qsort (filter (<= x) xs) ++ [x] ++ qsort (filter (> x) xs)
```

Auch `filter` kann mit Hilfe von foldr definiert werden:

```
filter p = foldr (\x ys → if p x then x:ys else ys) []
```

In der Tat ist `foldr` ein sehr allgemeines Skelett, es entspricht dem Katamorphismus der Kategorientheorie.
`foldr` hat manchmal aber auch Nachteile: So führt

```
foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))
```

zu einer sehr großen Berechnung, die zunächst auf dem Stack aufgebaut und erst zum Schluss ausgerechnet wird.
Eine bessere Lösung finden wir mit Hilfe der Akkumulatortechnik:

```
sum xs = sum' xs 0
  where sum' :: [Int]  → Int  → Int
        sum' []     s = s
        sum' (x:xs) s = sum' xs (x + s)
```

Damit wird ein Aufruf von `sum [1,2,3]` ersetzt durch `((0 + 1) + 2) + 3`, was direkt ausgerechnet werden kann.
Aber auch das ist als fold-Variante möglich:

```
foldl :: (a  → b  → a)  → a  → [b]  → a
foldl _ e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

Damit wird ein Aufruf von `foldl f e (`x_1 `:` x_2 `: ... :` x_n `: [])` ersetzt durch `f ... (f (f e` x_1`)` x_2`) ...` x_n
Jetzt können wir `sum` natürlich wieder viel einfacher definieren:

```
sum = foldl (+) 0
```

1.5.4 Kontrollstrukturen

Viele der Kontrollstrukturen, die wir aus anderen Programmiersprachen kennen, lassen sich auch in Haskell modellieren. Wir betrachten zum Beispiel die `while`-Schleife:

```
x = 1;
while x < 100
do
 x = 2*x
od
```

Eine `while`-Schleife besteht im Allgemeinen aus:

- dem Zustand vor der Schleife (Anfangswert)
- einer Bedingung
- einem Rumpf für die Zustandsänderung

In Haskell sieht das wie folgt aus:

```
while :: (a → Bool) → (a → a) → a → a
while p f x | p x       = while p f (f x)
            | otherwise = x
```

Dann wird `while (<100) (2*) 1` ausgewertet zu `128`.

Man beachte, dass es sich hierbei um keine Spracherweiterung handelt! Diese Kontrollstruktur ist nichts anderes als eine Funktion, ein Bürger erster Klasse.

1.5.5 Funktionen als Datenstrukturen

Was sind Datenstrukturen? Abstrakt betrachtet sind Datenstrukturen Objekte mit bestimmten Operationen:

- Konstruktoren (wie `(:)` oder `[]`)
- Selektoren (wie `head` oder `tail`, und pattern matching)
- Testfunktionen (wie `null`, und pattern matching)
- Verknüpfungen (wie `++`)

Wichtig ist dabei die Funktionalität, also die Schnittstelle, nicht die Implementierung. Eine Datenstruktur entspricht somit einem Satz von Funktionen.

Als Beispiel möchten wir Felder mit beliebigen Elementen in Haskell implementieren.

Die Konstruktoren haben folgenden Typ:

```
emptyArray :: Array a
putIndex :: Array a → Int → a → Array a
```

Wir benötigen nur einen einzigen Selektor:

```
getIndex :: Array a → Int → a
```

Nun wollen wir diese Schnittstelle implementieren. Dies können wir sehr einfach realisieren, in dem wir ein Feld nicht mit Hilfe anderer Datenstrukturen, wie z.B. Listen oder Bäume, implementieren, sondern ein Feld als Funktion implementieren. Die Umsetzung dieser Idee sieht dann zum Beispiel so aus:

```
type Array a = Int → a

emptyArray i =
  error ("Access to non-initialized component " ++ show i)

getIndex a i = a i

putIndex a i v = a'
```

```
  where a' j | i == j    = v
             | otherwise = a i
```

Der Vorteil dieser Implementierung ist ihre konzeptionelle Klarheit: Sie entspricht genau der Spezifikation. Ihr Nachteil liegt darin, dass die Zugriffszeit abhängig von der Anzahl der vorangegangenen `putIndex`-Aufrufe ist.

1.5.6 Wichtige Funktionen höherer Ordnung

Eine wichtige Funktion höherer Ordnung ist die Komposition von Funktionen `(.)`:

```
(.) :: (b → c) → (a → b) → a → c
(f . g) x = f (g x)
```

Eine weitere ist die bereits erwähnte Funktion `flip`, mit deren Hilfe die Reihenfolge der Parameter einer Funktion vertauscht werden kann:

```
flip :: (a → b → c) → b → a → c
flip f = \x y → f y x
```

Diese kann man zum Beispiel auf die Funktion `map` anwenden, um die bearbeitende Liste zuerst angeben zu können:

```
(flip map) :: [a] → (a → b) → [b]
(flip map) [1,2] :: (Int → b) → [b]
```

Zwei weitere interessante Funktionen höherer Ordnung sind die Funktionen `curry` und `uncurry`. Diese erlauben die Anwendung von Funktionen, die auf Tupeln definiert sind, auf einzelne Elemente, und umgekehrt:

```
curry :: ((a, b) → c) → a → b → c
curry f x y = f (x, y)

uncurry :: (a → b → c) → (a, b) → c
uncurry f (x, y) = f x y
```

Zuletzt betrachten wir noch die Funktion `const`, die zwei Argumente nimmt und das erste zurückgibt:

```
const :: a → b → a
const x _ = x
```

1.6 Typklassen und Überladung

Wir betrachten die Funktion `elem`, die überprüft, ob ein Element in einer Liste enthalten ist:

```
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

Was sind nun mögliche Typen von `elem`? Zum Beispiel wären dies:

```
Int  → [Int]  → Bool
Bool → [Bool] → Bool
Char → String → Bool
```

Leider ist "`a -> [a] -> Bool`" kein korrekter Typ, da ein beliebiger Typ `a` zu allgemein ist: `a` beinhaltet z.B. auch Funktionen, auf denen Gleichheit nur schwer definiert werden kann (und in Haskell nicht allgemein korrekt definierbar ist). Wir benötigen also eine Einschränkung auf Typen, für die die Gleichheit auf Werten definiert ist. Diese können wir in Haskell so ausdrücken:

```
elem :: Eq a => a → [a] → Bool
```

"`Eq a`" nennt man einen *Typconstraint*, der `a` einschränkt. Möchte man mehrere Typconstraints angeben, muss man diese durch Kommata getrennt in Klammern einfassen.

Die Klasse `Eq` ist in Haskell wie folgt definiert:

```
class Eq a where
  (==), (/=) :: a → a → Bool
```

In einer *Klasse* werden mehrere Funktionen zusammengefasst, die für alle Instanzen dieser Klasse (d.h. Typen) definiert sein müssen. Bei `Eq` sind das zum Beispiel die Funktionen `(==)` und `(/=)`.

Typen können als *Instanzen* einer Klasse definiert werden, in dem man die Implementierung der Funktionen der Klasse angibt:

```
data Tree = Empty | Node Tree Int Tree

instance Eq Tree where
  Empty        == Empty           = True
  Node tl n tr == Node tl' n' tr' = tl == tl' && n == n'
                                             && tr == tr'
  _            == _               = False

  t1 /= t2 = not (t1 == t2)
```

Dann ist `(==)` und `(/=)` für den Typ `Tree` verwendbar und wir können z.B. die obige Funktion `elem` auch auf Listen vom Elementtyp `Tree` anwenden.

Man kann auch polymorphen Typen zu Instanzen einer Klasse machen. Dann muss man eventuell auch Typconstraints bei der Instanzdefinition angeben, wie das folgende Beispiel zeigt:

```
data Tree a = Empty | Node (Tree a) a (Tree a)

instance Eq a => Eq (Tree a) where
  ...<wie oben>...
```

Beachte: So werden unendlich viele Typen zu Instanzen der Klasse `Eq`.

1.6.1 Vordefinierte Funktionen in einer Klasse

Die Definition von `(/=)` wird in fast jeder Instanzdefinition so wie oben aussehen.
Deshalb ist häufig eine Vordefinition in der Klassendefinition sinnvoll, welche aber in der Instanzdefinition überschrieben werden kann oder sogar muss:

```
class Eq a where
  (==), (/=) :: a  → a  → Bool
  x1 == x2 = not (x1 /= x2)
  x1 /= x2 = not (x1 == x2)
```

1.6.2 Vordefinierte Klassen

Für manche Typen ist es sinnvoll, eine totale Ordnung auf den Werten dieser Typen zu definieren. Diese finden wir in Haskell in einer Erweiterung von `Eq`:

```
data Ordering = LT | EQ | GT

class Eq a => Ord a where
  compare :: a  → a  → Ordering
  (<), (<=), (>=), (>) :: a  → a  → Bool
  max, min :: a  → a  → a

  ... -- vordefinierte Implementierungen
```

Eine minimale Instanzdefinition benötigt zumindest `compare` oder `(<=)`.
Weitere vordefinierte Klassen sind `Num`, `Show` und `Read`:

- `Num`: stellt Zahlen zum Rechnen dar (`(+) :: Num a => a -> a -> a`)
- `Show`: zum Verwandeln von Werten in Strings (`show :: Show a => a -> String`)

- `Read`: zum Konstruieren von Werten aus Strings (`read :: Read a => String -> a`)

Noch mehr vordefinierte Klassen werden im Master-Modul „Funktionale Programmierung“ vorgestellt.

Automatische Instantiierung vordefinierter Klassen (außer `Num`) erreicht man mittels `deriving` $(\kappa_1, ..., \kappa_n)$ hinter der Datentypdefinition.

Aufgabe: Überprüfen Sie die Typen aller in der Vorlesung definierten Funktionen auf ihren allgemeinsten Typ! Diese lauten zum Beispiel:

```
(+)   :: Num a => [a]  → [a]
nub   :: Eq a  => [a]  → [a]
qsort :: Ord a => [a]  → [a]
```

1.6.3 Die Klasse `Read`

Die Klasse `Show` dient zur textuellen Ausgabe von Daten. Um Daten zu lesen, d.h. aus einem String einen Wert zu erhalten, muss man einen String „parsen“, was eine schwierigere Aufgabe ist. Glücklicherweise gibt es auch hierzu eine vordefinierte Klasse. Allerdings muss man zur Benutzung etwas genauer verstehen, wie man Strings parsen kann.

Wir betrachten folgende Typdefinition, die den Typ von Funktionen zum Parsen von Strings in Werte definiert:

```
type ReadS a = String  → [(a,String)]
```

Was hat man sich hier beim Rückgabetyp von `ReadS` gedacht? Der erste Teil des Tupels ist der eigentliche Ergebniswert des Parsers, der zweite Teil ist der noch zu parsende Reststring: Betrachtet man zum Beispiel den String `"Node Empty 42 Empty"`, so wird schnell klar, dass nach dem Lesen der Anfangszeichenkette `Node` ein Baum folgen muss. Dann möchten wir den verbleibenden, noch zu parsenden Reststring erhalten, um ihn später zu betrachten.

Falls außerdem mehrere Ergebnisse möglich sind, werden diese als Liste zurückgegeben. Lässt sich der übergebene String nicht parsen, so ist der zurückgegebene Wert die leere Liste.

Das kann in der Anwendung so aussehen:

```
class Read a where
  readsPrec :: Int  → ReadS a
  readList :: ReadS [a] -- vordefiniert
```

Dann sind zwei Funktionen `reads` und `read` wie folgt definiert:

```
reads :: Read a => ReadS a
```

```
reads = readsPrec 0

read :: Read a => String  → a
read str = case reads str of
            [(x,"")]  → x
            _         → error "no parse"
```

Mittels `reads` kann man also einen String in einen Wert umwandeln und dabei prüfen, ob die Eingabe syntaktisch korrekt war. `read` kann man dagegen verwenden, wenn man sicher ist, dass die Eingabe syntaktisch korrekt ist.

Einige Auswertungen von Aufrufen von `reads` und `read` sehen dann zum Beispiel so aus:

```
reads "(3,'a')" :: [((Int,Char),String)]
       = [((3,'a'),"")]

reads "(3,'a')" :: [((Int,Int),String)]
       = []

read "(3,'a')" :: (Int,Char)
       = (3,'a')

read "(3,'a')" :: (Int,Int)
       = error: no parse

reads "3,'a')" :: [(Int,String)]
       = [(3,",'a'")]
```

1.7 Lazy Evaluation

Wir betrachten das Haskell-Programm

```
f x = 1
h = h
```

und den Ausdruck `f h`: Dann kann stets zuerst das `f` ausgewertet werden, was zum Ergebnis `1` führt, oder es wird zunächst versucht, `h` auszuwerten - was nie terminiert.

Nicht jeder Berechnungspfad terminiert also. Wir unterscheiden zwei ausgezeichnete Reduktionen:

- *leftmost-innermost* (LI): applikative Ordnung (strikte Funktionen)
- *leftmost-outermost* (LO): Normalordnung (nicht-strikte Funktionen)

Ein Vorteil der LO-Reduktion liegt darin, dass sie berechnungsvollständig ist: Alles, was irgendwie berechnet werden kann, wird auch berechnet. Allerdings kann LO auch ineffi-

zient sein, da Berechnungen verdoppelt werden können.
Kann LO trotzdem auch in der Praxis Vorteile bringen? Ja! Sie bietet

- Vermeidung überflüssiger (ggf. unendlicher) Berechnungen
- Rechnen mit unendlichen Datenstrukturen

Zum Beispiel definiert die folgende Funktion `from` die unendliche Liste natürlicher Zahlen, die mit `n` beginnt:

```
from :: Num a => a  → [a]
from n = n : from (n + 1)
```

Betrachten wir außerdem die schon bekannte Funktion `take`:

```
take :: Int  → [a]  → [a]
take n _      | n <= 0 = []
take _ []              = []
take n (x:xs)          = x : take (n - 1) xs
```

`take 1 (from 1)` wird ausgewertet zu `[1]`, denn LO liefert:

```
   take 1 (from 1)
= take 1 (1:from 2)
= 1 : take 0 (from 2)
= 1 : []
```

Der Vorteil liegt in der Trennung von Kontrolle (`take 1`) und Daten (`from 1`).

Als weiteres Beispiel betrachten wir die Primzahlberechnung mittels Sieb des Eratosthenes. Die Idee lautet wie folgt:

1. Betrachte die Liste aller Zahlen größer oder gleich 2.
2. Streiche alle Vielfachen der ersten (Prim-)Zahl.
3. Das erste Listenelement ist eine Primzahl. Mache bei 2. mit der Restliste weiter.

Dies lässt sich in Haskell zum Beispiel so implementieren:

```
sieve :: [Int]  → [Int]
sieve (p:xs) = p : sieve (filter (\x  → x `mod` p > 0) xs)

primes :: [Int]
primes = sieve (from 2)
```

Das Argument von `sieve` ist eine Eingabeliste, die mit einer Primzahl beginnt und in der alle Vielfachen kleinerer Primzahlen fehlen. Das Ergebnis ist eine Liste aller Primzahlen!
Jetzt liefert ein Aufruf von `take 10 primes` die ersten zehn Primzahlen:

```
[2,3,5,7,11,13,17,19,23,29]
```

Und mit Hilfe von `(!!)` können wir uns die zehnte Primzahl direkt ausgeben lassen: `primes!!9` wird ausgewertet zu `29`.

Die Programmierung mit unendlichen Datenstrukturen kann auch als Alternative zur Akkumulatortechnik verwendet werden. Wir nehmen dazu als Beispiel die Fibonaccifunktion. Um die n-te Fibonaccizahl zu erhalten, erzeugen wir die Liste aller Fibonaccizahlen und schlagen das n-te Element nach:

```
fibgen :: Int → Int → [Int]
fibgen n1 n2 = n1 : fibgen n2 (n1 + n2)

fibs :: [Int]
fibs = fibgen 0 1

fib :: Int → Int
fib n = fibs !! n
```

Dann wird `fib 10` ausgewertet zu `55`.

Ein Nachteil der LO-Strategie bleibt jedoch: Berechnungen können dupliziert werden. Wir betrachten erneut die einfache Funktion `double`:

```
double x = x + x
```

Übergeben wir dieser Funktion jetzt `(double 3)` als Argument, dann sieht die Auswertung nach der LI-Strategie so aus:

```
  double (double 3)
= double (3 + 3)
= double    6
= 6 + 6
= 12
```

Nach der LO-Strategie ergibt sich stattdessen:

```
  double (double 3)
= double 3 + double 3
= (3 + 3)  + double 3
=    6     + double 3
=    6     + (3 + 3)
=    6     +    6
=          12
```

Wegen der offensichtlichen Ineffizienz verwendet keine reale Programmiersprache die LO-Strategie.

Eine Optimierung der Strategie führt zur *Lazy-Auswertung*, bei der statt Termen Graphen reduziert werden. Variablen des Programms entsprechen dann Zeigern auf Ausdrücke, und die Auswertung eines Ausdrucks gilt für alle Variablen, die auf diesen Ausdruck verweisen: dies bezeichnet man als *sharing*. Man kann sich das Verhalten auch durch eine Normalisierung des Programms erklären, bei der für jeden Teilausdruck eine Variable eingeführt wird. Für obiges Beispiel sieht das aus wie folgt:

```
double x = x + x

main = let y = 3
           z = double y
       in double z
```

Dann verläuft die Auswertung so, wie auf Abbildung 1.3 dargestellt. Die schwarzen Linien zeigen dabei jeweils einen Reduktionsschritt an, blaue Linien sind Zeiger auf Ausdrücke.

Formalisiert wurde diese Strategie im Jahre 1993 von Launchbury [4]. Die Lazy-Auswertung ist optimal bzgl. Länge der Auswertung: Es erfolgt keine überflüssige Berechnung wie bei der LI-Strategie, und keine Duplikation wie bei der LO-Strategie. Allerdings benötigt sie manchmal viel Speicher.

In der Programmiersprache Haskell finden wir diese Lazy-Auswertung, in den Sprachen ML, Erlang, Scheme und Lisp finden wir hingegen die LI-Strategie.

Ein weiterer Vorteil der Lazy-Auswertung ist die schöne Komponierbarkeit von Funktionen: Angenommen, wir hätten eine Generator-Funktion, z.B. vom Typ `-> String`, und eine Konsumenten-Funktion, z.B. vom Typ `String -> Datei`.

Durch laziness entstehen hierbei keine großen Zwischendatenstrukturen, sondern nur die Teile, welche zur gegebenen Zeit benötigt werden, und auch diese werden danach direkt wieder freigegeben.

Haskell erlaubt auch *zyklische Datenstrukturen* wie z.B. die Liste

```
ones = 1 : ones
```

Diese kann sogar mit konstantem Speicherbedarf dargestellt werden.

1.8 Ein- und Ausgabe

Haskell ist eine reine funktionale Sprache, d.h. Funktionen haben keine Seiteneffekte. Wie kann Ein- und Ausgabe in solch eine Sprache integriert werden?

Die erste Idee lautet: Wie in anderen Sprachen (z.B. ML, Erlang oder Scheme) trotzdem als Seiteneffekte.

```
main = let str = getLine
        in putStr str
```

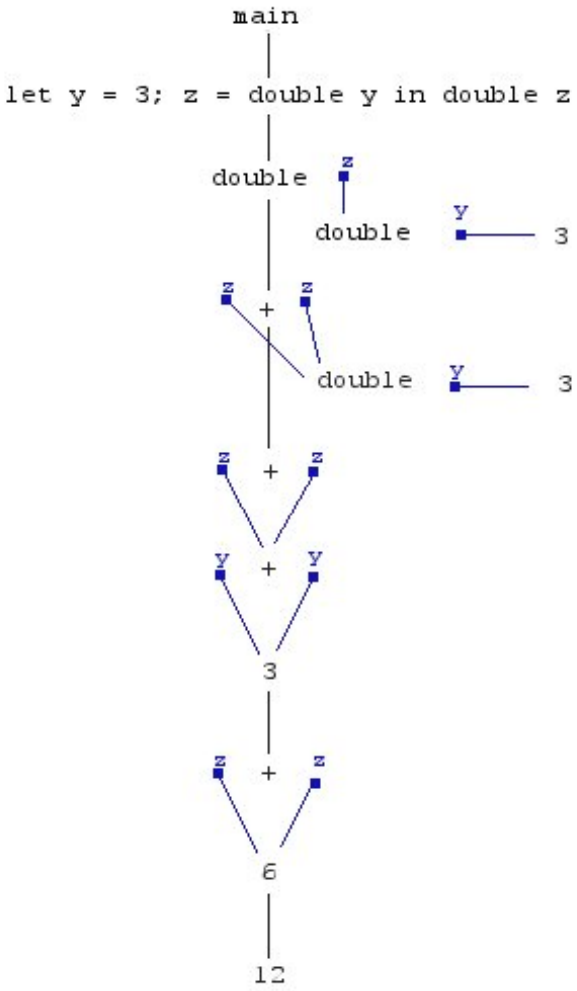


Abbildung 1.3: Sharing bei lazy evaluation

Hier soll `getLine` eine Zeile von der Tastatur einlesen und `putStr` einen String auf der Standardausgabe ausgeben. Was könnte der Typ von `main` oder `putStr` sein?

In Haskell haben wir als kleinsten Typ `()` (sprich: unit), dessen einziger Wert `()::()` ist (entspricht `void` in Sprachen wie Java). Hat `main` aber als Ergebnistyp `()`, dann ist wegen laziness doch eigentlich keine Ein- oder Ausgabe notwendig, um das Ergebnis `()` berechnen zu können...

Deshalb muss `putStr` als Seiteneffekt den übergebenen String ausgeben, bevor das Ergebnis `()` zurückgegeben wird.

Wir betrachten ein weiteres Beispiel:

```
main = let x = putStr "Hi"
       in x; x
```

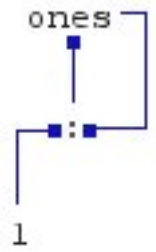


Abbildung 1.4: Zyklische Liste `ones`

Hier soll das Semikolon "`;`" bewirken, dass `Hi` zweimal hintereinander ausgegeben wird. Das Problem dabei ist jedoch, dass wegen der laziness `x` nur einmal berechnet wird, was wiederum bedeutet, dass der Seiteneffekt, die Ausgabe, nur einmal durchgeführt wird.

Ein weiteres Problem ergibt sich bei folgendem häufigen Szenario:

```
main = let dataBase = readDBFromUser
           request  = readRequestFromUser
       in print (lookup request dataBase)
```

Hier verhindert die laziness die gewünschte Eingabereihenfolge. All diese Probleme werden in Haskell mit Monaden gelöst.

1.8.1 I/O-Monade

Die Ein- und Ausgabe erfolgt in Haskell *nicht* als Seiteneffekt. Stattdessen liefern I/O-Funktionen eine *Aktion* zur Ein- oder Ausgabe als Ergebnis, also als Wert. Zum Beispiel ist

```
putStr "Hi"
```

eine Aktion, welche `Hi` auf die Standardausgabe schreibt, *wenn* diese ausgeführt wird.

Eine Aktion ist im Prinzip eine Abbildung vom Typ

```
World  →  (a, World)
```

wobei `World` den gesamten gegenwärtigen Zustand der „äußeren Welt“ beschreibt. Eine Aktion nimmt also den aktuellen Zustand der Welt und liefert einen Wert (z.B. die gelesene Eingabe) und einen veränderten Weltzustand. Wichtig ist die Tatsache, dass die Welt nicht direkt zugreifbar ist. Daher wird dieser Typ auch abstrakt mit "`IO a`" bezeichnet.

Ausgabeaktionen verändern nur die Welt und geben nichts zurück. Aus diesem Grund haben diese den Typ `IO ()`. Es gibt z.B. die folgenden vordefinierten Ausgabeaktionen:

```
putStr :: String  → IO ()
putChar :: Char  → IO ()
```

IO-Aktionen sind wie alle anderen Funktionen auch "first class citizens", sie können damit z.B. in Datenstrukturen gespeichert werden. Ein interaktives Programm definiert somit konzeptuell eine große IO-Aktion, welche auf die initiale Welt beim Programmstart angewendet wird und abschließend eine veränderte Welt liefert:

```
main :: IO ()
```

Das Zusammensetzen von IO-Aktionen erreicht man mit dem Sequenzoperator

```
(>>) :: IO ()  → IO ()  → IO ()
```

Dann sind äquivalent:

```
main = putStr "Hi" >> putStr "Hi"

main = let pHi = putStr "Hi"
        in pHi >> pHi

main = let actions = repeat (putStr "Hi")
        in actions !! 0 >> actions !! 42
```

Hier liefert `repeat :: a -> [a]` eine unendliche Liste mit identischen Elementen (dem Argument von `repeat`), also hier mit der übergebenen Aktion als Elemente.

Man kann die IO-Aktionen mit rein funktionalen Berechnungen wie üblich kombinieren. Wir können z.B. die Ergebnisse von Berechnungen ausgeben:

```
fac :: Int  → Int
fac n = if n == 0 then 1 else n * fac (n - 1)

main = putStr (show (fac 42))
```

Oder auch einfacher so:

```
main = print (fac 42)
```

Dabei ist `print` eine Funktion, die einen übergebenen Wert zuerst in eine Zeichenkette umwandelt und dann auf die Standardausgabe schreibt:

```
print :: Show a => a  → IO ()
print x = putStr (show x) >> putChar '\n'
```

Wir betrachten noch ein Beispiel: Folgende Definitionen von `putStr` sind äquivalent.

```
putStr :: String → IO ()
putStr ""     = return ()
putStr (c:cs) = putChar c >> putStr cs

putStr = foldr (\c → (putChar c >>)) (return ())
```

`return ()` ist hier sozusagen die „leere IO-Aktion“, die nichts macht und nur ihr Argument zurück gibt:

```
return :: a → IO a
```

Zur Eingabe von Daten gibt es entsprechende Aktionen, bei denen der Typ des Rückgabewertes dem Typ der Eingabedaten entspricht:

```
getChar :: IO Char
getLine :: IO String
```

Wie kann das Ergebnis einer IO-Aktion in einer nachfolgenden IO-Aktion weiter verwendet werden? Hierzu verwendet man den „Bind-Operator“:

```
(>>=) :: IO a → (a → IO b) → IO b
```

In der Verwendung sieht das zum Beispiel so aus:

```
getChar >>= putChar
```

Hier hat `getChar` den Typ `IO Char`, `putChar` hat den Typ `Char -> IO ()`. Folglich hat `getChar >>= putChar` also den Typ `IO ()`. Es liest ein Zeichen ein und gibt es wieder aus.
Als nächstes möchten wir eine ganze Zeile einlesen:

```
getLine :: IO String
getLine =
  getChar >>= \c → if c == '\n'
                   then return ""
                   else getLine >>= \cs → return (c:cs)
```

Wenn man genauer hinsieht, dann ähnelt die Zeichenfolge

```
... >>= \ ...
```

einer Zuweisung in einer imperativen Sprache, nur dass die linke und die rechte Seite vertauscht sind. Aus diesem Grund hat man zur besseren Lesbarkeit eine spezielle Notation eingeführt.

1.8.2 `do`-Notation

Mit `do {` $a_1; ...; a_n$ `}` oder

`do` a_1
$\vdots$
a_n

(man beachte die Layout-Regel nach dem `do`!) steht eine alternative, „imperative“ Notation zur Verfügung. Hierbei steht “`p <-` e_1`;` e_2” für den Bind-Aufruf

e_1 `>>= \p` $\rightarrow e_2$

Damit können wir die Operation `getLine` auch wie folgt definieren:

```
getLine = do
  c <- getChar
  if c == '\n' then return ""
               else do cs <- getLine
                       return (c:cs)
```

1.8.3 Ausgabe von Zwischenergebnissen

Wir möchten eine Funktion schreiben, die die Fakultät berechnet, und dabei alle Zwischenergebnisse der Berechnung ausgibt. Dies ist möglich, wenn wir die Funktion zu einer IO-Aktion umformulieren.

```
fac :: Int  → IO Int
fac n | n==0      = return 1
      | otherwise = do f <- fac (n-1)
                       print (n-1,f)
                       return (n * f)

main :: IO ()
main = do putStr "n: "
          str <- getLine
          facn <- fac (read str)
          putStrLn ("Factorial: " ++ show facn)
```

Die Verwendung sieht dann so aus:

```
> main
n: 6
(0,1)
(1,1)
```

```
(2,2)
(3,6)
(4,24)
(5,120)
Factorial: 720
```

Aber solche Programme sollte man vermeiden! Es ist immer besser, Ein- und Ausgabe auf der einen Seite und Berechnungen auf der anderen Seite zu trennen. Als gängiges Schema hat sich etabliert:

```
main = do input <- getInput
          let res = computation input
          print res
```

Hier ist die Zeile

```
          let res = computation input
```

eine rein funktionale Berechnung. Das `let` benötigt im `do`-Block kein `in`.

1.9 List Comprehensions

Haskell stellt noch ein wenig syntaktischen Zucker für Listendefinitionen zur Verfügung: So lässt sich die Liste der ganzen Zahlen von eins bis vier in Haskell mit `[1..4]` beschreiben. Man kann auch die Obergrenze weglassen und erhält eine unendliche Liste, wie z.B. in "`take 10 [1..]`". Man kann auch die ersten beiden Listenelement vorgeben, um einen anderen Abstand als +1 zu verwenden. Zum Beispiel wird `[3,1..]` zu der unendlichen Liste ausgewertet, die mit drei beginnt und dann immer mit dem Vorvorgänger fortfährt: `[3,1,-1,-3,-5,...]`.
Wir können sogar Listen mit einer an die Mathematik angelehnten Notation beschreiben: So liefert der Ausdruck

```
[(i,j) | i <- [1..3], j <- [2,3,4], i /= j]
```

die Liste `[(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4)]`. Erlaubt sind dabei *Generatoren* wie `i <- [1..3]` und *boolesche Bedingungen* wie `i /= j`. Auch `let` ist erlaubt. Somit liefert

```
[[0..n] | n <- [0..]]
```

alle endlichen Anfangsfolgen der Menge der natürlichen Zahlen.
Abschließend betrachten wir noch eine andere Definition der Funktion `concat`, die aus einer Liste von Listen eine Ergebnisliste mit deren Einträgen macht:

```
concat :: [[a]] → [a]
```

```
concat xss = [y | ys <- xss, y <- ys]
```

1.10 Module

Wie fast jede andere Programmiersprache unterstützt auch Haskell das Programmieren im Großen durch Aufteilung eines Programms in mehrere Module. Da Module in Haskell ähnlich wie in anderen Sprachen organisiert sind, geben wir im Folgenden nur einen kurzen Überblick.

Ein *Modul* definiert eine Menge von Namen (von Funktionen oder Datenkonstruktoren), die verwendet werden können, wenn man dieses Modul importiert. Dabei kann man die Menge der Namen mittels *Exportdeklarationen* einschränken, sodass z.B. die Namen von nur lokal relevanten Funktionen nicht exportiert werden, d.h. außen nicht sichtbar sind. Der Quelltext eines Moduls beginnt daher wie folgt:

```
module MyProgram (f, g, h) where
```

Hierbei ist `MyProgram` der *Modulname*, der identisch mit dem Dateinamen sein muss (Ausnahme: hierarchische Modulnamen, die wir hier nicht weiter diskutieren). Somit wird dieses Modul in der Datei `MyProgram.hs` gespeichert. Dahinter folgt in Klammern die Liste der exportierten Namen, in diesem Fall werden also die Namen `f`, `g` und `h` exportiert. Normalerweise müssen die Deklarationen hinter einem `where` weiter eingerückt werden, allerdings können diese bei einem Modul auch in der ersten Spalte beginnen, sodass wir Programme wie üblich aufschreiben können. Somit könnte unser gesamtes Modul wie folgt aussehen:

```
module MyProgram (f, g, h) where

f = 42

g = f*f

h = f+g
```

Hierbei ist zu beachten, dass Modulnamen immer mit einem Großbuchstaben beginnen müssen!

Von diesem allgemeinen Schema kann man wie folgt abweichen:

- Die Exportliste (d.h. "`(f, g, h)`") kann auch fehlen; in diesem Fall werden *alle* in dem Modul definierten Namen exportiert.
- In der Exportliste können auch Typkonstruktoren aufgelistet werden. In diesem Fall wird auch dieser Typkonstruktorname exportiert, jedoch nicht die zugehörigen Datenkonstruktoren. Sollen auch diese exportiert werden, dann muss man hinter Typkonstruktornamen in der Exportliste "`(..)`" schreiben, wie z.B. in

```
module Nats(Nat(..),add,mult) where

data Nat = Z | S Nat
 deriving Show

add Z     y = y
add (S x) y = S (add x y)

mult Z     _ = Z
mult (S x) y = add y (mult x y)
```

- Falls kein Modulkopf in der Quelldatei angegeben ist, wird implizit der Modulkopf

  ```
  module Main(main) where
  ```

  eingesetzt.

Wir haben bisher ein interaktives Haskell-System wie `ghci` verwendet, um kleine Programme auszutesten. Man kann auch eine ausführbare Datei ("executable") aus einem Haskell-Programm mit Hilfe des `ghc` erzeugen. Hierzu muss das Programm (bzw. der Hauptteil des Programms) im Modul `Main` abgelegt sein, das wiederum eine Hauptfunktion

```
main :: IO ()
```

enthalten muss, d.h. eine I/O-Aktion, welche als Hauptprogramm ausgeführt wird.[1] Dann kann man ein ausführbares Maschinenprogramm `myexec` z.B. durch

```
ghc -o myexec Main.hs
```

erzeugen. Typischerweise benutzt das Hauptprogramm eine Reihe anderer Module, die mittels einer `import`-Deklaration importiert werden können. Z.B. könnte unser Hauptprogramm wie folgt aussehen:

```
module Main where

import Nats

main = print (add (S Z) (S Z))
```

Wenn wir dieses mit

```
ghc -o myexec Main.hs
```

übersetzen, erhalten wir eine ausführbare Datei `myexec`, die wir dann direkt ausführen

[1] Genau hierdurch wird also eine I/O-Aktion mit einem Weltzustand verbunden!

können:

```
> ./myexec
S (S Z)
```

Durch eine `import`-Deklaration werden die von dem importierten Modul exportierten Namen in dem aktuellen Modul sichtbar gemacht. Hierbei gibt es eine Reihe von Varianten:

- Man kann die importierten Namen durch eine Aufzählung einschränken. Z.B. wird durch

  ```
  import Nats (Nat(..),add)
  ```

  der Name `mult` nicht importiert.
- Man kann auch alle Namen bis auf einige Ausnahmen mittels einer `hiding`-Einschränkung importieren:

  ```
  import Nats hiding (mult)
  ```

- Das Standardmodul `Prelude` wird immer implizit importiert, wenn dies nicht explizit angegeben ist, wie z.B. in

  ```
  import Prelude hiding (map,foldr)
  ```

  wodurch die Standardoperationen `map` und `foldr` nicht importiert werden.
- Zur Vermeidung von Mehrdeutigkeiten kann man innerhalb eines Moduls auf importierte Namen auch qualifiziert zugreifen, wie z.B. in

  ```
  module Main where

  import Nats

  main = Prelude.print (Nats.add (S Z) (S Z))
  ```

- Wenn man erzwingen will, dass auf importierte Namen immer qualifiziert zugegriffen werden muss, kann man dies durch die Einschränkung `qualified` kenntlich machen:

  ```
  module Main where

  import qualified Nats

  main = print (Nats.add (Nats.S Nats.Z) (Nats.S Nats.Z))
  ```

- Wenn dadurch zu lange Namen entstehen, kann man das Modul beim Import auch umbenennen:

```
module Main where

import qualified Nats as N

main = print (N.add (N.S N.Z) (N.S N.Z))
```

Darüberhinaus gibt es noch weitere Möglichkeiten und Regeln bei der Verwendung von Modulen, die hier nicht alle beschrieben werden können. Hierzu sollte man die Sprachdefinition von Haskell konsultieren.

2 Einführung in die Logikprogrammierung

2.1 Motivation

Die Logikprogrammierung hat eine ähnliche Motivation wie die funktionale Programmierung:

- Abstraktion von der konkreten Ausführung eines Programms auf dem Rechner
- Programme als mathematische Objekte, aber hier: Relationen statt Funktionen
- Computer soll *Lösungen finden* (und nicht nur einen Wert berechnen)

Im Vergleich zur funktionalen Programmierung zeichnet sich die Logikprogrammierung durch folgende Eigenschaften aus:

- weniger Aussagen über die Richtung von Berechnungen/Daten
- Spezifikation von Zusammenhängen (Relationen) zwischen Objekten
- flexible Benutzung der Relationen

Beispiel: Verwandtschaftsbeziehungen

Als Einstiegsbeispiel möchten wir einmal Verwandschaftsbeziehungen implementieren. Ziel ist die Berechnung von Antworten auf Fragen wie

„Wer ist die Mutter von Monika?“

„Welche Großväter hat Andreas?“

„Ist Monika die Tante von Andreas?“

und ähnliche andere Fragen. Unser konkretes Beispiel sieht so aus:

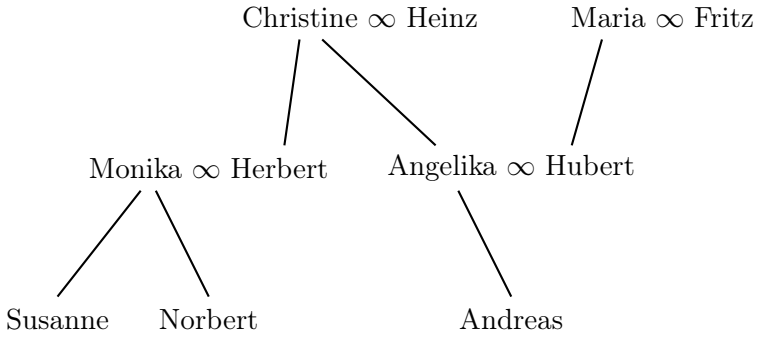
- Christine ist verheiratet mit Heinz.
- Christine hat zwei Kinder: Herbert und Angelika.
- Herbert ist verheiratet mit Monika.
- Monika hat zwei Kinder: Susanne und Norbert.
- Maria ist verheiratet mit Fritz.
- Maria hat ein Kind: Hubert.
- Angelika ist verheiratet mit Hubert.
- Angelika hat ein Kind: Andreas.

Wir stellen unsere Beispielverwandschaft graphisch dar, wobei wir die folgenden Beziehungen einbeziehen:

∞ : verheiratet

/ : Mutter-Kind-Beziehung

Dann könnte unsere Beispielverwandschaft wie folgt aussehen:



In der funktionalen Programmiersprache Haskell lassen sich diese Beziehungen zum Beispiel wie folgt modellieren: Personen modellieren wir als Datentyp (man könnte stattdessen auch String o.ä. nehmen):

```
data Person = Christine | Heinz | Maria | ... | Andreas
```

Die „verheiratet“-Beziehung definieren wir als Funktion

```
ehemann :: Person  → Person
ehemann   Christine = Heinz
ehemann   Maria     = Fritz
ehemann   Monika    = Herbert
ehemann   Angelika  = Hubert
```

Die „Mutter-Kind“-Beziehung definieren wir als Funktion

```
mutter :: Person  → Person
mutter   Herbert  = Christine
mutter   Angelika = Christine
mutter   Hubert   = Maria
mutter   Susanne  = Monika
mutter   Norbert  = Monika
mutter   Andreas  = Angelika
```

Aus diesen grundlegenden Beziehungen können wir weitere allgemeine Beziehungen ableiten:
Der Vater ist der Ehemann der Mutter (in einer streng katholischen Verwandtschaft!):

```
vater :: Person  → Person
vater kind = ehemann (mutter kind)
```

Die Enkel-Großvater-Beziehung ist im Allgemeinen eine *Relation*:

```
grossvater :: Person  → Person  → Bool
grossvater e g | g == vater (vater e ) = True
               | g == vater (mutter e) = True
               | otherwise             = False
```

Jetzt finden sich schnell Antworten auf Fragen wie „Wer ist Vater von Norbert?“ oder „Ist Heinz Großvater von Andreas?“:

```
> vater Norbert
Herbert

> grossvater Andreas Heinz
True
```

Folgende Fragen kann unser Programm aber nicht direkt beantworten:

1. Welche Kinder hat Herbert?
2. Welche Großväter hat Andreas?
3. Welche Enkel hat Heinz?

Dies wäre möglich, falls *Variablen in Ausdrücken* zulässig wären:

1. `vater` k `== Herbert` $\rightsquigarrow$ k `= Susanne` oder k `= Norbert`
2. `grossvater Andreas` g $\rightsquigarrow$ g `= Heinz` oder g `= Fritz`
3. `grossvater` e `Heinz` $\rightsquigarrow$ e `= Susanne` oder e `= Norbert` oder e `= Andreas`

Genau dies ist in Logiksprachen wie Prolog erlaubt. Diese haben folgende Charakteristik:

- „Freie“ („logische“) Variablen sind in Ausdrücken (und Regeln) erlaubt.
- Berechnung von *Lösungen*, d.h. Werte für freie Variablen, so dass der Ausdruck berechenbar („beweisbar“) ist.
- Problem: Wie findet man konstruktiv die Lösungen? → später
- Berechnungsprinzip: Schlussfolgerungen aus gegebenem Wissen ziehen.

Ein *Prolog-Programm* ist eine Menge von Fakten und Regeln für Prädikate, wobei Prädikate Aussagen über die Objekte sind. Eine *Aussage* wiederum besteht aus

1. der Art der Aussage (Eigenschaft): 3 ist *Primzahl*
2. den beteiligten Objeken: *3* ist Primzahl.

Diese gibt man in Prolog in der *Standardpräfixschreibweise* an:

`name(`$objekt_1, ..., objekt_n$`)`

z.B. `primzahl(3)` oder `ehemann(monika, herbert)`. Dabei ist folgendes zu beachten:

1. Alle Namen (auch *Atome* genannt) werden klein geschrieben.
2. Die Reihenfolge der Objekte in einer Aussage ist relevant.
3. Vor der öffnenden Klammer darf kein Leerzeichen stehen.

4. Es gibt auch andere Schreibweisen (Operatoren, später).

Als *Fakten* bezeichnen wir Aussagen, die als richtig angenommen werden. Syntaktisch werden diese durch einen Punkt und ein whitespace (Leerzeichen, Zeilenvorschub) am Ende abgeschlossen:

```
ehemann(christine, heinz).
ehemann(maria, fritz).
ehemann(monika, herbert).
ehemann(angelika, hubert).

mutter(herbert,christine).
mutter(angelika,christine).
mutter(hubert,maria).
mutter(susanne,monika).
mutter(norbert,monika).
mutter(andreas,angelika).
```

Fakten alleine sind nicht ausreichend, denn sie entsprechen einem Notizbuch oder einer relationalen Datenbank. Für komplexere Probleme benötigen wir *Regeln*Regel oder *Schlussfolgerungen* zur Ableitung neuer richtiger Aussagen:

Wenn Aussage1 und Aussage2 richtig sind, *dann* ist Aussage3 richtig.

Dies schreiben wir in Prolog folgendermaßen:

```
Aussage3 :- Aussage1, Aussage2.
```

Hier steht das Komma "`,`" für das logische Und ($\wedge$), "`:-`" steht für einen Schlussfolgerungspfeil ($\Leftarrow$).

So lautet zum Beispiel die Regel „Der Vater von Susanne ist Herbert, falls Herbert Ehemann von Monika und Monika Mutter von Susanne ist." in Prolog:

```
vater(susanne,herbert) :-
    ehemann(monika,herbert),
    mutter(susanne,monika).
```

Diese Regel ist natürlich sehr speziell, aber Prolog erlaubt hier natürlich eine Verallgemeinerung dieser Regel. Wir können statt fester Namen auch unbekannte Objekte angeben, die auch als *Variablen* bezeichnet werden. Hierbei ist zu beachten:

- Variablen beginnen mit einem Großbuchstaben
- Variablen stehen für beliebige andere Objekte
- Regeln oder Fakten mit Variablen repräsentieren unendlich viele Regeln

Mit Variablen können wir z.B. folgende allgemeine Regeln für Vater- und Großvaterbeziehungen formulieren:

```
vater(Kind,Vater) :- ehemann(Mutter,Vater),
                     mutter(Kind,Mutter).

grossvater(E,G) :- vater(E,V),vater(V,G).
grossvater(E,G) :- mutter(E,M),vater(M,G).
```

Variablen in Regeln haben die folgende Bedeutung: die Regeln sind richtige Schlussfolgerungen für alle Werte, die wir an Stelle der Variablen einsetzen (ähnlich wie bei Gleichungsdefinitionen in Haskell).

Anfragen

Fakten und Regeln zusammen entsprechen dem *Wissen* über ein Problem. Nachdem wir diese eingegeben haben, können wir *Anfragen* an das Prolog-System stellen: Anfragen sind Aussagen, deren Wahrheitsgehalt geprüft werden soll.

```
?- vater(norbert,herbert).
yes
?- vater(andreas,herbert).
no
```

Mittels Variablen in Anfragen können wir nun unser Wissen flexibel verwenden. Variablen in Anfragen haben die Bedeutung: Für welche Werte an Stelle der Variablen ist die Aussage richtig?
Die Anfrage „Wer ist der Mann von `monika`?“ kann so formuliert werden:

```
?- ehemann(monika,Mann).
Mann = herbert
```

Und die Anfrage „Welche Enkel hat Heinz?“ so:

```
?- grossvater(Enkel,heinz).
Enkel = susanne
```

Die Eingabe eines Semikolons “`;`” fordert das Prolog-System auf, weitere Lösungen zu suchen:

```
?- grossvater(Enkel,heinz).
Enkel = susanne ;
Enkel = norbert ;
Enkel = andreas ;
no
```

Hier gibt das Prolog-System mit einem “`no`” zu verstehen, dass keine weiteren Lösungen gefunden wurden.

Zusammenfassung: Begriffe der Logikprogrammierung:

Ein Logikprogramm besteht also im Wesentlichen aus den folgenden Bestandteilen:

- *Atome* (elementare Objekte)
- *Fakten* (gültige Aussagen)
- *Regeln* (wenn-dann-Aussagen)
- *Anfragen* (Ist eine Aussage gültig?)
- *Variablen* (Für welche Werte ist eine Aussage gültig?)

Eine Eigenschaft oder Beziehung kann sowohl mit Fakten als auch mit Regeln definiert werden. Daher nennt man Fakten und Regeln auch *Klauseln* für diese Eigenschaft oder Beziehung. Allgemein nennt man letzteres auch *Prädikat* (angewendet auf Objekte: entweder wahr oder falsch) oder *Relation*.

Soweit ist Prolog das, was man mit der Prädikatenlogik 1. Stufe beschreiben kann. Es gibt Konstanten, Prädikate, Variablen (Funktionen) und Quantifizierung über Individualvariablen (um genau zu sein: Logikprogrammierung basiert auf einer Teilmenge der Prädikatenlogik 1. Stufe). Da aber die Prädikatenlogik 1. Stufe nicht entscheidbar ist, kann es für jedes leistungsfähige Beweissystem eine Menge von Klauseln geben, bei denen eine bestimmte Anfrage nicht mit yes oder no beantwortet werden kann. Dies bedeutet, dass auch Prolog-Programme nicht in jedem Fall terminieren. Die genaue Auswertungsstrategie von Prolog wird später vorgestellt.

2.2 Syntax von Prolog

Wie in jeder Programmiersprache sind Objekte in Prolog entweder elementar (d.h. *Zahlen* oder *Atome*) oder strukturiert. Zur genauen Definition der Syntax zeichnen wir vier Kategorien von Zeichenmengen aus:

Großbuchstaben: `A B ... Z`

Kleinbuchstaben: `a b ... z`

Ziffern: `0 1 ... 9`

Sonderzeichen: `+ - * / < = > ‘ \ : . ? @ # $ & ^ ~`

Dann sind Prolog-Objekte, auch *Terme* genannt, wie folgt aufgebaut:

Zahlen sind Folgen von Ziffern (oder Gleitkommazahlen in üblicher Syntax)

Atome bilden unzerlegbare Prolog-Objekte:

- Folge von Kleinbuchstaben, Großbuchstaben, Ziffern, “`_` ”, beginnend mit einem Kleinbuchstaben; oder
- Folge von Sonderzeichen; oder
- beliebige Sonderzeichen, eingefasst in `’`, z.B. `’ein Atom!’`; oder
- „Sonderatome“ (nicht beliebig verwendbar): `, ; ! []`

Konstanten sind Zahlen oder Atome.

Strukturen entstehen durch Zusammenfassung mehrerer Objekte zu einem. Eine Struktur besteht aus:

- *Funktor* (entspricht Konstruktor)
- *Komponenten* (beliebige Prolog-Objekte)

Ein Beispiel für eine Struktur: `datum(1,6,27)`. Hier bezeichnet `datum` den Funktor, `1`, `6` und `27` sind die Komponenten. Zwischen Funktor und Komponenten darf *kein Leerzeichen* stehen.

Strukturen können auch geschachtelt werden:

```
person(fritz,meier,datum(1,6,27))
```

Der Funktor einer Struktur ist dabei relevant: `datum(1,6,27)` ist nicht das Gleiche wie `zeit(1,6,27)`.

Listen

Listen sind wie in Haskell die wichtigste Strukturierungsmöglichkeit für eine beliebige Anzahl von Objekten. *Listen* sind in Prolog induktiv definiert durch:

- leere Liste: `[]`
- Struktur der Form `.(E,L)`, wobei `E` das erste Element der Liste darstellt und `L` die Restliste.

Eine Liste mit den Elementen `a`, `b` und `c` könnte also so aussehen:

```
.(a, .(b, .(c,[])))
```

Auch in Prolog gibt es Kurzschreibweisen für Listen: `[`E_1`,`E_2`,...,`E_n`]` steht für eine Liste mit den Elementen $E_1, E_2, ..., E_n$, `[E|L]` steht für `.(E,L)`.

Die folgenden Listen sind also äquivalent:

```
.(a,.(b,.(c,[])))
[a,b,c]
[a|[b,c]]
[a,b|[c]]
```

Texte werden in Prolog durch Listen von ASCII-Werten beschrieben:
Der Text `"Prolog"` entspricht also der Liste `[80,114,111,108,111,103]`.

Operatoren

Auch in Prolog gibt es *Operatoren*: So lässt sich „Die Summe von 1 und dem Produkt von 3 und 4“ beschreiben durch die Struktur `+(1,*(3,4))`. Doch es gibt auch die natürliche Schreibweise: `1+3*4` beschreibt also das Gleiche.

Die *Operatorschreibweise* von Strukturen sieht in Prolog so aus:

1. Strukturen mit einer Komponente
   a) *Präfixoperator*: `-2` entspricht `-(2)`
   b) *Postfixoperator*: `2 fak fak` entspricht `fak(fak(2))`
2. Strukturen mit zwei Komponenten
   a) *Infixoperator*: `2+3` entspricht `+(2,3)`

Bei Infixoperatoren entsteht natürlich sofort das Problem der Eindeutigkeit:

- `1-2-3` kann interpretiert werden als
  - "`-(-(1,2),3)`": dann ist "`-`" linksassoziativ
  - "`-(1,-(2,3))`": dann ist "`-`" rechtsassoziativ
- `12/6+1`
  - `+(/(12,6),1)`: "`/`" bindet stärker als "`+`"
  - `/(12,+(6,1))`: "`+`" bindet stärker als "`/`"

Der Prolog-Programmierer kann selbst Operatoren definieren. Dazu muss er Assoziativität und Bindungsstärke angeben. Dabei verwendet man die *Direktive* "`:- op(...)`" (genaueres kann man im Prolog-Handbuch nachlesen). Die üblichen mathematischen Operatoren wie z.B. `+ - *` sind bereits vordefiniert. Natürlich ist es auch immer möglich, Klammern zu setzen: `12/(6+1)`.

Variablen

Variablen in Prolog werden durch eine Folge von Buchstaben, Ziffern und `_`, beginnend mit einem Großbuchstaben oder `_` beschrieben:

- `datum(1,4,Jahr)` entspricht allen ersten Apriltagen
- `[a|L]` entspricht der Liste mit `a` als erstem Element
- `[A,B|L]` entspricht allen mindestens zwei-elementigen Listen

Variablen können in einer Anfrage oder Klausel auch mehrfach auftreten: So entspricht `[E,E|L]` allen Listen mit mindestens zwei Elementen, wobei die ersten beiden Elemente identisch sind.

Auch Prolog bietet *anonyme Variablen*: `_` repräsentiert ein Objekt, dessen Wert nicht interessiert. Hier steht jedes Vorkommen von `_` für einen anderen Wert. Als Beispiel greifen wir auf unser Verwandschaftsbeispiel zurück:

```
istEhemann(Person) :- ehemann(_, Person).
```

Die Anfrage `?- mutter(_,M).` fragt das Prolog-System nach allen Müttern.

Jede Konstante, Variable oder Struktur in Prolog ist ein *Term*. Ein *Grundterm* ist ein Term ohne Variablen.

Rechnen mit Listenstrukturen

Wir betrachten nun ein Beispiel zum Rechnen mit Listenstrukturen. Unser Ziel ist es, ein Prädikat `member(E,L)` zu definieren, das wahr ist, falls `E` in der Liste `L` vorkommt. Wir müssen unser Wissen über die Eigenschaften von `member` als Fakten und Regeln ausdrücken. Eine intuitive Lösung wäre, hierfür viele Regeln anzugeben:

- Falls `E` erstes Element von `L` ist, dann ist `member(E,L)` wahr.
- Falls `E` zweites Element von `L` ist, dann ist `member(E,L)` wahr.
- Falls `E` drittes Element von `L` ist, dann ist `member(E,L)` wahr.
- ...

Da dies zu unendlich vielen Regeln führen würde, können wir uns auch folgendes überlegen:

> `member(E,L)` ist wahr, falls `E` das erste Element von `L` ist oder im Rest von `L` vorkommt.

In Prolog können wird dies einfach wie folgt ausdrücken:

```
member(E,[E|_]).
member(E,[_|R]) :- member(E,R).
```

Nun können wir Anfragen an das Prolog-System stellen:

```
?- member(X,[1,2,3]).
X=1 ;
X=2 ;
X=3 ;
no
```

Gleichheit von Termen

Wir betrachten einmal *Gleichheit von Termen* etwas genauer. Vergleichsoperatoren in Sprachen wie Java oder Haskell, z.B. ==, beziehen sich immer auf die Gleichheit nach dem Ausrechnen der Ausdrücke auf beiden Seiten. In Prolog bezeichnet = hingegen die strukturelle Termgleichheit: Es wird nichts ausgerechnet!

```
?- 5 = 2+3.
no

?- datum(1,4,Jahr) = datum(Tag,4,2009).
Jahr = 2009
Tag = 1
```

2.3 Elementare Programmiertechniken

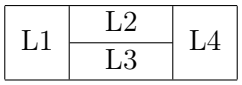
In diesem Kapitel zeigen wir einige grundlegende logische Programmiertechniken.

2.3.1 Aufzählung des Suchraumes

Beim Färben einer Landkarte mit beispielsweise vier Ländern hat man die vier Farben rot, gelb, grün und blau zur Verfügung und sucht eine Zuordnung, bei der aneinandergrenzende Länder verschiedene Farben haben. Die vier Länder seien wie folgt angeordnet:

- L1 grenzt an L2 und L3.
- L2 grenzt an L1, L3 und L4.
- L3 grenzt an L1, L2 und L4.
- L4 grenzt an L2 und L3.

Graphisch:



Was *wissen* wir über das Problem?

1. Es stehen vier Farben zur Verfügung:

```
farbe(rot).
farbe(gelb).
farbe(gruen).
farbe(blau).
```

2. Jedes Land hat eine dieser Farben:

```
faerbung(L1,L2,L3,L4) :- farbe(L1),farbe(L2),farbe(L3),farbe(L4).
```

3. Wann sind zwei Farben verschieden?

```
verschieden(rot,gelb).
verschieden(rot,gruen).
verschieden(rot,blau).
...
verschieden(gruen,blau).
```

4. Korrekte Lösung: Aneinandergrenzende Länder haben verschiedene Farben:

```
korrekteFaerbung(L1,L2,L3,L4) :-
    verschieden(L1,L2),
    verschieden(L1,L3),
```

```
verschieden(L2,L3),
verschieden(L2,L4),
verschieden(L3,L4).
```

5. Gesamtlösung des Problems:

```
?- faerbung(L1,L2,L3,L4), korrekteFaerbung(L1,L2,L3,L4).
L1 = rot
L2 = gelb
L3 = gruen
L4 = rot
```

   (weitere Lösungen durch Eingabe von ";")

Wir wollen das obige Beispiel einmal analysieren. Dieses Beispiel ist typisch für die Situation, dass man nicht weiß, wie man eine Lösung auf systematischem Weg erhält. Um in diesem Fall mit Hilfe der Logikprogrammierung zu einer Lösung zu kommen, benötigen wir die folgenden Dinge:

- Angabe der potentiellen Lösungen (`faerbung`): Wir beschreiben die Struktur möglicher Lösungen.
- Charakterisierung der korrekten Lösungen
- Gesamtschema:

```
loesung(L) :- moeglicheLoesung(L), korrekteLoesung(L).
```

Dieses Schema wird *generate-and-test* genannt.
Die Komplexität hängt im Wesentlichen von der Menge der möglichen Lösungen ab (auch *Suchraum* genannt). In diesem Beispiel gibt es $4^4 = 256$ mögliche Lösungen. Dies ist in diesem Fall akzeptabel, aber manchmal kann dies auch wesentlich schlimmer sein.

Sortieren von Zahlen

Um einzusehen, dass die Komplexität auch sehr groß werden kann, betrachten wir als nächstes Beispiel das Sortieren von Zahlen, `sortiere(UL,SL)`. Hierbei sei `UL` eine Liste von Zahlen und `SL` eine sortierte Variante von `UL`.

1. Wann ist eine Liste sortiert? D.h. was sind korrekte Lösungen?
   Wenn jedes Element kleiner oder gleich dem nachfolgenden ist.
   Ausgedrückt in Prolog-Standardoperationen auf Listen (hier ist das Prädikat `x =< y` erfüllt, wenn die Werte von `x` und `y` Zahlen sind, die in der kleiner-gleich Beziehung stehen):

```
sortiert([]).
sortiert([_]).
sortiert([E1,E2|L]) :- E1 =< E2, sortiert([E2|L]).
```

2. Was sind mögliche Lösungen?
   Die sortierte Liste `SL` ist eine Permutation von `UL`. Eine Permutation enthält die gleichen Elemente aber eventuell in einer anderen Reihenfolge. Wir definieren Permutation durch Streichen von Elementen:

   ```
   perm([],[]).
   perm(L1,[E|R2]) :- streiche(E,L1,R1), perm(R1,R2).
   ```

   Definition von `streiche` darüber, ob das zu streichende Element im Kopf der Liste vorhanden ist oder nicht:

   ```
   streiche(E,[E|R],R).
   streiche(E,[A|R],[A|RohneE]) :- streiche(E,R,RohneE).
   ```

3. Nach dem Schema erhalten wir folgende Gesamtlösung:

   ```
   sortiere(UL,SL) :-
       perm(UL,SL),  % moegliche Loesung
       sortiert(SL). % korrekte Loesung
   ```

4. Diese logische Spezifikation ist ausführbar:

   ```
   ?- sortiere([3,1,4,2,5],SL).
   SL = [1,2,3,4,5]
   ```

Die Komplexität dieses Beispiels liegt für eine n-elementige Liste in der Größenordnung $O(n!)$, denn eine n-elementige Liste hat $n!$ mögliche Permutationen. Dies bedeutet, dass für eine Liste mit zehn Elementen es bereits 3.628.800 mögliche Lösungen gibt. Daher ist diese Lösung für die Praxis unbrauchbar.
In solchen Fällen hilft nur eine genauere Problemanalyse (Entwicklung von besseren Verfahren, z.B. Sortieralgorithmen) weiter.

2.3.2 Musterorientierte Wissensrepräsentation

Ein typisches Beispiel für musterorientiere Wissensrepräsentation ist die Listenverarbeitung. Häufig reicht hier eine einfache Fallunterscheidung, im Fall von Listen unterscheiden wir die Fälle der leeren und nicht-leere Liste. Daraus resultiert ein kleiner, möglicherweise sogar ein-elementiger Suchraum.

Als Beispiel betrachten wir das Prädikat `append(L1,L2,L3)`, das zwei Listen `L1` und `L2` zu einer Liste `L3` konkatenieren soll. Genauer soll gelten:

`append(L1,L2,L3)` $\Longleftrightarrow$
`L1` $= [a_1, \ldots, a_m]$ $\wedge$ `L2` $= [b_1, \ldots, b_n]$ $\wedge$ `L3` $= [a_1, ..., a_m, b_1, ..., b_n]$

Wir können dies durch Fallunterscheidung über die Struktur der ersten Liste `L1` definie-

ren:

1. Wenn `L1` leer ist, dann ist `L3` gleich `L2`.
2. Wenn `L1` nicht leer ist, dann ist das erste Element von `L3` gleich dem ersten Element von `L1` und die Restliste von `L3` ist die Konkatenation der Elemente der Restliste von `L1` und `L2`.

In Prolog können wir dies so ausdrücken:

```
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
```

Dieses Prädikat ist *musterorientiert* definiert: Die erste Klausel ist nur für leere Listen `L1` anwendbar, die zweite nur für nicht-leere Listen. Bei gegebener Liste `L1` passt also nur eine Klausel.

Berechnungsbeispiel:

```
?- append([a,b], [c], [a,b,c]).
     ⊢ (2. Klausel)
?- append([b], [c], [b,c]).
     ⊢ (2. Klausel)
?- append([], [c], [c]).
     ⊢ (1. Klausel)
?- .
```

Anmerkungen:

- Der Suchraum ist ein-elementig.
- Die Berechnung ist vollkommen deterministisch.
- Die Vearbeitungsdauer ist linear abhängig von der Eingabe.

2.3.3 Verwendung von Relationen

Häufig sind die zu lösenden Probleme funktionaler Natur: n Eingabewerten soll ein Ausgabewert zugeordnet werden. Mathematische Schreibweise:

$$\begin{aligned} f : M_1 \times \cdots \times M_n &\rightarrow M \\ (x_1, \ldots, x_n) &\mapsto y \end{aligned}$$

Die Implementierung von Funktionen ist in Prolog in Form von Relationen möglich: die Relation $f(X_1, X_2, \ldots, X_n, Y)$ ist genau dann erfüllt, wenn Y der Ergebniswert bei Eingabe von $X_1, \ldots, X_n$ von f ist.

Die Definition dieser Relation erfolgt allerdings durch Klauseln, nicht durch einen funktionalen Ausdruck! Dies hat wichtige Konsequenzen, denn in Prolog kann man diese Relation auf verschiedene Arten verwenden.

Benutzung als Funktion: x_i sind feste Werte, und für das Ergebnis `Y` setzen wir eine Variable ein:

```
?- f(x1,...,xn,Y).
Y = y
```

Genauso gut kann man die Definition aber auch als Umkehrfunktion bzw. -relation verwenden, in dem wir den „Ergebniswert“ vorgeben, d.h. nun ist y ein fester Wert und $\texttt{X}_i$ sind Variablen:

```
?- f(X1,...,Xn,y).
X1 = x1
...
Xn = xn
```

Konsequenz: Programmiert man eine Funktion in Prolog, so hat man auch die Umkehrfunktion bzw. -relation zur Verfügung.

Die Anwendung des obigen `append` als Funktion sieht so aus:

```
?- append([1,2],[3,4],L).
L = [1,2,3,4]
```

Und als Umkehrfunktion lässt sich `append` wie folgt verwenden:

```
?- append(X,[3,4],[1,2,3,4]).
X = [1,2]
?- append([a], Y, [a,b,c]).
y = [b,c]
```

Wir können es sogar als Umkehrrelation benutzen, um z.B. eine Liste zu zerlegen:

```
?- append(X,Y,[1,2]).
X = []
Y = [1,2] ;
X = [1]
Y = [2] ;
X = [1,2]
Y = [] ;
no
```

Diese Flexibilität lässt sich ausnutzen, um neue Funktionen und Relationen zu definieren:
Anhängen eines Elementes an eine Liste:

```
anhang(L, E, LundE) :- append(L, [E], LundE).
```

Letztes Element einer Liste:

```
letztes(L,E) :- append(_,[E],L).  % letztes Element einer Liste
```

Ist ein Element in einer Liste enthalten?

```
member(E,L) :- append(L1,[E|L2],L).  % Element einer Liste
```

Streichen eines Elementes aus einer Liste:

```
streiche(L1,E,L2) :-
   append(Xs, [E|Ys], L1),
   append(Xs, Ys, L2).
```

Ist eine Liste Teil einer anderen?

```
teilliste(T,L) :-
    append(T1,TL2,L),
    append(T,L2,TL2).
```

Wir merken uns also für die Logikprogrammierung:

- Denke in Relationen (Beziehungen) statt in Funktionen!
- Alle Parameter sind gleichberechtigt (keine Ein-/Ausgabeparameter)!
- Nutze vorhandene Prädikate! Achte bei neuen Prädikaten auf Allgemeinheit bzw. andere Anwendungen.

2.4 Programmieren mit Constraints

Die ursprüngliche Motivation für die Einführung der Constraint-Programmierung war die unvollständige Arithmetik, die in Prolog vorhanden ist. Aus diesem Grund schauen wir uns zunächst diese an.

2.4.1 Arithmetik in Prolog

Da Prolog eine universelle Programmiersprache ist, kann man natürlich auch in Prolog mit arithmetischen Ausdrücken rechnen. Ein *arithmetischer Ausdruck Arithmetik (Prolog)* ist eine Struktur mit Zahlen und Funktoren wie beispielsweise `+ - * /` oder `mod`. Vordefiniert ist zum Beispiel das Prädikat `is(X,Y)`, wobei `is` auch Infixoperator ist. "`X is Y`" ist gültig oder beweisbar, falls

1. `Y` ein (zum Zeitpunkt des Beweises) variablenfreier arithmetischer Ausdruck ist, und
2. `X=Z` gilt, wenn `Z` der ausgerechnete Wert von `Y` ist.

Beispiele:

```
?- 16 is 5*3+1.
```

```
yes
?- X is 5*3+1.
X = 16
?- 2+1 is 2+1.
no
```

Arithmetische Vergleichsprädikate:

Bei arithmetischen Vergleichen in Prolog werden beide Argumente vor dem Vergleich mit `is` ausgewertet:

Prädikat	Bedeutung
`X =:= Y`	Wertgleichheit
`X =\= Y`	Wertungleichheit
`X < Y`	kleiner
`X > Y`	größer
`X >= Y`	größer oder gleich
`X =< Y`	kleiner oder gleich

Eine Definition der Fakultätsfunktion in Prolog sieht also wie folgt aus:

```
fak(0,1).
fak(N,F) :- N > 0,
            N1 is N-1,
            fak(N1, F1), % Reihenfolge wichtig!
            F is F1 * N.
```

Bei der Verwendung der Arithmetik in Prolog ist allerdings zu beachten, dass das `is`-Prädikat partiell ist: Ist bei "`X is Y`" das `Y` kein variablenfreier arithmetischer Ausdruck, dann wird die Berechnung mit einer Fehlermeldung abgebrochen. Daher ist die Reihenfolge bei der Verwendung von `is` wichtig:

```
?- X=2, Y is 3+X. % links-rechts-Auswertung
Y = 5
X = 2
?- Y is 3 + X, X = 2.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Die Arithmetik in Prolog ist also logisch unvollständig:

```
?- 5 is 3+2.
yes
?- X is 3+2.
X = 5
?- 5 is 3+X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Eine globale Konsequenz der Benutzung dieser Arithmetik ist, dass Prolog-Programme mit Arithmetik häufig nur noch in bestimmten Modi ausführbar sind. Eine mögliche Lösung für dieses Problem ist die nachfolgend diskutierte Constraint-Programmierung (*Constraint Logic Programming*, CLP).

2.4.2 Constraint-Programmierung mit Zahlen

Als Verbesserung des obigen Problems der einfachen Prolog-Arithmetik kann man spezielle Verfahren zur Lösung arithmetischer (Un-)Gleichungen in das Prolog-System integrieren: das Gaußsche Eliminationsverfahren für Gleichungen und das Simplexverfahren für Ungleichungen.

Arithmetische Contraints sind Gleichungen und Ungleichungen zwischen arithmetischen Ausdrücken. Die Erweiterung von Logiksprachen um arithmetische (und auch andere Arten von) Constraints wird *Constraint Logic Programming (CLP)* genannt. Dieses ist zwar kein Standard in Prolog, aber in vielen Systemen realisiert. In SICStus-Prolog oder SWI-Prolog (Version 5.6.x) wird es einfach als Bibliothek dazugeladen:

```
:- use_module(library(clpr)).
```

Alle arithmetischen Constraints werden in geschweifte Klammern eingeschlossen:

```
?- {8 = Y + 3.5}.
Y = 4.5

?- {Y =< 3, 2 + X >= 0, Y - 5 = X}.
X = -2.0,
Y = 3.0
```

Beispiel: Schaltkreisanalyse

Das Ziel ist die Analyse von Spannung und Strom in elektrischen Schaltkreisen. Als erstes müssen wir Schaltkreise als Prolog-Objekte darstellen:

- `wider(R)`: Widerstand vom Wert `R`
- `reihe(S1,S2)`: Reihenschaltung der Schaltkreise `S1` und `S2`
- `parallel(S1,S2)`: Parallelschaltung der Schaltkreise `S1` und `S2`

Die Analyse implementieren wir als Relation `sk(S,V,I)`: Diese steht für einen Schaltkreis `S` mit Spannung `V` und Stromdurchfluss `I`. Die Definition dieses Prädikats erfolgt durch jeweils eine Klausel für jede Art des Schaltkreises, d.h. bei Widerständen Anwendung des Ohmschen Gesetzes und bei Parallel- und Reihenschaltung Anwendung der Kirchhoffschen Gesetze:

```
:- use_module(library(clpr)).
```

```
% OHMsches Gesetz
sk(wider(R),V,I) :- {V = I * R}.

% KIRCHHOFFsche Gesetze
sk(reihe(S1,S2),V,I) :-
    {I = I1, I = I2, V = V1 + V2},
    sk(S1,V1,I1),
    sk(S2,V2,I2).

sk(parallel(S1,S2),V,I) :-
    {I = I1 + I2, V = V1, V = V2},
    sk(S1,V1,I1),
    sk(S2,V2,I2).
```

Nun können wir die Stromstärke bei einer Reihenschaltung von Widerständen berechnen:

```
?- sk(reihe(wider(180),wider(470)), 5, I).
I = 0.00769
```

Das System kann uns auch die Relation zwischen Widerstand und Spannung in einer Schaltung ausgeben:

```
?- sk(reihe(wider(R),reihe(wider(R),wider(R))), V, 5).
{ V = 15.0 * R }
```

Beispiel: Hypothekenberechnung

Für die Beschreibung aller nötigen Zusammenhänge beim Rechnen mit Hypotheken verwenden wir folgende Parameter:

- `P`: Kapital
- `T`: Laufzeit in Monaten
- `IR`: monatlicher Zinssatz
- `B`: Restbetrag
- `MP`: monatliche Rückzahlung

Die Zusammenhänge lassen sich nun wie folgt in CLP ausdrücken:

```
mortgage(P,T,IR,B,MP) :- % Laufzeit maximal ein Monat
  {T >= 0, T =< 1, B = P * (1 + T * IR) - T * MP}.
mortgage(P,I,IR,B,MP) :- % Laufzeit mehr als ein Monat
  {T > 1},
  mortgage(P * (1 + IR) - MP, T - 1, IR, B, MP).
```

Jetzt kann das Prolog-System für uns die monatliche Rückzahlung einer Hypothek bei einer gegebenen Laufzeit ausrechnen:

```
?- mortgage(100000, 180, 0.01, 0, MP).
MP = 1200.17
```

Oder wir fragen das System, wie lange wir eine Hypothek zurückzahlen müssen, wenn wir die monatliche Rückzahlung festhalten:

```
?- mortgage(100000, T, 0.01, 0, 1400).
T = 125.901
```

Es kann uns sogar die Relation zwischen dem aufgenommenen Kapital, der monatlichen Rückzahlung und dem Restbetrag ausgeben:

```
?- mortgage(P, 180, 0.01, B, MP).
{ P = 0.166783 * B + 83.3217 * MP }
```

CLP ist also eine Erweiterung der Logikprogrammierung. Es ersetzt Terme durch Contraint-Strukturen, Datentypen mit festgelegter Bedeutung, und enthält Lösungsalgorithmen für diese Constraints.

Ein konkretes Beispiel ist CLP($\mathcal{R}$) für die reellen Zahlen:

- Struktur: Terme, reelle Zahlen und arithmetische Funktionen
- Contraints: Gleichungen und Ungleichungen mit arithmetischen Ausdrücken
- Lösungsalgorithmen: Termunifikation, Gauß'sche Elimination und Simplexverfahren

Weitere Constraint-Strukturen existieren für:

- Boolesche Ausdrücke `(A and (B or C))`: relevant für Hardwareentwurf und -verifikation
- unendliche zyklische Bäume
- Listen
- *endliche Bereiche*, welche zahlreiche Anwendungen Operations Research finden: für Planungsaufgaben wie zum Beispiel die Maschinenplanung für die Fertigung, die Containerbeladung, die Flughafenabfertigung, und andere

Da die Constraint-Struktur der endlichen endliche Bereiche (finite domains) für die Praxis die wichtigsten Anwendungen hat, betrachten wir die Sprache CLP(FD) nun genauer.

2.4.3 Constraint-Programmierung über endlichen Bereichen

CLP(FD) ist eine Erweiterung der Logikprogrammierung mit Constraints über endlichen Bereichen (finite domains):

- Struktur: endliche Mengen/Bereiche, dargestellt durch eine endliche Menge ganzer Zahlen

- Elementare Contraints: Gleichungen, Ungleichungen, Elementbeziehungen
- Constraints: logische Verknüpfungen zwischen Constraints
- Lösungsalgorithmen: OR-Methoden zur Konsistenzprüfung (Knoten-, Kantenkonsistenz), d.h. es ist nicht sichergestellt, dass die Constraints immer erfüllbar sind, da ein solcher Test sehr aufwändig (NP-vollständig) wäre. Aus diesem Grund erfolgt die konkrete Überprüfung einzelner Lösungen durch Aufzählen (Prinzip: "constrain-and-generate")
- Anwendungen: Lösen schwieriger kombinatorischer Probleme, z.B. Stundenplanerstellung, Personalplanung, Fertigungsplanung etc.

  Industrielle Anwendungen im Bereich Planungs-/Optimierungsprobleme:
  - Personalplanung (z.B. Lufthansa)
  - Flottenplanung (z.B. SNCF, EDF)
  - Produktionsplanung (z.B. Renault)
  - Container-Verladung (z.B. Hongkong)
  - Netzwerke (z.B. Gebäudeverkabelung)
  - Krisenmanagementsysteme (Albertville'92)
  - Planungen in großtechnischen Anlagen (Chemie, Öl, Energie)

Der Constraint-Löser für FD-Probleme nimmt nur Konsistenzprüfungen vor: Bei der Gleichung `X = Y` wird zum Beispiel geprüft, ob die Wertebereiche von `X` und `Y` nicht disjunkt sind. Aus diesem Grund sieht die allgemeine Vorgehensweise bei der CLP(FD)-Programmierung wie folgt aus:

1. Definiere den Wertebereich der FD-Variablen.
2. Beschreibe die Constraints, die diese Variablen erfüllen müssen.
3. Zähle die Werte im Wertebereich auf, d.h. belege FD-Variablen mit ihren konkreten Werten.

Der dritte Schritt ist hierbei wegen der Unvollständigkeit des Lösers notwendig. Trotz dieser Unvollständigkeit erhalten wir eine gute Einschränkung des Wertebereichs: So werden die Constraints

```
X in 1..4, Y in 3..6, X = Y
```

zusammengefasst zu "`X in 3..4, Y in 3..4`", und die Constraints

```
X in 1..4, Y in 3..6, Z in 4..10, X = Y, Y = Z
```

ergeben "`X = 4, Y = 4, Z = 4`".

Aus diesem Grund werden in Schritt 3 in der Regel nur noch wenige Lösungen wirklich ausgetestet (im Gegensatz zum naiven "generate-and-test").

Die FD-Constraints in Prolog...

- sind kein Standard, aber häufig als Bibliotheken vorhanden

- haben bei verschiedenen Systemen einen unterschiedlichen Umfang ($\rightsquigarrow$ in entsprechende Handbücher schauen)
- können in SICStus-Prolog oder SWI-Prolog hinzugeladen werden durch

  ```
  :- use_module(library(clpfd)).
  ```

- enthalten viele elementare Constraints, die mit einem `#` beginnen:
  - `X #= Y` für Gleichheit
  - `X #\= Y` für Ungleichheit
  - `X #> Y` für größer als
  - `X #< Y`, `X #>= Y`, `X #=< Y`,...
- und noch viel mehr (einige werden weiter unten im Beispiel erläutert)

Beispiel: Krypto-arithmetisches Puzzle

Gesucht ist eine Zuordnung von Buchstaben zu Ziffern, so dass verschiedene Buchstaben verschiedenen Ziffern entsprechen und die folgende Rechnung stimmt:

```
  SEND
+ MORE
------
 MONEY
```

Mit SICStus-Prolog und CLP(FD) finden wir mit folgendem Programm eine Lösung:

```
:- use_module(library(clpfd)).

puzzle(L) :-
    L = [S,E,N,D,M,O,R,Y],
    domain(L,0,9),   % Wertebereich festlegen
    S #> 0, M #> 0,  % Constraints festlegen
    all_different(L),
                  1000 * S + 100 * E + 10 * N + D
                + 1000 * M + 100 * O + 10 * R + E
    #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y,
    labeling([], L). % Aufzählung der Variablenwerte (Instantiierung)
```

Eine Lösung können wir dann so berechnen:

```
?- puzzle([S,E,N,D,M,O,R,Y]).
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2
```

Da es noch keinen Standard für CLP-Sprachen gibt, benutzen unterschiedliche Systeme verschiedene Namen für die Basisconstraints. Daher geben wir im folgenden einige

Hinweise zur **CLP(FD)-Programmierung in SICStus-Prolog und SWI-Prolog**:

- Vor der Benutzung müssen die entsprechenden Bibliotheken geladen werden (s.o.).
- Zur Festlegung bzw. Einschränkung der Wertebereiche von Variablen gibt es zwei Constraints:
  - `domain(Vs, Min, Max)`:[1] Der Wertebereich aller Variablen in der Variablenliste `Vs` muss zwischen `Min` und `Max` (einschließlich) liegen.
  - `X in Min..Max`: Der Wertebereich der Variablen `X` muss zwischen `Min` und `Max` (einschließlich) liegen. Statt "`Min..Max`" sind auch eine Reihe anderer *Intervallausdrücke* erlaubt, wie z.B. Wertebereiche anderer Variablen, Durchschnitt, Vereinigung etc. (vgl. SICStus-Prolog-Manual).
- Neben elementaren Constraints (z.B. Gleichheit, s.o.) gibt es eine Reihe komplexer **kombinatorischer Constraints** , wie z.B. `all_different(Vs)`: die Werte aller Variablen in der Variablenliste `Vs` müssen paarweise verschieden sein. Z.B. ist `all_different([X,Y,Z])` äquivalent zu

  ```
  X #\= Y, Y #\= Z, X #\= Z
  ```

  allerdings wird `all_different([X,Y,Z])` anders abgearbeitet als die einzelnen Constraints, so dass der Constraint-Solver den Suchbereich mit `all_different` stärker einschränken kann.
- Es gibt eine Reihe vordefinierter Prädikate zum Aufzählen der Werte von FD-Variablen:

  ```
  labeling([], Vs)
  ```

  belegt nacheinander die Variablen in der Variablenliste `Vs` mit ihren Werten. Das erste Argument erlaubt dabei, zusätzliche Steuerungsoptionen anzugeben, womit man die Reihenfolge der Aufzählung beeinflussen kann (vgl. SICStus-Prolog-Manual).

  ```
  ?- X in 3..4, Y in 4..5, labeling([], [X,Y]).
  X = 3
  Y = 4 ;
  X = 3
  Y = 5 ;
  X = 4
  Y = 4 ;
  X = 4
  Y = 5 ;
  no
  ```

  Das Aufzählen ist notwendig wegen der Unvollständigkeit des Constraint-Lösers. Z.B. liefert

[1] In SWI-Prolog muss man hierfür "`Vs ins Min..Max`" schreiben.

```
?- X in 4..5, Y in 4..5, Z in 4..5, X #\= Y, Y #\= Z, X #\= Z.
X in 4..5,
Y in 4..5,
Z in 4..5
```

Auf Grund der paarweisen Constraints kann der Constraint-Löser keine weiteren Einschränkungen der Wertebereiche vornehmen. Beim konkreten Testen einzelner Werte wird dann aber festgestellt, dass es insgesamt keine Lösung gibt:

```
?- X in 4..5, Y in 4..5, Z in 4..5, X #\= Y, Y #\= Z, X #\= Z,
   labeling([], [X,Y,Z]).
no
```

Beispiel: 8-Damen-Problem

Das Ziel ist die Platzierung von acht Damen auf einem Schachbrett, so dass keine eine andere nach den Schachregeln schlagen kann. Klar ist: In jeder Spalte muss eine Dame sein; die wichtige Frage ist also: In welcher Zeile steht die k-te Dame?

Wie modellieren das Problem wie folgt: Jede Dame entspricht einer FD-Variablen mit einem Wert im Intervall [1..8], der Zeilennummer.

Wir formulieren folgende Constraints:

- Die Damen befinden sich in paarweise verschiedenen Zeilen, da sie sich sonst schlagen können.
- Die Damen dürfen sich auch in den Diagonalen nicht schlagen können.

Wir verallgemeinern das Problem auf ein $n \times n$-Schachbrett:

```
queens(N,L) :-
    length(L,N),      % L ist Liste der Laenge N,
                      % d.h. L enthaelt N verschiedene Variablen
    domain(L, 1, N),  % Wertebereich jeder Dame: [1..N]
    all_safe(L),      % alle Damen sind sicher
    labeling([], L).

all_safe([]).
all_safe([Q|Qs]) :- safe(Q,Qs,1), all_safe(Qs).

safe(_, [], _).
safe(Q, [Q1|Qs], P) :-
    no_attack(Q,Q1,P),
    P1 #= P + 1,
    safe(Q,Qs,P1).
```

```
% Damen koennen waagerecht und diagonal schlagen
no_attack(Q, Q1, P) :-
    Q #\= Q1,
    Q #\= Q1 + P,  % P ist der Spaltenabstand
    Q #\= Q1 - P.
```

Nun liefert uns das Prolog-System für ein 4×4-Schachbrett zum Beispiel zwei Lösungen:

```
?- queens(4,L).
L = [2,4,1,3] ;
L = [3,1,4,2]
```

Eine mögliche Lösung für ein 8×8-Schachbrett lautet:

```
?- queens(8,L).
L = [1,5,8,6,3,7,2,4]
```

Obwohl es prinzipell $8^8 = 16.777.216$ potentielle Plazierungen für 8 Damen gibt, wird die letzte Lösung auf Grund der Einschränkungen der Bereiche durch die Propagation von Constraints in wenigen Millisekunden berechnet. Dies zeigt die Mächtigkeit der Constraint-Programmierung für komplexe kombinatorische Probleme.

Auch größere Werte für n können mittels FD-Constraints beliebig berechnet werden. Z.B. liefert

```
?- queens(16,L).
L = [1,3,5,2,13,9,14,12,15,6,16,7,4,11,8,10]
```

in wenigen Millisekunden. Bei größeren n macht sich dann schon die Rechenzeit bemerkbar. Z.B. dauert die Berechnung

```
?- queens(24,L).
L = [1,3,5,2,4,9,11,14,18,22,19,23,20,24,10,21,6,8,12,16,13,7,17,15]
```

schon einige Sekunden. Dies kann aber durch eine einfache Verbesserung beschleunigt werden. Das Aufzählungsprädikat

```
labeling([],Vs)
```

lässt im ersten Argument verschiedene Optionen zu, um die Reihenfolge der Aufzählung der Variablen zu beeinflussen. Ohne Angabe werden alle Variablen einfach nacheinander mit Werten aus ihrem Wertebereich belegt. Die Option "`ff`" ("first fail") belegt hingegen zuerst die Variablen mit dem kleinsten noch möglichen Wertebereich. Hierdurch werden i.allg. weniger Werte aufgezählt. Wenn wir also die Definition von "`queens`" abändern zu

```
queens(N,L) :-
       length(L,N),
```

```
domain(L,1,N),
all_safe(L),
labeling([ff],L). % first fail labeling
```

dann dauert die Berechnung einer Lösung für $n = 24$ auch nur noch Millisekunden, und selbst für $n = 100$ erhält man eine Lösung in weniger als einer Sekunde, obwohl es prinzipiell $100^{100} = 10^{200}$ potentielle Platzierungen für 100 Damen gibt! Zum Vergleich: das Universum ist ca. $4,3 * 10^{20}$ Millisekunden alt.

Weitere FD-Constraints

Weil FD-Constraints für viele Probleme verwendet werden können, bieten konkrete Implementierungen zahlreiche weitere spezielle FD-Constraints an, wie z.B. (in SICStus-Prolog)

`sum(Xs,Relation,Value)` Dieses Constraint ist erfüllt, wenn die Summe der Variablen in der Liste `Xs` mit dem Wert `Value` in der FD-Relation `Relation` steht, z.B. ist

```
sum([X,Y,Z],#=,4)
```

für die Belegung `X=1, Y=2, Z=1` erfüllt.

`serialized(Starts,Durations)` Dieses Constraint ist erfüllt, wenn `Starts` eine Variablenliste mit Startzeitpunkten, `Durations` eine Variablenliste mit Zeitdauern ist, so dass die entsprechenden „Aufträge“ sich nicht überlappen. Damit kann man z.B. Schedulingprobleme elegant ausdrücken.

Constraint-Programmierung in anderen Sprachen

Auf Grund der Möglichkeiten, mit Constraints Planungs- und Optimierungsprobleme auf einem hohen Niveau auszudrücken, gibt es auch Ansätze, Constraint-Programmierung außerhalb der Logikprogrammierung zu verwenden, obwohl die Logikprogrammierung die natürlichste Verbindung bildet. Zu diesen Ansätzen gehören z.B.

- spezielle Sprachen (z.B. OPL, Optimization Programming Language, von Pascal Van Hentenryck)
- Bibliotheken zur Nutzung von Constraint-Lösern aus konventionellen Sprachen wie C++ oder Java

2.5 Rechnen in der Logikprogramierung

Rechnen in Prolog entspricht im Wesentlichen dem Beweisen von Aussagen. Aber wie beweist Prolog Aussagen? Um das zu verstehen, betrachten wir zunächst ein einfaches Resolutionsprinzip.

Wir kennen die folgenden Elemente der Logikprogrammierung:

- *Fakten* sind beweisbare Aussagen. Eine Aussage ist die Anwendung eines Prädikates auf Objekte, was manchmal auch als *Literal* bezeichnet wird.
- *Regeln* sind logische Schlussfolgerungen und haben die folgende Semantik: Wenn L `:-` $L_1, ..., L_n$`.` eine Regel ist, und die Literale $L_1, ..., L_n$ beweisbar sind, dann ist auch L beweisbar. Diese Regel wird als *modus ponens* oder auch *Abtrennungsregel* bezeichnet.
- *Anfragen*Anfrage sind zu überprüfende Aussagen mit der folgenden Semantik: Wenn `?-` $L_1, ..., L_n$`.` eine Anfrage ist, dann wird überprüft, ob $L_1, ..., L_n$ mit den gegebenen Fakten und Regeln beweisbar ist.

Dies führt zu der folgenden Idee: Um die Aussage einer Anfrage zu überprüfen, suche eine dazu passende Regel und kehre den modus ponens um zum:

> *Einfaches Resolutionsprinzip:* Reduziere den Beweis des Literals L auf den Beweis der Literale $L_1, ..., L_n$, falls L `:-` $L_1, ..., L_n$ eine Regel ist. Hierbei werden Fakten als Regeln mit leerem Rumpf interpretiert.

Wir betrachten das folgende Beispiel:

```
ehemann(monika, herbert).

mutter(susanne, monika).

vater(susanne, herbert) :-
    ehemann(monika, herbert),
    mutter(susanne, monika).
```

Mittels des einfachen Resolutionsprinzips können wir folgende Ableitung durchführen:

```
?- vater(susanne, herbert).
 ⊢ Regel von vater:
?- ehemann(monika, herbert), mutter(susanne, monika).
 ⊢ Faktum fuer ehemann:
?- mutter(susanne, monika).
 ⊢ Faktum fuer mutter:
?- .
```

> *Kann eine Anfrage in mehreren Schritten mittels des Resolutionsprinzips auf die leere Anfrage reduziert werden, dann ist die Anfrage beweisbar.*

Unifikation

Das Problem ist häufig, dass die Regeln oft nicht „direkt“ passen: So passt zum Beispiel die Regel

```
vater(K,V) :- ehemann(M,V), mutter(K,M).
```

nicht zu der Anfrage

```
?- vater(susanne, herbert).
```

Aber `K` und `V` sind Variablen, wir können also beliebige Objekte einsetzen. Wenn wir also `K` durch `susanne` und `V` durch `herbert` ersetzen, können wir die Regel und damit das Resolutionsprinzip anwenden.

Allgemein können wir die Ersetzung von Variablen in einem Term durch den Begriff einer Substitution präzisieren. Hierbei betrachten wir eine Konstante c auch als Struktur $c()$ ohne Argumente.

Definition 2.1 *Eine* Substitution *ist eine Abbildung* $\sigma : Terme \to Terme$*, die Variablen durch Terme ersetzt, mit folgenden Eigenschaften:*

1. *Für alle Terme* $f(t_1, ..., t_n)$ *gilt:* $\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n))$*.* σ *ist strukturerhaltend oder ein Homomorphismus. Diese Eigenschaft gilt analog für Literale.*
2. *Die Menge* $\{X \mid X$ *ist Variable mit* $\sigma(X) \neq X\}$ *ist endlich.*

Daher ist σ *eindeutig darstellbar als Paarmenge*

$$\{X \mapsto \sigma(X) \mid X \neq \sigma(X), X \text{ ist Variable}\}$$

In unserem Beispiel sieht σ also so aus: $\sigma = \{\texttt{K} \mapsto \texttt{susanne}, \texttt{V} \mapsto \texttt{herbert}\}$. Die Anwendung der Substitution sieht dann so aus:

$$\sigma(\texttt{vater(K,V)}) = \texttt{vater}(\sigma(\texttt{K}),\sigma(\texttt{V}))) = \texttt{vater(susanne,herbert)}$$

Aber wir müssen auch die Variablen nicht nur in Regeln, sondern auch in Anfragen wie

```
?- ehemann(monika, M).
```

ersetzen können. Dazu Bedarf es der *Unifikation*: Terme müssen durch Variablenersetzung gleich gemacht werden. Für die Terme `datum(Tag,Monat, 83)` und `datum(3, M, J)` gibt es mehrere mögliche Substitutionen, die diese gleich machen:
$\sigma_1 = \{Tag \mapsto 3, Monat \mapsto 4, M \mapsto 4, J \mapsto 83\}$

$\sigma_2 = \{Tag \mapsto 3, Monat \mapsto M, J \mapsto 83\}$

Sowohl σ_1 als auch σ_2 machen Terme gleich, σ_1 ist aber spezieller.

Definition 2.2 (Unifikator) *Eine Substitution* σ *heißt* Unifikator *für* t_1 *und* t_2*, falls* $\sigma(t_1) = \sigma(t_2)$*.* t_1 *und* t_2 *heißen dann* unifizierbar*.*
σ *heißt* allgemeinster Unifikator (most general unifier (mgu))*, falls für alle Unifikatoren* σ' *eine Substitution* ϕ *existiert mit* $\sigma' = \phi \circ \sigma$*, wobei die Komposition* $\phi \circ \sigma$ *definiert ist durch* $\phi \circ \sigma(t) = \phi(\sigma(t))$*.*

Es ist wichtig, mit mgus zu rechnen: Wir erhalten weniger Beweise, und müssen also weniger suchen. Es stellt sich also die Frage: Gibt es immer mgus? Und wie kann man diese berechnen?

Die Antwort hat *Robinson* im Jahr 1965 [8] gefunden: es gibt immer mgus für unifizierbare Terme. Für ihre Berechnung definieren wir den Begriff der *Unstimmigkeitsmenge von Termen*:

Definition 2.3 *Sind t, t' Terme, dann ist die* Unstimmigkeitsmenge (disagreement set) *$ds(t, t')$ definiert durch:*

1. *Falls $t = t'$: $ds(t, t') = \emptyset$*
2. *Falls t oder t' Variable und $t \neq t'$: $ds(t, t') = \{t, t'\}$*
3. *Falls $t = f(t_1, ..., t_n)$ und $t' = g(s_1, ..., s_m)$ $(n, m \geq 0)$:*
   - *Falls $f \neq g$ oder $m \neq n$: $ds(t, t') = \{t, t'\}$*
   - *Falls $f = g$ und $m = n$ und $t_i = s_i$ für alle $i < k$ und $t_k \neq s_k$: $ds(t, t') = ds(t_k, s_k)$*

Intuitiv bedeutet diese Definition: $ds(t, t')$ enthält die Teilterme von t und t' an der linkesten Position, an denen t und t' verschieden sind.

Daraus ergibt sich unmittelbar der folgende *Unifikationsalgorithmus*.

Unifikationsalgorithmus:

Eingabe: Terme (Literale) t_0, t_1
Ausgabe: mgu σ für t_0, t_1, falls diese unifizierbar sind, und „fail“ sonst

1. k `:= 0;` σ_0 `:=` $\{\}$
2. Falls $\sigma_k(t_0) = \sigma_k(t_1)$, dann ist σ_k mgu
3. Falls $ds(\sigma_k(t_0), \sigma_k(t_1)) = \{x, t\}$ mit x Variable und x kommt nicht in t vor, dann: σ_{k+1} `:=` $\{x \mapsto t\} \circ \sigma_k$; k `:=` k `+ 1`; gehe zu 2;
   sonst: „fail“

Wir wollen den Algorithmus einmal an einigen Beispielen nachvollziehen:

1. t_0 = `ehemann(monika,M)`, t_1 = `ehemann(F,herbert)`
   - $ds(t_0, t_1) = \{$`F, monika`$\}$
   - $\sigma_1 = \{$`F` $\mapsto$ `monika`$\}$
   - $ds(\sigma_1(t_0), \sigma_1(t_1)) = \{$`M, herbert`$\}$
   - $\sigma_2 = \{$`M` $\mapsto$ `herbert`, `F` $\mapsto$ `monika`$\}$
   - $ds(\sigma_2(t_0), \sigma_2(t_1)) = \emptyset$

   $\Longrightarrow \sigma_2$ ist mgu
2. Das nächste Beispiel zeigt, wie die Unifikation auch fehlschlagen kann:
   t_0 = `equ(f(1),g(X))`, t_1 = `equ(Y,Y)`
   - $ds(t_0, t_1) = \{$`Y, f(1)`$\}$

- $\sigma_1 = \{\texttt{Y} \mapsto \texttt{f(1)}\}$
- $ds(\sigma_1(t_0), \sigma_1(t_1)) = \{\texttt{g(X)}, \texttt{f(1)}\}$

$\Longrightarrow$ nicht unifizierbar

3. Ein letztes Beispiel soll den Sinn der Überprüfung von Schritt 3 zeigen, ob x nicht in t vorkommt:

   $t_0 = \texttt{X}$, $t_1 = \texttt{f(X)}$

   - $ds(t_0, t_1) = \{\texttt{X}, \texttt{f(X)}\}$

   $\Longrightarrow$ nicht unifizierbar, da `X` in `f(X)` vorkommt!

Die Abfrage in Punkt 3 des Algorithmus' heißt auch *Vorkommenstest* (*occur check*) und ist relevant für dessen Korrektheit. Viele Prolog-Systeme verzichten aber aus Gründen der Effizienz auf diesen Test, da er selten erfolgreich ist, d.h. es passiert bei den meisten praktischen Programmen nicht, dass auf Grund des Vorkommenstest eine Unifikation fehlschlägt. Theoretisch kann dies aber zu einer fehlerhaften Unifikation und zur Erzeugung zyklischer Terme führen.
Es gilt folgender Satz:

Satz 2.1 (Unifikationssatz von Robinson [8]) *Seien t_0, t_1 Terme. Sind diese unifizierbar, dann gibt der obige Algorithmus einen mgu für t_0, t_1 aus. Sind sie es nicht, dann gibt er „fail“ aus.*

Beweis: **Terminierung:**

1. Ein Schleifendurchlauf erfolgt nur, falls $ds(\sigma_k(t_0), \sigma_k(t_1))$ mindestens eine Variable enthält.
2. In jedem Schleifendurchlauf wird eine Variable eliminiert
   (d.h. in $\sigma_{k+1}(t_i)$, $i = 0, 1$, kommt x nicht mehr vor).
3. Da in t_0 und t_1 nur endlich viele Variablen vorkommen, gibt es aufgrund von 1. und 2. nur endlich viele Schleifendurchläufe.

$\Longrightarrow$ Algorithmus terminiert immer

Korrektheit:

1. Seien t_0 und t_1 nicht unifizierbar: Falls der Algorithmus in Schritt 2 anhält, dann sind t_0 und t_1 unifizierbar. Da der Algorithmus auf jeden Fall anhält und t_0 und t_1 nicht unifizierbar sind, muss der Algorithmus in Schritt 3 anhalten, d.h. es wird „fail“ ausgegeben.
2. Seien t_0 und t_1 unifizierbar: Sei θ beliebiger Unifikator für t_0 und t_1. Wir zeigen:

   $\forall k \geq 0\ \exists$ Substitution γ_k mit $\theta = \gamma_k \circ \sigma_k$ und es wird im k. Durchlauf nicht „fail“ ausgegeben
   (auf Grund der Terminierung bedeutet dies dann, dass die Ausgabe tatsächlich ein mgu ist)

   Beweis durch Induktion über k:
   $k = 0:$ Sei $\gamma_0 := \theta$. Dann ist $\gamma_0 \circ \sigma_0 = \theta \circ \{\} = \theta$

$k \Rightarrow k+1$: *Induktionsvoraussetzung:* $\theta = \gamma_k \circ \sigma_k$, d.h. γ_k ist Unifikator für $\sigma_k(t_0)$ und $\sigma_k(t_1)$.

Entweder: $\sigma_k(t_0) = \sigma_k(t_1)$: Dann gibt es keinen $k+1$-ten Schleifendurchlauf.

Oder: $\sigma_k(t_0) \neq \sigma_k(t_1)$: Also ist

$$\emptyset \neq ds(\sigma_k(t_0), \sigma_k(t_1)) = \{x, t\}$$

und x kommt nicht in t vor (in allen anderen Fällen wäre $\sigma_k(t_0)$ und $\sigma_k(t_1)$ nicht unifizierbar!). Daraus folgt:

- es wird im $(k+1)$-ten Durchlauf nicht „fail“ ausgegeben
- $\sigma_{k+1} = \{x \mapsto t\} \circ \sigma_k$

Sei nun $\gamma_{k+1} := \gamma_k \backslash \{x \mapsto \gamma_k(x)\}$ (d.h. nehme aus γ_k die Ersetzung für x heraus). Dann gilt:

$$\begin{array}{ll}
\gamma_{k+1} \circ \sigma_{k+1} & \\
= \gamma_{k+1} \circ \{x \mapsto t\} \circ \sigma_k & \\
= \{x \mapsto \gamma_{k+1}(t)\} \circ \gamma_{k+1} \circ \sigma_k & \text{(da } x \text{ in } \gamma_{k+1} \text{ nicht ersetzt wird)} \\
= \{x \mapsto \gamma_k(t)\} \circ \gamma_{k+1} \circ \sigma_k & \text{(da } x \text{ in } t \text{ nicht vorkommt)} \\
= \gamma_k \circ \sigma_k & (\gamma_k(x) = \gamma_k(t) \text{ und Definition von } \gamma_{k+1}) \\
= \theta & \text{(Induktionsvoraussetzung)}
\end{array}$$

Damit ist die Induktionsbehauptung und somit auch der Satz bewiesen.

■

Wir schließen daraus: Unifizierbare Terme haben immer einen allgemeinsten Unifikator. Wir betrachten noch kurz die **Komplexität des Unifikationsalgorithmus**: Der Algorithmus hat im schlechtesten Fall eine exponentielle Laufzeit bezüglich der Größe der Eingabeterme. Dies liegt an exponentiell wachsenden Termen:

Sei $t_0 = p(x_1, \ldots, x_n)$ und $t_1 = p(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}))$
dann:

- $\sigma_1 = \{x_1 \mapsto f(x_0, x_0)\}$
- $\sigma_2 = \{x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0))\} \circ \sigma_1$
- ...

σ_k ersetzt x_k durch einen Term mit $2^k - 1$ `f`-Symbolen. Somit hat der für σ_n notwendige Vorkommenstest eine exponentielle Laufzeit.

Anmerkungen:

1. Auch ohne Vorkommenstest hat der Algorithmus eine exponentielle Laufzeit, da die exponentiell wachsenden Terme aufgebaut werden müssen.
2. Ausweg: Keine explizite Darstellung der Terme, sondern z.B. durch Graphen. Dadurch Laufzeitverbesserungen bis zu linearen Algorithmen (siehe z.B. [3, 5, 6])
3. Exponentielles Wachstum ist in der Praxis äußerst selten
   $\Longrightarrow$ klassischer Algorithmus ausreichend mit „Sharing“ von Variablen

Allgemeines Resolutionsprinzip

Das *allgemeine Resolutionsprinzip* vereinigt Resolution und Unifikation und wird auch als *SLD-Resolution* (**L**inear Resolution with **S**election Function for **D**efinite Clauses) bezeichnet. Hierbei wird durch eine *Selektionsfunktion* festgelegt, welches Literal aus einer Anfrage im nächsten Beweisschritt ausgewählt wird. Mögliche Selektionsregeln sind z.B. FIRST (wähle immer das erste Literal) oder LAST (wähle immer das letzte Literal).

Definition 2.4 (SLD-Resolution) *Gegeben sei eine Selektionsregel und die Anfrage*

`?-` $A_1, \ldots, A_{i-1}, A_i, A_{i-1}, \ldots, A_m$ `.`

wobei die Selektionsregel aus dieser Anfrage das Literal A_i auswählt. Falls

L `:-` $L_1, \ldots, L_n$ `.`

eine Regel (mit neuen Variablen, wobei auch $n = 0$ erlaubt ist) und σ ein allgemeinster Unifikator für A_i und L ist, dann ist die Anfrage

`?-` $\sigma(A_1, \ldots, A_{i-1}, L_1, \ldots, L_n, A_{i-1}, \ldots, A_m)$ `.`

in einem SLD-Resolutionsschritt ableitbar aus der Anfrage und der Regel bezüglich der Selektionsregel. Bezeichnet G die ursprüngliche Anfrage und G' die abgeleitete Anfrage, dann notieren wir diesen Resolutionsschritt auch in der Form $G \vdash_\sigma G'$.

Als Beispiel betrachten wir noch einmal die *Gleichheit in Prolog*. In Prolog-Systemen ist die Klausel

```
=(X,X).
```

vordefiniert, wobei "=" als Infixoperator deklariert ist.
Konsequenz: Die Anfrage

`?-` t_0 `=` t_1 `.`

ist genau dann beweisbar, wenn t_0 und t_1 unifizierbar sind.

Die Forderung, dass eine *Regel mit neuen Variablen* (man nennt dies dann auch *Variante* einer Regel) in einem Resolutionsschritt genommen werden muss, ist gerechtfertigt, weil die Variablen in einer Regel für beliebige Werte stehen und somit beliebig gewählt werden können, ohne dass sich die Bedeutung einer Regel ändert. Zum Beispiel ist die Klausel

```
=(X,X).
```

gleichbedeutend mit der Klausel

```
=(Y,Y).
```

Die Wahl einer Regel mit *neuen* Variablen ist manchmal notwendig, um durch Namenskonflikte verursachte Fehlschläge zu vermeiden. Wenn z.B. die Klausel

```
p(X).
```

gegeben ist (die besagt, dass das Prädikat `p` für jedes Argument beweisbar ist) und wir versuchen, die Anfrage

```
?- p(f(X)).
```

zu beweisen, dann würde ohne neue Regelvariablen kein Resolutionsschritt möglich sein, da die Unifikation von `p(X)` und `p(f(X))` auf Grund des Vorkommenstests fehlschlägt. Wenn wir allerdings den Resolutionsschritt mit der Regelvariante

```
p(Y).
```

durchführen, ist dies erfolgreich möglich, weil `p(Y)` und `p(f(X))` unifizierbar sind.

Als weiteres Beispiel für das allgemeine Resolutionsprinzip betrachten wir das folgende Programm:

```
vater(hans,peter).
vater(peter,frank).
grossvater(X,Z) :- vater(X,Y), vater(Y,Z).
```

Anfrage: `?- grossvater(hans,E).`
Beweis nach dem Resolutionsprinzip:

```
?- grossvater(hans,E).

   ⊢ {X↦hans, Z↦E}:

?- vater(hans,Y), vater(Y,E).

   ⊢ {Y↦peter}:

?- vater(peter,E).

   ⊢ {E↦frank}:

?- .
```

Antwort: `E = frank`

Auswertungstrategie und SLD-Baum

Wir haben bisher nur gezeigt, was einzelne SLD-Schritte sind und wie diese möglicherweise zu einer erfolgreichen Ableitung zusammengesetzt werden können. Es gibt allerdings für eine Anfrage eventuell viele unterschiedliche Ableitungen, von denen manche erfolgreich und manche nicht erfolgreich sind. Eine genaue Auswertungsstrategie sollte also festlegen, wie man diese unterschiedlichen Ableitungen konstruiert oder durchsucht.

Um einen Überblick über die unterschiedlichen SLD-Schritte und SLD-Ableitungen für eine Anfrage zu erhalten, fassen wir diese in einer Baumstruktur zusammen, die auch als *SLD-Baum* bezeichnet wird.

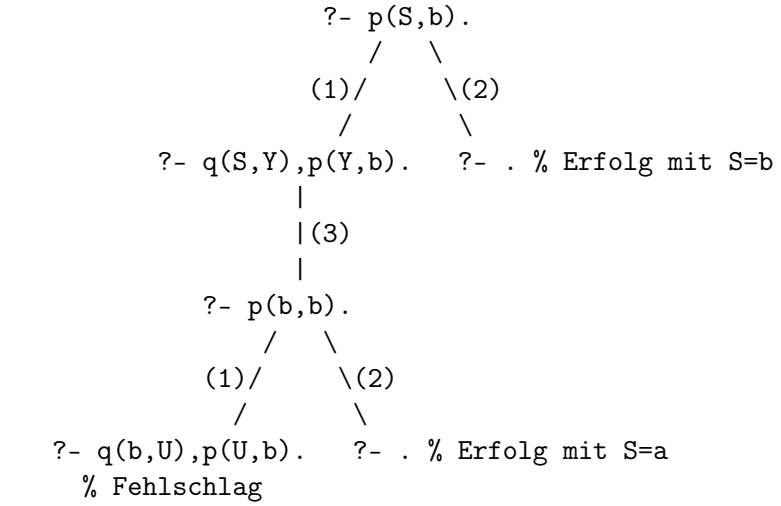
Definition 2.5 (SLD-Baum) *Gegeben sei ein Programm P und eine Anfrage G. Ein* SLD-Baum *für G ist ein Baum, dessen Knoten mit Anfragen (hierbei ist auch die leere Anfrage ohne Literale erlaubt) markiert sind und für den gilt:*

1. *Die Wurzel ist mit G markiert.*
2. *Ist S ein Knoten, der mit G_0 markiert ist und sind $G_1, \ldots, G_n$ alle Anfragen, die aus G_0 mittels einer Klausel aus P (und der gegebenen Selektionsregel) ableitbar sind, dann hat S genau die Kinder $S_1, \ldots, S_n$, die jeweils mit $G_1, \ldots, G_n$ markiert sind.*
3. *Eine leere Anfrage ohne Literale hat keinen Kindknoten.*

Als Beispiel betrachten wir das folgende Programm (wobei wir die Regeln nummerieren):

```
(1)   p(X,Z) :- q(X,Y), p(Y,Z).
(2)   p(X,X).
(3)   q(a,b).
```

Dann sieht ein SLD-Baum für dieses Programm und die Anfrage "`?- p(S,b).`" wie folgt aus:



Aus diesem SLD-Baum ist ersichtlich, dass es drei verschiedene SLD-Ableitungen für die ursprüngliche Anfrage gibt.

Eine Auswertungsstrategie kann damit als Regel zum Durchsuchen eines SLD-Baumes interpretiert werden. Eine *sichere Strategie*, d.h. eine Strategie, die immer eine Lösung findet, wenn diese existiert, wäre ein Breitendurchlauf durch den SLD-Baum. Ein Nachteil des Breitendurchlaufs ist der hohe Speicheraufwand, da man sich immer die Knoten der jeweiligen Ebene merken muss. Aus diesem Grund verzichtet Prolog auf eine sichere Strategie und verwendet eine effizientere Strategie, die aber *unvollständig* ist, d.h. in manchen Fällen eine erfolgreiche SLD-Ableitung nicht findet.

Auswertungstrategie von Prolog

Die Auswertungsstrategie von Prolog verwendet die Selektionsstrategie FIRST, d.h. Prolog wählt immer das linkeste Literal zum Beweis, und durchläuft den SLD-Baum mit einem Tiefendurchlauf. Natürlich wird der SLD-Baum nicht real aufgebaut, sondern Prolog verwendet eine Auswertungsmethode, die letztendlich einem Tiefendurchlauf im SLD-Baum entspricht. Diese wird mittels folgender *Backtracking-Methode* realisiert:

1. Die Klauseln haben eine Reihenfolge, und zwar die, in der sie im Programm definiert werden.
2. In einem Resolutionsschritt wird die *erste passende* Klausel für das linke Literal gewählt. Bei Sackgassen werden die letzen Schritte rückgängig gemacht und die nächste Alternative probiert (*backtrack*).
3. Bei der Anwendung einer Regel werden durch die Unifikation die Variablen durch Terme ersetzt. Dann wird eine Variable an einen Term *gebunden* oder auch *instantiiert*.

Wir veranschaulichen die *Auswerungsstrategie von Prolog* an folgendem Beispiel:

```
p(a).
p(b).
q(b).

?- p(X), q(X).
    ⊢ {X ↦ a}
?- q(a).
    % Sackgasse. Rücksetzen (d.h. 2. Klausel für p):
    ⊢ {X ↦ b}
?- q(b).
    ⊢ {}
?- .
```

Das folgende Programm zeigt jedoch die Probleme der Backtracking-Methode, d.h. die Unvollständigkeit der Auswertungsstrategie von Prolog:

```
p :- p.
p.

?- p.
    ⊢ ?- p.
    ⊢ ?- p.
    ...
```

Das System landet in einer Endlosschleife, statt `yes` zurückzugeben. Prolog ist also unvollständig als „Theorembeweiser“.

Wir betrachten ein weiteres Beispiel, dass die Relevanz der Klauselreihenfolge zeigt:

```
last([K|R], E) :- last(R,E).
last([E], E).

?- last(L, 3).
    ⊢ {L ↦ [K1|R1]}
?- last(R1, 3).
    ⊢ {R1 ↦ [K2|R2]}
?- last(R2, 3).
...
```

Diese Ableitung endet also nicht.

Man kann sich daher folgende Empfehlung merken: Klauseln für Spezialfälle sollten stets *vor* allgemeineren Klauseln angegeben werden! Denn vertauschen wir im letzten Beispiel die Klauseln, so terminiert die Anfrage sofort:

```
?- last(L, 3).
    ⊢ {L ↦ [3]}
?- .
```

2.6 Der „Cut“-Operator

Mit „Cut“ (`!`) kann man das Backtracking teilweise unterdrücken, d.h. konzeptuell kann man damit Teile des SLD-Baumes „abschneiden“. Dies möchte man manchmal aus verschiedenen Gründen machen:

1. Effizienz (Speicherplatz und Laufzeit)
2. Kennzeichnen von Funktionen
3. Verhinderung von Laufzeitfehlern (z.B. bei `is`)

In Prolog kann `!` anstelle von Literalen im Regelrumpf stehen:

```
p :- q, !, r.
```

Operational bedeutet dies: Wird diese Regel zum Beweis von `p` benutzt, dann gilt:

1. Falls `q` nicht beweisbar ist: wähle nächste Regel für `p`.
2. Falls `q` beweisbar ist: `p` ist nur beweisbar, falls `r` beweisbar ist. Mit anderen Worten, es wird kein Alternativbeweis für `q` und keine andere Regel für `p` ausprobiert.

Wir betrachten das folgende Beispiel:

```
ja :- ab(X), !, X = b.
ja.

ab(a).
ab(b).

?- ja.
```

Rein logisch ist die Anfrage auf zwei Arten beweisbar. Operational wird jedoch die erste Regel angewandt, `X` an `a` gebunden, es folgt ein Cut, dann ein Fehlschlag weil `X = b` nicht bewiesen werden kann, und schließlich wird keine Alternative mehr ausprobiert. Damit ist die Anfrage in Prolog nicht beweisbar. Man sollte also Cut nur vorsichtig verwenden!

Häufig verwendet man einen Cut zur Fallunterscheidung:

```
p :- q, !, r.
p :- s.
```

Dies entspricht so etwas wie

```
p :- if q then r else s.
```

Tatsächlich gibt es genau dafür eine spezielle Syntax in Prolog:

```
p :- q -> r; s.
```

Als Beispiel wollen wir einmal die Maximumrelation implementieren:

```
max(X,Y,Z) :- X >= Y, !, Z = X.
max(X,Y,Z) :- Z = Y.  % rein logisch unsinnig!
```

Alternativ könnte man diese auch so aufschreiben:

```
max(X,Y,Z) :- X >= Y -> Z = X ; Z = Y.
```

2.7 Negation

Das folgende Beispiel soll zeigen, warum man häufig die Negation benötigt:

```
geschwister(S, P) :-  mutter(S, M), mutter(P, M).
```

Hier fehlt allerdings noch eine Bedingung wie "`nicht S = P`", sonst wäre jeder, der eine Mutter hat, Geschwister von sich selbst.
In Prolog ist die *Negation als Fehlschlag (NAF)* implementiert: "`\+ p`" ist beweisbar, falls alle Beweise für `p` fehlschlagen.

```
?- \+ monika = susanne.
yes
```

Man sollte aber beachten, dass die Negation als Fehlschlag nicht mit der prädikatenlogischen Negation übereinstimmt. Betrachtet man das Beispiel

```
p :- \+ p.
```

so ergibt sich bei logischer Negation:

$$\neg p \Rightarrow p \;\equiv\; \neg(\neg p) \vee p \;\equiv\; p \vee p \;\equiv\; p$$

d.h. `p` ist wahr. Ein Prolog-System gerät aber bei dem Versuch, `p` zu beweisen, in eine Endlosschleife. Clark [2] hat deshalb der Negation in Prolog eine etwas andere Bedeutung gegeben (deren Details wir hier überspringen) und die NAF-Regel (*negation as finite failure*) als operationales Prinzip der Negation eingeführt:

> Falls alle Beweise für `p` endlich und fehlgeschlagen sind, dann ist `\+ p` beweisbar.

Die NAF-Regel ist effektiv implementierbar: Zum Beispiel können wir

```
p :- \+ q.
```

formulieren als (hierbei ist `fail` ein nie beweisbares Prädikat):

```
p :- q, !, fail.
p.
```

Es gibt allerdings noch ein weiteres Problem: Die Negation in Prolog ist inkorrekt, falls das negierte Literal Variablen enthält.

```
p(a,a).
p(a,b).

?- \+ p(b,b).
yes
?- \+ p(X,b).
no
```

Rein logisch hätte bei der letzten Anfrage die Antwort `{X↦b}` berechnet werden sollen, aber wegen der NAF-Regel werden Variablen in einer Negation nie gebunden!

Die Konsequenz daraus ist: Beim Beweis von "`\+` p" darf p keine Variablen enthalten! Nun schauen wir uns noch einmal auf unser Verwandschaftsbeispiel an:

```
geschwister(S,P) :-
    mutter(S, M),
    mutter(P, M),
    \+ S = P.  % hier ok, da S und P immer gebunden sind
```

Es ist also zu empfehlen, negierte Literale in den Regeln möglichst weit rechts anzugeben, damit sichergestellt ist, dass das negative Literal variablenfrei ist, wenn es bewiesen werden soll.

Ein Ausweg, der in manchen Prolog-Systemen angeboten wird, ist die **verzögerte Negation:**

Idee: Verzögere Beweis negativer Literale, bis sie keine Variablen enthalten, um eine logisch sichere Negation zu erhalten.
Falls dies so realisiert wird, könnten wir unser Verwandschaftsbeispiel auch so schreiben:

```
geschwister(S,P) :- \+ S=P, mutter(S,M), mutter(P,M).
```

Ein Beweis könnte dann wie folgt ablaufen:

```
?- geschwister(angelika,P).
  ⊢
?- \+ angelika=P, mutter(angelika,M), mutter(P,M).
  ⊢ verzögere Auswertung des 1. Literals und beweise 2. Literal
?- \+ angelika=P, mutter(P,christine).
  ⊢
?- \+ angelika=herbert.
  ⊢
?-.
```

Die Verzögerung der Auswertung von Literalen ist theoretisch gerechtfertigt, weil die Auswahl von Literalen in der Logikprogrammierung prinzipiell beliebig erfolgten kann.
Implementieren kann man die Verzögerung der Auswertung recht einfach, falls das Prolog-Systemen auch *Koroutining* enthält, was bei vielen heutigen Prolog-Systemen der Fall ist. So kann man z.B. in SICStus-Prolog oder SWI-Prolog statt "`\+ p`" besser

```
when(ground(p),\+ p)
```

schreiben. Hierdurch wird die Auswertung von "`\+ p`" verzögert, bis `p` variablenfrei ist.

3 Java Generics

3.1 Einführung

Seit der Version 5.0 (2004 veröffentlicht) bietet Java die Möglichkeit der *generischen Programmierung*: Klassen und Methoden können mit Typen parametrisiert werden. Hierdurch eröffnen sich ähnliche Möglichkeiten wie mit Templates in C++.

Als einfaches Beispiel betrachten wir eine sehr einfache Containerklasse, welche keinen oder einen Wert speichern kann. In Java könnte das z.B. wie folgt definiert werden:

```
public class Maybe {
    private Object value;
    private boolean empty;

    public Maybe() { empty = true; }

    public Maybe(Object v) {
        value = v;
        empty = false;
    }

    public boolean isEmpty() {
        return empty;
    }

    public Object fromMaybe() {
        if (!empty) {
            return value;
        }
        throw new MaybeException();
    }
}
```

Wir nehmen an, `MaybeException` sei abgeleitet von `RuntimeException`.

Die Verwendung der Klasse könnte dann wie folgt aussehen:

```
Maybe mv = new Maybe(new Integer(42));

...
```

```
if (!mv.isEmpty()) {
    Integer n = (Integer)mv.fromMaybe();
}
```

Der Zugriff auf das gespeicherte Objekt der Containerklasse erfordert also jedes Mal explizite Typumwandlungen (type casts). Es ist *keine Typsicherheit* gegeben: Im Falle eines falschen Casts tritt eine `ClassCastException` zur Laufzeit auf.

Beachte: Die Definition der Klasse `Maybe` ist völlig unabhängig vom Typ des gespeicherten Wertes; der Typ von `value` ist `Object`. Wir können also beliebige Typen erlauben und in der Definition der Klasse von diesen abstrahieren: Das Übergeben des Typs als Parameter nennt man *parametrisierter Polymorphismus*.

Syntaktisch markieren wir den Typparameter durch spitze Klammern und benutzen den Parameternamen an Stelle von `Object`. Die Klassendefinition lautet dann wie folgt:

```
public class Maybe<T> {
    private T value;
    private boolean empty;

    public Maybe() { empty = true; }

    public Maybe(T v) {
        value = v;
        empty = false;
    }

    public boolean isEmpty() {
        return empty;
    }

    public T fromMaybe() {
        if (!empty) {
            return value;
        }
        throw new MaybeException();
    }
}
```

Die Verwendung der Klasse sieht dann so aus:

```
Maybe<Integer> mv = new Maybe<Integer>(new Integer(42));

...
```

```
if (!mv.isEmpty()) {
    Integer n = mv.fromMaybe();
}
```

Es sind also keine expliziten Typcasts mehr erforderlich, und ein Ausdruck wie

```
mv = new Maybe<Integer>(mv);
```

liefert nun bereits eine Typfehlermeldung zur Compilezeit.
Natürlich sind auch mehrere Typparameter erlaubt:

```
public class Pair<A, B> {
    private A first;
    private B second;

    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public A first() { return first; }
    public B second() { return second; }
}
```

3.2 Zusammenspiel mit Vererbung

Es ist auch möglich, Typparameter so einzuschränken, dass dafür nur Klassen eingesetzt werden können, die bestimmte Methoden zur Verfügung stellen. Als Beispiel wollen wir unsere Klasse `Maybe` so erweitern, dass sie auch das Interface Comparable implementiert:

```
public class Maybe<T extends Comparable<T>>
             implements Comparable<Maybe<T>> {

    ...

    public int compareTo(Maybe<T> o) {
        if (empty) {
            if (o.isEmpty()) {
                return 0;
            } else {
                return -1;
            }
        } else {
```

```
            if (o.isEmpty()) {
                return 1;
            } else {
                return value.compareTo(o.fromMaybe());
            }
        }
    }
}
```

Hierbei wird sowohl für Interfaces als auch für echte Vererbung das Schlüsselwort `extends` verwendet. Wir können auch mehrere Einschränkungen an Typvariablen aufzählen. Für drei Typparameter sieht das aus wie folgt:

```
<T extends A<T>, S, U extends B<T,S>>
```

Hier muss `T` die Methoden von `A<T>` und `U` die Methoden von `B<T,S>` zur Verfügung stellen.

3.3 Wildcards

Die Klasse `Integer` ist Unterklasse der Klasse `Number`. Somit sollte doch auch der folgende Code möglich sein:

```
Maybe<Integer> mi = new Maybe<Integer>(new Integer(42));
Maybe<Number> mn = mi;
```

Das Typsystem erlaubt dies aber nicht, weil dies festlegt, dass `Maybe<Integer>` **kein** Untertyp von `Maybe<Number>` ist! Das mögliche Problem wird deutlich, wenn wir zur Klasse `Maybe` noch eine Setter-Methode hinzufügen:

```
public void set(T v) {
    empty = false;
    value = v;
}
```

Dann wäre auch folgendes möglich:

```
mn.set(new Float(42));
Integer i = mi.fromMaybe();
```

Da `mi` und `mn` die gleichen Objekte bezeichnen, würde dieser Code bei der Ausführung zu einem Typfehler führen. Aus diesem Grund erlaubt das Typsystem den obigen Code nicht.

Dennoch benötigt man manchmal einen Obertyp für polymorphe Klassen, also einen Typ, der alle anderen Typen umfasst: So beschreibt

```
Maybe<?> mx = mi;
```

einen `Maybe` von unbekanntem Typ. Hierbei wird "`?`" auch als *Wildcard*-Typ bezeichnet. `Maybe<?>` repräsentiert alle anderen `Maybe`-Instantiierungen, z.B. `Maybe<Integer>` oder `Maybe<Maybe<Object>>`.

Damit können aber nur Methoden verwendet werden, die für jeden Typ passen, also z.B.

```
Object o = mx.fromMaybe();
```

Können wir auch die Methode `set` für `mx` aufrufen? Hierzu benötigen wir einen Wert, der zu allen Typen gehört: `null`.

```
mx.set(null);
```

Andere `set`-Aufrufe sind nicht möglich.

Der Wildcard-Typ "`?`" ist aber leider oft nicht ausreichend, z.B. wenn man in einer Collection auch verschiedene Untertypen speichert, wie GUI-Elemente. Dann kann man sog. *beschränkte Wildcards* (*bounded wildcards*) verwenden:

`<? extends A>` steht für *alle* Untertypen des Typs `A` *(Kovarianz)*
`<? super A>` steht für *alle* Obertypen von `A` *(Kontravarianz)*

Es folgen einige Beispiele.

Ausdruck	erlaubt?	Begründung
`Maybe<? extends Number> mn = mi;`		
`Integer i = mn.fromMaybe();`	nein!	`?` könnte auch `Float` sein
`Number n = mn.fromMaybe();`	ja	
`mn.set(n);`	nein!	`?` könnte spezieller als `Number` sein
`mn.set(new Integer(42));`	nein!	`?` könnte auch `Float` sein
`mn.set(null);`	ja	
`Maybe<? super Integer> mx = mi;`		
`Integer i = mx.fromMaybe();`	nein!	`?` könnte auch `Number` oder `Object` sein
`Number n = mx.fromMaybe();`	nein!	`?` könnte auch `Object` sein
`Object o = mx.fromMaybe();`	ja	
`mx.set(o);`	nein!	`?` könnte auch `Number` sein
`mx.set(n);`	nein!	`n` könnte auch vom Typ `Float` sein
`mx.set(i);`	ja	

Tabelle 3.1: Ausdrücke mit Wildcards

Wir können also aus einem `Maybe<? extends A>` Objekte vom Typ `A` herausholen, und in einen `Maybe<? super A>` Objekte vom Typ `A` hereinstecken.

Im folgenden betrachten wir noch ein paar Erweiterungen der Klasse `Maybe`, welche teilweise erst durch die Verwendung von Typparametern möglich werden. Gegeben sei folgendes Interface für einen Wrapper, der es erlaubt, Funktionen als Objekte zu speichern:

```
interface Fun<Arg, Res> {
    public Res apply(Arg a);
}
```

Für konkrete Funktionen erstellen wir einfach Klassen, die obiges Interface implementieren. Nun können wir unsere Klasse `Maybe` wie folgt erweitern:

```
public T fromMaybeWithDefault(T def) {
    return (empty ? def : value);
}

public void map(Fun<T,T> f) {
    if (!empty) {
        value = f.apply(value);
    }
}
```

Die Verwendung unserer erweiterten Klasse könnte wie folgt aussehen:

```
Maybe<Integer> mi = new Maybe<Integer>(new Integer(42));
mi.map(new Fun<Integer, Integer>() {
    public Integer apply(Integer n) {
        return n + 1;
    }
});
```

Die Konzepte der Funktionen `fromMaybeWithDefault(T)` und `map(Fun<T, T>)` lassen sich sogar vereinen:

```
public <resT> resT applyWithDefault(resT def, Fun<T,resT> f) {
    if (empty) {
        return def;
    } else {
        return f.apply(value);
    }
}
```

Die Verwendung der neuen Methode könnte so aussehen:

```
Boolean b = mi.applyWithDefault(true,
    new Fun<Integer, Boolean) {
        public Boolean apply(Integer n) {
            return (n >= 42);
        }
    });
```

Beachte: Die Verwendung von Funktionsobjekten `Fun` wird erst durch Typparameter sinnvoll, da sonst keinerlei Typsicherheit gewährleistet werden kann, ohne viele spezielle Interfaces für einzelne Verwendungen zu definieren.

Bei der Übersetzung von Klassen mit Typparametern wird für jede generische Klasse genau eine ungetypte Variante erzeugt, bei der statt der Typvariablen die speziellsten möglichen Typen verwendet werden. Im Gegensatz dazu werden die Templates in C++ für jede Verwendung spezialisiert, d.h. es werden Klassen `MaybeInteger`, `MaybeNumber` oder `MaybeBoolean` erzeugt.

Vorteile der Java-Lösung:

- weniger Code
- Codeerzeugung unabhängig von Klassenverwendung

Nachteil der Java-Lösung:

- keine Typinstanziierung durch primitive Typen wie `int` oder `float`

4 Nebenläufige Programmierung in Java

4.1 Allgemeine Vorbemerkungen

4.1.1 Motivation

Wozu benötigen wir nebenläufige Programmierung in Java oder auch in anderen Programmiersprachen? Häufig möchten wir, dass eine Anwendung mehrere Aufgaben übernehmen soll. Gleichzeitig soll die *Reaktivität* der Anwendung erhalten bleiben. Beispiele für solche Anwendungen sind:

- GUIs
- Betriebssystemroutinen
- verteilte Applikationen (Webserver, Chat, ...)

4.1.2 Lösung

Dies erreichen wir mittels *Nebenläufigkeit (Concurrency)*. Durch die Verwendung von Threads bzw. Prozessen können einzelne Aufgaben einer Anwendung unabhängig von anderen Aufgaben programmiert und ausgeführt werden.

4.1.3 Weitere Begriffe

Parallelität. Durch die parallele Ausführung mehrerer Prozesse soll eine schnellere Ausführung erreicht werden (High-Performance-Computing).

Verteiltes System. Mehrere Komponenten in einem Netzwerk arbeiten zusammen an einem Problem. Meist gibt es dabei eine verteilte Aufgabenstellung, manchmal nutzt man verteilte Systeme auch zur Parallelisierung.

4.1.4 Arten von Multitasking

Wir sprechen von *Multitasking*, wenn die Prozessorzeit durch einen Scheduler auf die nebenläufigen Threads bzw. Prozesse verteilt wird. Wir unterscheiden dabei zwei Arten von Multitasking:

1. *Kooperatives Multitasking*: Ein Thread rechnet so lange, bis er die Kontrolle wieder abgibt (z.B. mit `yield()`) oder auf Nachrichten wartet (`suspend()`). In Java finden wir dies bei den sog. *green threads*.

2. *Präemptives Multitasking*: Der Scheduler kann Tasks auch die Kontrolle entziehen. Hierbei genießen wir oft mehr Programmierkomfort, da wir uns nicht so viele Gedanken machen müssen, wo wir überall die Kontrolle wieder abgeben sollten.

4.1.5 Interprozesskommunikation und Synchronisation

Neben der Generierung von Threads bzw. Prozessen ist auch die Kommunikation zwischen diesen wichtig. Sie geschieht meist über geteilten Speicher bzw. Variablen.
Wir betrachten folgendes Beispiel in Pseudocode:

```
int i = 0;

par
    { i = i + 1; }
    { i = i * 2; }
end par;

print(i);
```

Nebenläufigkeit macht Programme nicht-deterministisch, d.h. es können je nach Scheduling unterschiedliche Ergebnisse herauskommen. So kann obiges Programm die Ausgaben 1, 2 oder sogar 0 erzeugen.
Die Frage, die sich stellt, ist: Welche Aktionen werden wirklich atomar ausgeführt? Durch Übersetzung des Programms in Byte- oder Maschinencode können sich folgende Instruktionen ergeben:

1. `i = i + 1` → `LOAD i; INC; STORE i;`
2. `i = i * 2` → `LOAD i; SHIFTL; STORE i;`

Dann führt der folgende Ablauf zur Ausgabe 0:

```
(2) LOAD i;

(1) LOAD i;
(1) INC;
(1) STORE i;

(2) SHIFTL;
(2) STORE i;
```

Wir benötigen also Synchronisation zur Gewährleistung der atomaren Ausführung bestimmter Codeabschnitte, welche nebenläufig auf gleichen Ressourcen arbeiten. Solche Codeabschnitte nennen wir *kritische Bereiche*.

4.1.6 Synchronisation mit Semaphoren

Ein bekanntes Konzept zur Synchronisation nebenläufiger Threads oder Prozesse geht auf Dijkstra aus dem Jahre 1968 zurück. Dijkstra entwickelte einen abstrakten Datentyp mit dem Ziel, die atomare (ununterbrochene) Ausführung bestimmter Programmabschnitte zu garantieren. Diese *Semaphore* stellen zwei atomare Operationen zur Verfügung:

```
p(s) {
       if   s >= 1
       then s = s - 1;
       else trage ausfuehrenden Thread in Warteliste zu s ein
            und suspendiere ihn;
}

v(s) {
       if   Warteliste zu s nicht leer
       then wecke ersten Prozess der Warteliste wieder auf
       else s = s + 1;
}
```

Dabei steht `p(s)` für passieren oder *passeer*, `v(s)` steht für verlassen oder *verlaat*.
Nun können wir bei unserem obigen Programm die Ausgabe 0 wie folgt verhindern:

```
int i = 0;
Semaphore s = 1;

par
    { p(s); i = i + 1; v(s); }
    { p(s); i = i * 2; v(s); }
end par;
```

Der Initialwert der Semaphore bestimmt dabei die maximale Anzahl der Prozesse im kritischen Bereich. Meist finden wir hier den Wert 1, solche Semaphore nennen wir auch *binäre Semaphore*.
Eine andere Anwendung von Semaphoren ist das *Producer-Consumer-Problem*: n Producer erzeugen Waren, die von m Consumern verbraucht werden. Eine einfache Lösung für dieses Problem verwendet einen unbeschränkten Buffer:

```
Semaphore num = 0;
```

Code für den Producer:

```
while (true) {
    product = produce();
    push(product, buffer);
```

```
    v(num);
}
```

Code für den Consumer:

```
while (true) {
    p(num);
    prod = pull(buffer);
    consume(prod);
}
```

Was nun noch fehlt ist die Synchronisation auf `buffer`. Dies kann durch Hinzufügen einer weiteren Semaphore realisiert werden:

```
Semaphore num = 0;
Semaphore bufferAccess = 1;
```

Code für den Producer:

```
while (true) {
    product = produce();
    p(bufferAccess);
    push(product, buffer);
    v(bufferAccess);
    v(num);
}
```

Code für den Consumer:

```
while (true) {
    p(num);
    p(bufferAccess);
    prod = pull(buffer);
    v(bufferAccess);
    consume(prod);
}
```

Die Semaphore bringen jedoch auch einige Nachteile mit sich. Der Code mit Semaphoren wirkt schnell unstrukturiert und unübersichtlich. Außerdem können wir Semaphore nicht kompositionell verwenden: So kann der einfache Code `p(s); p(s);` auf einer binären Semaphore `s` bereits einen *Deadlock* erzeugen.

Eine Verbesserung bieten hier die *Monitore*, die wir bereits aus der Vorlesung „Betriebssysteme“ kennen. In der Tat verwendet Java einen Mechanismus ähnlich dieser Monitore zur Synchronisierung.

4.1.7 Dining Philosophers

Das Problem der dinierenden Philosophen (*dining philosophers* mit n Philosophen lässt sich wie folgt mit Hilfe von Semaphoren modellieren:

```
Semaphore[n] sticks = [1, 1, ..., 1];
```

Code für Philosoph `i`:

```
while (true) {
    think();

    p(stick[(i - 1) mod n]);
    p(stick[i]);

    eat();

    v(stick[(i - 1) mod n]);
    v(stick[i]);
}
```

Dabei tritt jedoch ein Deadlock auf, falls alle Philosophen gleichzeitig ihr linkes Stäbchen nehmen. Diesen Deadlock können wir durch Zurücklegen vermeiden:

```
while (true) {
    think();

    p(stick[(i - 1) mod n]);

    if
        (l(stick[i]) == 0)
    then
        v(stick[(i - 1) mod n]);
        continue;
    else
        p(stick[i]);

    eat();

    v(stick[(i - 1) mod n]);
    v(stick[i]);
}
```

Hier bezeichnet `l(s)` eine Lookup-Funktion, die uns den Wert einer Semaphore `s` zurückgibt.

Das Programm hat nun noch einen Livelock, d.h. einzelne Philosophen können verhungern. Diesen möchten wir hier nicht weiter behandeln.

4.2 Threads in Java

4.2.1 Die Klasse `Thread`

Die API von Java bietet im Package `java.lang` eine Klasse `Thread` an. Eigene Threads können von dieser abgeleitet werden. Der Code, der dann nebenläufig ausgeführt werden soll, wird in die Methode `run()` geschrieben. Nachdem wir einen neuen Thread einfach mit Hilfe seines Konstruktors erzeugt haben, können wir ihn zur nebenläufigen Ausführung mit der Methode `start()` starten.
Wir betrachten als Beispiel folgenden einfachen Thread:

```
public class ConcurrentPrint extends Thread {
    private String s;

    public ConcurrentPrint(String s) {
        this.s = s;
    }

    public void run() {
        while (true) {
            System.out.print(s + " ");
        }
    }

    public static void main(String[] args) {
        new ConcurrentPrint("a").start();
        new ConcurrentPrint("b").start();
    }
}
```

Der Ablauf des obigen Programms kann zu vielen möglichen Ausgaben führen:

```
a a b b a a b b ...
a a a b b ...
a b a a a b a a b b ...
a a a a a a a a a a ...
```

Letztere ist dann garantiert, wenn kooperatives Scheduling vorliegt.

4.2.2 Das Interface `Runnable`

Java bietet keine Mehrfachvererbung. Deshalb ist eine Erweiterung der Klasse `Thread` häufig ungünstig. Eine Alternative bietet das Interface `Runnable`:

```
public class ConcurrentPrint implements Runnable {
    private String s;

    public ConcurrentPrint(String s) {
        this.s = s;
    }

    public void run() {
        while (true) {
            System.out.print(s + " ");
        }
    }

    public static void main(String[] args) {
        Runnable aThread = new ConcurrentPrint("a");
        Runnable bThread = new ConcurrentPrint("b");

        new Thread(aThread).start();
        new Thread(bThread).start();
    }
}
```

Beachte: Innerhalb der obigen Implementierung von `ConcurrentPrint` liefert `this` kein Objekt vom Typ `Thread` mehr. Das aktuelle Thread-Objekt erreicht man dann über die statische Methode `Thread.currentThread()`.

4.2.3 Eigenschaften von Thread-Objekten

Jedes Thread-Objekt in Java hat eine Reihe von Eigenschaften:

- *Name.* Beispiele für Namen von Threads sind „main-Thread“, „Thread-0“ oder „Thread-1“. Zugriff auf den Namen eines Threads erfolgt über die Methoden `getName()` und `setName(String)`. Man verwendet sie in der Regel zum Debuggen.
- *Zustand.* Jeder Thread befindet stets in einem bestimmten Zustand. Eine Übersicht dieser Zustände und der Zustandsübergänge ist in Abbildung 4.1 dargestellt.

  Ein Thread-Objekt bleibt auch im Zustand *terminiert* noch solange erhalten, bis alle Referenzen auf ihn verworfen wurden.
- *Dämon.* Ein Thread kann mit Hilfe des Aufrufs `setDaemon(true)` *vor* Aufruf der `start()`-Methode als Hintergrundthread deklariert werden. Die JVM terminiert,

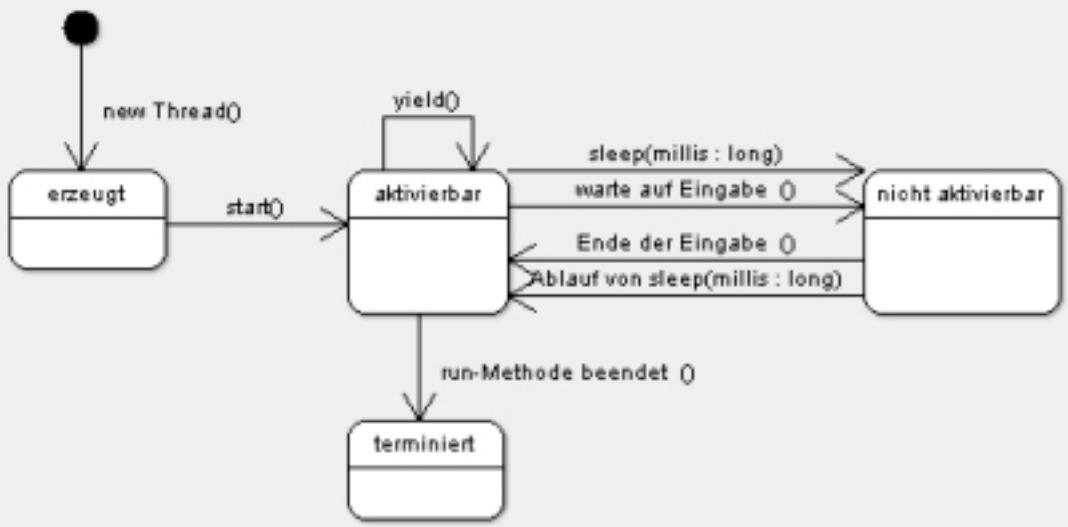


Abbildung 4.1: Zustände von Threads

sobald nur noch Dämonenthreads laufen. Beispiele für solche Threads sind AWT-Threads oder der Garbage Collector.

- *Priorität.* Jeder Thread in Java hat eine bestimmte Priorität. Die genaue Staffelung ist plattformspezifisch.
- *Threadgruppen.* Threads können zur gleichzeitigen Behandlung auch in Gruppen eingeteilt werden.
- *Methode* `sleep(long)`. Lässt den Thread die angegebene Zeit schlafen. Ein Aufruf dieser Methode kann eine `InterruptedException` werfen, welche aufgefangen werden muss.

4.2.4 Synchronisation von Threads

Zur Synchronisation von Threads bietet Java ein Monitor-ähnliches Konzept, das es erlaubt, Locks auf Objekten zu setzen und wieder freizugeben.

Die Methoden eines Threads können in Java als `synchronized` deklariert werden. In allen synchronisierten Methoden eines Objektes darf sich dann maximal ein Thread zur Zeit befinden. Hierzu zählen auch Berechnungen, die in einer synchronisierten Methode aufgerufen werden und auch unsynchronisierte Methoden des gleichen Objektes. Dabei wird eine synchronisierte Methode nicht durch einen Aufruf von `sleep(long)` oder `yield()` verlassen.

Ferner besitzt jedes Objekt ein eigenes *Lock* . Beim Versuch der Ausführung einer Methode, die als `synchronized` deklariert ist, unterscheiden wir drei Fälle:

1. Ist das Lock freigegeben, so nimmt der Thread es sich.
2. Besitzt der Thread das Lock bereits, so macht er weiter.

3. Ansonsten wird der Thread suspendiert.

Das Lock wird wieder freigegeben, falls die Methode verlassen wird, in der es genommen wurde.

Im Vergleich zu Semaphoren wirkt der Ansatz von Java strukturierter, man kann kein unlock vergessen. Dennoch ist er weniger flexibel.

4.2.5 Die Beispielklasse `Account`

Ein einfaches Beispiel soll die Verwendung von `synchronized`-Methoden veranschaulichen. Wir betrachten eine Implementierung einer Klasse für ein Bankkonto:

```
public class Account {
    private double balance;

    public Account(double initialDeposit) {
        balance = initialDeposit;
    }

    public synchronized double getBalance() {
        return balance;
    }

    public synchronized void deposit(double amount) {
        balance += amount;
    }
}
```

Wir möchten die Klasse nun wie folgt verwenden:

```
Account a = new Account(300);
⋮
a.deposit(100); // nebenläufig, erster Thread
⋮
a.deposit(100); // nebenläufig, zweiter Thread
⋮
System.out.println(a.getBalance());
```

Die Aufrufe der Methode `deposit(double)` sollen dabei nebenläufig von verschiedenen Threads aus erfolgen. Ohne das Schlüsselwort `synchronized` wäre die Ausgabe `500`, aber auch die Ausgabe `400` denkbar (vgl. dazu Abschnitt 4.1.4. Interprozesskommunikation und Synchronisation). Mit Anwendung des Schlüsselwortes ist eine Ausgabe von `500` garantiert.

4.2.6 Genauere Betrachtung von `synchronized`

Vererbte synchronisierte Methoden müssen nicht zwingend wieder synchronisiert sein. Wenn man solche Methoden überschreibt, so kann man das Schlüsselwort `synchronized` auch weglassen. Dies bezeichnet man als *verfeinerte Implementierung*. Die Methode der Oberklasse bleibt dabei `synchronized`. Andererseits können unsynchronisierte Methoden auch durch synchronisierte überschrieben werden.

Klassenmethoden, die als synchronisiert deklariert werden
(`static synchronized`) haben keine Wechselwirkung mit synchronisierten Objektmethoden. Die Klasse hat also ein eigenes Lock.

Man kann auch einzelne Anweisungen synchronisieren:

```
synchronized (expr) block
```

Dabei muss `expr` zu einem Objekt auswerten, dessen Lock dann zur Synchronisation verwendet wird. Streng genommen sind synchronisierte Methoden also nur syntaktischer Zucker: So steht die Methodendeklaration

```
synchronized A m(args) block
```

eigentlich für

```
A m(args) {
    synchronized (this) block
}
```

Einzelne Anweisungen zu synchronisieren ist sinnvoll, um weniger Code synchronisieren bzw. sequentialisieren zu müssen:

```
private double state;

public void calc() {
    double res;

    // do some really expensive computation
    ...

    // save the result to an instance variable
    synchronized (this) {
        state = res;
    }
}
```

Synchronisierung auf einzelne Anweisungen ist auch nützlich, um auf andere Objekte zu synchronisienen. Wir betrachten als Beispiel eine einfache Implementierung einer syn-

chronisierten Collection:

```
class Store {
    public synchronized boolean hasSpace() {
        ...
    }

    public synchronized void insert(int i)
        throws NoSpaceAvailableException {

        ...
    }
}
```

Wir möchten diese Collection nun wie folgt verwenden:

```
Store s = new Store();

...

if (s.hasSpace()) {
    s.insert(42);
}
```

Dies führt jedoch zu Problemen, da wir nicht ausschließen können, dass zwischen den Aufrufen von `hasSpace()` und `insert(int)` ein Re-Schedule geschieht. Da sich das Definieren spezieller Methoden für solche Fälle oft als unpraktikabel herausstellt, verwenden wir die obige Collection also besser folgendermaßen:

```
synchronized(s) {
    if (s.hasSpace()) {
        s.insert(42);
    }
}
```

4.2.7 Unterscheidung der Synchronisation im OO-Kontext

Wir bezeichnen synchronisierte Methoden und synchronisierte Anweisungen in Objektmethoden als *server-side synchronisation*. Synchronisation der Aufrufe eines Objektes bezeichnen wir als *client-side synchronisation*.

Aus Effizienzgründen werden Objekte der Java-API, insbesondere Collections, nicht mehr synchronisiert. Für Collections stehen aber synchronisierte Versionen über Wrapper wie `synchronizedCollection`, `synchronizedSet`, `synchronizedSortedSet`, `synchronizedList`, `synchronizedMap` oder

`synchronizedSortedMap` zur Verfügung.

Sicheres Kopieren einer Liste in ein Array kann nun also auf zwei verschiedene Weisen bewerkstelligt werden: Als erstes legen wir eine Instanz einer synchronisierten Liste an:

```
List<Integer> unsyncList = new List<Integer>();

// fill the list
...

List<Integer> list = Collections.synchronizedList(unsyncList);
```

Nun können wir diese Liste entweder mit der einfachen Zeile

```
Integer[] a = list.toArray(new Integer[0]);
```

oder über

```
Integer[] b;

synchronized (list) {
    b = new Integer[list.size()];
    list.toArray(b);
}
```

in ein Array kopieren. Bei der zweiten, zweizeiligen Variante ist die Synchronisierung auf die Liste unabdingbar: Wir greifen in beiden Zeilen auf die Collection zu, und wir können nicht garantieren, dass nicht ein anderer Thread die Collection zwischenzeitig verändert. Dies ist ein klassisches Beispiel für client-side synchronisation.

4.2.8 Kommunikation zwischen Threads

Threads kommunizieren über geteilte Objekte miteinander. Wie finden wir nun heraus, wann eine Variable einen Wert enthält? Dafür gibt es mehrere Lösungsmöglichkeiten.

Die erste Möglichkeit ist die Anzeige des Veränderns einer Komponente des Objektes, beispielsweise durch Setzen eines Flags (`boolean`). Dies hat jedoch den Nachteil, dass das Prüfen auf das Flag zu busy waiting führt. Deshalb suspendiert man mittels einer Methode des Objektes `wait()`, und weckt es mittels `notify()` oder `notifyAll()` wieder auf.
Das sieht zum Beispiel so aus:

```
public class C {
    private int state = 0;

    public synchronized void printNewState()
```

```
        throws InterruptedException {

        wait();
        System.out.println(state);
    }

    public synchronized void setValue(int v) {
        state = v;
        notify();
        System.out.println("value set");
    }
}
```

Zwei Threads führen nun die Methodenaufrufe `printNewState()` und `setValue(42)` nebenläufig aus. Nun ist die *einzig* mögliche Ausgabe

```
value set
42
```

Falls der Aufruf von `wait()` erst kommt, nachdem die Methode `setValue(int)` vom ersten Thread schon verlassen wurde, so führt dies nur zur Ausgabe `value set`.

Die Methoden `wait()`, `notify()` und `notifyAll()` dürfen nur innerhalb von `synchronized`-Methoden oder -Blöcken benutzt werden und sind Methoden für das gesperrte Objekt. Sie haben dabei folgende Semantik:

- `wait()` legt den ausführenden Thread schlafen und gibt das Lock des Objektes wieder frei.
- `notify()` erweckt *einen* schlafenden Thread des Objekts und fährt mit der eigenen Berechnung fort. Der erweckte Thread bewirbt sich nun um das Lock. Wenn kein Thread schläft, dann geht das `notify()` verloren.
- `notifyAll()` tut das gleiche wie `notify()`, nur für alle Threads, die für dieses Objekt mit `wait()` schlafen gelegt wurden.

Dabei ist zu beachten dass diese 3 Methoden nur auf Objekten aufgerufen werden dürfen, deren Lock man vorher erhalten hat. Der Aufruf muss daher in einer `synchronized`-Methode bzw. in einem `synchronized`-Block erfolgen, ansonsten wird zur Laufzeit eine `IllegalMonitorStateException` geworfen.
Wir möchten nun ein Programm schreiben, das alle Veränderungen des Zustands ausgibt:

```
...

private boolean modified = false; // zur Anzeige der Zustandsaenderung

...
```

```
public synchronized void printNewState() {
    while (true) {
        if (!modified) {
            wait();
        }

        System.out.println(state);
        modified = false;
    }
}

public synchronized void setValue(int v) {
    state = v;
    notify();
    modified = true;
    System.out.println("value set");
}
```

Ein Thread führt nun `printNewState()` aus, andere Threads verändern den Zustand mittels `setValue(int)`. Dies führt zu einem Problem: Bei mehreren setzenden Threads kann die Ausgabe einzelner Zwischenzustände verloren gehen. Also muss auch `setValue(int)` ggf. warten und wieder aufgeweckt werden:

```
public synchronized void printNewState() {
    while (true) {
        if (!modified) {
            wait();
        }

        System.out.println(state);
        modified = false;
        notify();
    }
}

public synchronized void setValue(int v) {
    if (modified) {
        wait();
    }

    state = v;
    notify();
    modified = true;
    System.out.println("value set");
```

```
}
```

Nun ist es aber nicht gewährleistet, dass der Aufruf von `notify()` in der Methode `setValue(int)` den `printNewState`-Thread aufweckt! In Java lösen wir dieses Probem mit Hilfe von `notifyAll()` und nehmen dabei ein wenig busy waiting in Kauf:

```
public synchronized void printNewState() {
    while (true) {
        while (!modified) {
            wait();
        }

        System.out.println(state);
        modified = false;
        notify();
    }
}

public synchronized void setValue(int v) {
    while (modified) {
        wait();
    }

    state = v;
    notifyAll();
    modified = true;
    System.out.println("value set");
}
```

Die Methode `wait()` ist in Java außerdem mehrmals überladen:

- `wait(long)` unterbricht die Ausführung für die angegebene Anzahl an Millisekunden.
- `wait(long, int)` unterbricht die Ausführung für die angegebende Anzahl an Milli- und Nanosekunden.

Anmerkung: Es ist dringend davon abzuraten, die Korrektheit des Programms auf diese Überladungen zu stützen!
Die Aufrufe `wait(0)`, `wait(0, 0)` und `wait()` führen alle dazu, dass der Thread solange wartet, bis er wieder aufgeweckt wird.

4.2.9 Fallstudie: Einelementiger Puffer

Ein einelementiger Puffer ist günstig zur Kommunikation zwischen Threads. Da der Puffer einelementig ist, kann er nur leer oder voll sein. In einen leeren Puffer kann über eine

Methode `put` ein Wert geschrieben werden, aus einem vollen Puffer kann mittels `take` der Wert entfernt werden. `take` suspendiert auf einem leeren Puffer, `put` suspendiert auf einem vollen Puffer.

```
public class Buffer1<T> {
    private T content;
    private boolean empty;


    public Buffer1() {
        empty = true;
    }

    public Buffer1(T content) {
        this.content = content;
        empty = false;
    }


    public synchronized T take() throws InterruptedException {
        while (empty) {
            wait();
        }

        empty = true;
        notifyAll();

        return content;
    }

    public synchronized void put(T o) throws InterruptedException {
        while (!empty) {
            wait();
        }

        empty = false;
        notifyAll();
        content = o;
    }

    public synchronized boolean isEmpty() {
        return empty;
    }
}
```

Unschön an der obigen Lösung ist, dass zuviele Threads erweckt werden, d.h. es werden durch `notifyAll()` immer sowohl alle lesenden als auch alle schreibenden Threads erweckt, von denen dann die meisten sofort wieder schlafen gelegt werden. Können wir Threads auch gezielt erwecken? Ja! Dazu verwenden wir spezielle Objekte zur Synchronisation der taker und putter.

```
public class Buffer1<T> {
    private T content;
    private boolean empty;
    private Object r = new Object();
    private Object w = new Object();


    public Buffer1() {
        empty = true;
    }

    public Buffer1(T content) {
        this.content = content;
        empty = false;
    }


    public T take() throws InterruptedException {
        synchronized (r) {
            while (empty) {
                        r.wait();
            }

            synchronized (w) {
                        empty = true;
                        w.notify();

                        return content;
            }
        }
    }

    public void put(T o) throws InterruptedException {
        synchronized(w) {
            while (!empty) {
                        w.wait();
            }
```

```
            synchronized (r) {
                        empty = false;
                        r.notify();
                        content = o;
            }
        }
    }

    public boolean isEmpty() {
          return empty;
    }
}
```

Hier ist das `while` wichtig! Ein anderer Thread, der die Methode von außen betritt, könnte sonst einen wartenden (und gerade aufgeweckten) Thread ansonsten noch überholen!

4.2.10 Beenden von Threads

Java bietet mehrere Möglichkeiten, Threads zu beenden:

1. Beenden der `run()`-Methode
2. Abbruch der `run()`-Methode
3. Aufruf der `destroy()`-Methode (deprecated, z. T. nicht mehr implementiert)
4. Dämonthread- und Programmende

Bei 1. und 2. werden alle Locks freigegeben. Bei 3. werden Locks nicht freigegeben, was diese Methode unkontrollierbar macht. Aus diesem Grund sollte diese Methode auch nicht benutzt werden. Bei 4. sind die Locks egal.

Java sieht zusätzlich eine Möglichkeit zum Unterbrechen von Threads über *Interrupts* vor. Jeder Thread hat ein Flag, welches Interrupts anzeigt.

Die Methode `interrupt()` sendet einen Interrupt an einen Thread, das Flag wird gesetzt. Falls der Thread aufgrund eines Aufrufs von `sleep()` oder `wait()` schläft, wird er erweckt und eine `InterruptedException` geworfen.

```
synchronized (o) {
    ...

    try {
        ...

        o.wait();

        ...
    } catch (InterruptedException e) {
```

```
            ...
        }
    }
```

Bei Interrupt nach dem Aufruf von `wait()` wird der `catch`-Block erst betreten, wenn der Thread das Lock auf das Objekt `o` des umschließenden `synchronized`-Blocks wieder erlangt hat!

Im Gegensatz dazu wird bei der Suspension durch `synchronized` der Thread nicht erweckt, sondern nur das Flag gesetzt.

Die Methode `public boolean isInterrupted()` testet, ob ein Thread Interrupts erhalten hat. `public static boolean interrupted()` testet den aktuellen Thread auf Interrupt und löscht das Interrupted-Flag.

Falls man also in einer `synchronized`-Methode auf Interrupts reagieren möchte, ist dies wie folgt möglich:

```
synchronized void m(...) {
    ...

    if (Thread.currentThread().isInterrupted()) {
        throw new InterruptedException();
    }
}
```

Falls eine `InterruptedException` aufgefangen wird, wird das Flag ebenfalls gelöscht. Dann muss man das Flag erneut setzten!

4.3 Verteilte Programmierung in Java

Java bietet als Abstraktion der Netzwerkkommunikation die *Remote Method Invocation (RMI)* an. Hiermit können Remote-Objekte auf anderen Rechnern verwendet werden, als wären es lokale Objekte. Argumente und Ergebnisse müssen hierbei in Byte-Folgen umgewandelt werden.

4.3.1 Serialisierung von Daten

Die Serialisierung eines Objektes o liefert eine Byte-Sequenz, die Deserialisierung der Byte-Sequenz liefert ein neues Objekt o'. Beide Objekte sollen bezüglich ihres Verhaltens gleich sein, haben aber unterschiedliche Objektidentitäten.

Die (De-)Serialisierung erfolgt rekursiv. Enthaltene Objekte müssen also auch (de-)serialisiert werden. Um festzulegen, dass ein Objekt serialisiert werden kann, bietet Java das Interface `Serializable`:

```
public class C implements java.io.Serializable { ...  }
```

Dieses Interface enthält keine Methoden und ist somit nur eine „Markierungsschnittstelle“, die festlegt, dass Objekte dieser Klasse (de)serialisiert werden können. Die eigentliche (De-)Serialisierung erfolgt mit den Methoden `readObject` und `writeObject` (s.u.).

Bei der Serialisierung können bestimmte zeit- oder sicherheitskritische Teile eines Objektes mit Hilfe des Schlüsselwortes `transient` ausgeblendet werden:

```
protected transient String password;
```

Transiente Werte sollten nach dem Deserialisieren explizit gesetzt (z.B. Timer) bzw. nicht verwendet werden (z.B. Passwort).

Zum Serialisieren verwendet man die Klasse `ObjectOutputStream`, zum Deserialisieren die Klasse `ObjectInputStream`. Deren Konstruktoren wird ein `OutputStream` bzw. `InputStream` übergeben.

Danach können Objekte mittels

```
public void writeObject(Object o)
```

geschrieben und mit Hilfe von

```
public final Object readObject()
```

und Cast in den entsprechenden Typ gelesen werden.

Hiermit könnte man „von Hand“ Objekte von einem Rechner zu einem anderen übertragen (z.B. unter Benutzung von Socket-Verbindungen) und dann mit diesen auf dem anderen Rechner arbeiten. Man kann diese Handarbeit jedoch umgehen, wenn es nur darum geht, bestimmte Funktionalitäten eines Objektes woanders zu verwenden. Dies wird in Java durch Remote Method Invocation ermöglicht.

4.3.2 Remote Method Invocation (RMI)

In der OO-Programmierung haben wir eine Client-Server-Sicht von Objekten. Beim Aufruf einer Methode wird das aufrufende Objekt als Klient und das aufgerufene Objekt als Server gesehen.

Im verteilten Kontext werden Nachrichten dann echte Nachrichten im Internet (TCP). Prozesse, die miteinander kommunizieren, sind dann:

- Server, welche Informationen zur Verfügung stellen
- Klienten, die dies anfragen

Hiermit können beliebige Kommunikationsmuster (z.B. peer-to-peer) abgebildet werden. Die Idee von RMI geht auf *Remote Procedure Call (RPC)* zurück, welches für C entwickelt wurde.

Zunächst benötigt ein RMI-Client eine Referenz auf das Remote-Objekt. Dazu dient die RMI-Registrierung. Er fragt die Referenz mit Hilfe einer URL an:

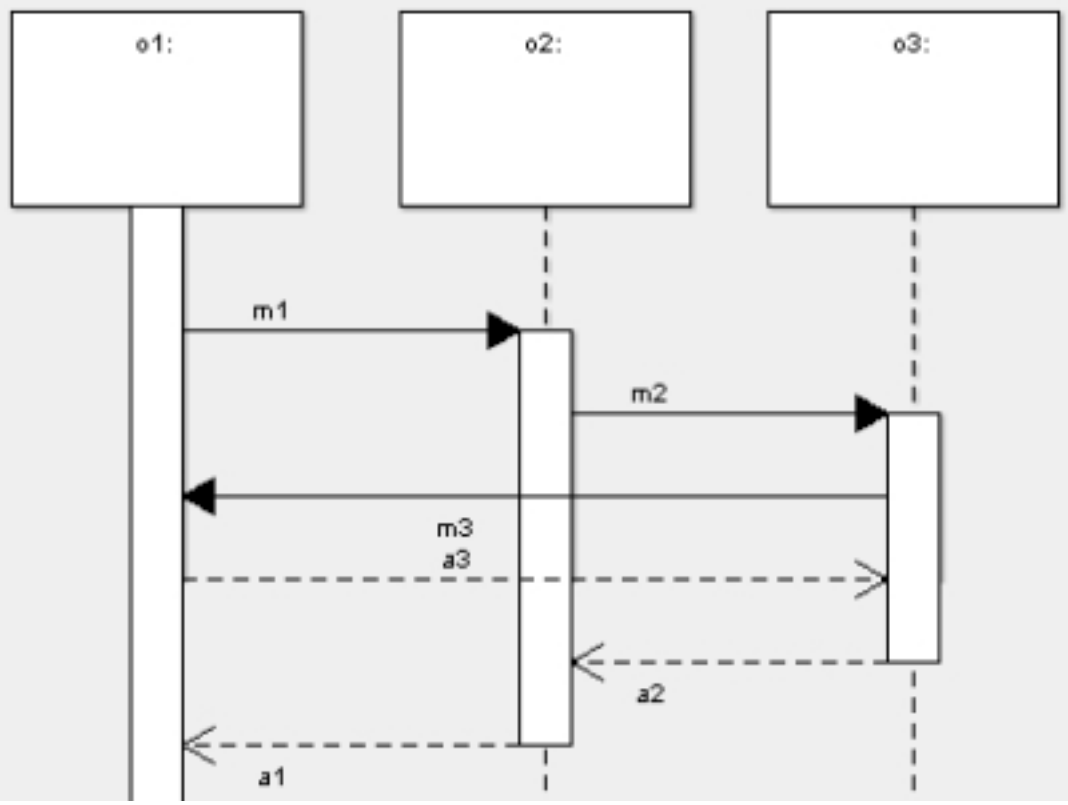


Abbildung 4.2: Remote Method Invocation in Java

```
rmi://hostname:port/servicename
```

Dabei kann `hostname` ein Rechnername oder eine IP-Adresse sein, `servicename` ist ein String, der ein Objekt beschreibt. Der Standardport von RMI ist 1099.

Es gibt auch noch eine zweite Möglichkeit, den Zugriff auf ein Remote-Objekt in einem Programm zu erhalten, und zwar als Argument eines Methodenaufrufs. In der Regel verwendet man die oben genannte Registrierung nur für den „Erstkontakt“, danach werden Objekte ausgetauscht und transparent (wie lokale Objekte) verwendet.

Solche Objekte können auf beliebigen Rechnern verteilt liegen. Methodenaufrufe bei entfernten Objekten werden durch Netzwerkkommunikation umgesetzt.

Das Interface für RMI ist aufgeteilt in Stub und Skeleton. Seit Java 5 sind diese jedoch nicht mehr sichtbar, sondern werden zur Laufzeit implizit generiert.

Die Netzwerkkommunikation erfolgt über TCP/IP, aber auch diese ist für den Anwendungsprogrammierer nicht sichtbar.

Die Parameter und der Rückgabewert einer entfernt aufgerufenen Methode müssen dafür natürlich in Bytes umgewandelt und übertragen werden. Hierfür gelten die folgenden Regeln:

- Bei Remote-Objekten wird nur eine Referenz des Objektes übertragen.
- Serializable-Objekte werden in `byte[]` umgewandelt. Auf der anderen Seite wird dann eine Kopie des Objektes angelegt.
- Primitive Werte werden kopiert.

Um Objekte von entfernten Knoten verwenden zu können, muss zunächst ein RMI Service Interface für dieses Objekt definiert werden. Dieses Interface beschreibt die Methoden, die dann von diesem Objekt von einem anderen Rechner aufgerufen werden können. Betrachten wir als Beispiel einen einfachen "Flip"-Server, d.h. ein Objekt, dass einen Booleschen Zustand enthält, der durch eine `flip`-Nachricht gewechselt werden kann. Das RMI Service Interface kann dann wie folgt definiert werden:

```
import java.rmi.*;

public interface FlipServer extends Remote {
    public void flip() throws RemoteException;
    public boolean getState() throws RemoteException;
}
```

Eine Implementierung des RMI-Interfaces sieht auf der Serverseite dann zum Beispiel so aus:

```
public class FlipServerImpl
    extends java.rmi.server.UnicastRemoteObject
    implements FlipServer {

    private boolean state;

    public FlipServerImpl() throws RemoteException {
        state = false;
    }

    public void flip() {
        state = !state;
    }

    public boolean getState() {
        return state;
    }
}
```

Früher musste man die Stub- und Skeleton-Klassen mit `rmic` generieren - das ist jetzt nicht mehr notwendig.

4.3.3 RMI-Registierung

Um ein Server-Objekt von einem anderen Rechner zu nutzen, muss dessen Name in einer RMI-Registry bekannt gemacht werden. Hierzu muss auf dem Rechner, auf dem das Server-Objekt laufen soll, ein RMI-Registry-Server initialisiert werden, der z.B. explizit mittels des Kommandos `rmiregistry` (z.B. in einer UNIX-Shell) gestartet werden kann. Seine Verwendung zur Objektregistrierung wird am folgenden Beispiel gezeigt:

```
public class Server {
  public static void main(String[] args) {
    try {
      FlipServerImpl server = new FlipServerImpl();

      String host;
      if (args.length >= 1) {
        host = args[0];
      } else {
        host = "localhost";
      }

      String url = "rmi://" + host + "/FlipServer";
      Naming.rebind(url, server);
    } catch (RemoteException e) {
      ...
    } catch (MalformedURLException e) {
      ...
    }
  }
}
```

Durch `rebind` wird der Name des Server-Objektes registriert. Auf der Client-Seite muss die RMI-Registry kontaktiert werden, um eine Referenz auf das Remote-Objekt zu erhalten, damit dann dessen `flip`-Methode aufgerufen werden kann:

```
public class Client {
  public static void main(String[] args) {
    try {
      String host;
      if (args.length >= 1) {
        host = args[0];
      } else {
        host = "localhost";
      }

      String url = "rmi://" + host + "/FlipServer";
```

```
      FlipServer s = (FlipServer)Naming.lookup(url);

      s.flip();
      System.out.println("State: " + s.getState());
    } catch (MalformedURLException e) {
      ...
    } catch (RemoteException e) {
      ...
    } catch (NotBoundException e) {
      ...
    }
  }
}
```

Damit stellt RMI stellt eine Fortsetzung der sequentiellen Programmierung auf verteilten Objekten dar. Somit können auch mehrere verteilte Prozesse auf ein Objekt „gleichzeitig“ zugreifen. Man muss also auch hier das Problem der nebenläufigen Synchronisation beachten!

Dynamisches Laden, Sicherheitskonzepte oder verteilte Speicherbereinigung (garbage collection) sind weitere Aspekte von Java RMI, die wir hier aber nicht betrachten.

Literaturverzeichnis

[1] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[3] J. Corbin and M. Bidoit. A rehabilitation of robinson's unification algorithm. In *Proc. IFIP '83*, pages 909–914. North-Holland, 1983.

[4] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[5] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[6] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[7] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.

[8] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

Abbildungsverzeichnis

Index