Skript zur Vorlesung

Fortgeschrittene Programmierung

WS 2020/21

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

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

Version vom 12. Februar 2021

Vorwort

In dieser Vorlesung werden fortgeschrittene 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 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. Dieses Skript enthält auch ein Kapitel zu „Java Generics“, welches nicht Bestandteil dieser Vorlesung ist, weil dessen Inhalte aus der Vorlesung zur objektorientierten Programmierung bekannt sein sollten. Weil dieses wichtige Sprachkonzept in dieser Vorlesung verwendet wird, wird es hier der Vollständigkeit halber noch einmal skizziert.

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, Januar 2021 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 Java Generics

1.1 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 Optional {

  private Object value;
  private boolean present;

  public Optional() {
    present = false;
  }

  public Optional(Object v) {
    value = v;
    present = true;
  }

  public boolean isPresent() {
    return present;
  }

  public Object get() {
    if (present) {
      return value;
    }
    throw new NoSuchElementException();
  }
}
```

Wir nutzen dabei aus, dass `NoSuchElementException` von `RuntimeException` abgeleitet ist, d.h. es ist eine sogenannte *unchecked exception* und muss nicht deklariert werden.
Die Verwendung der Klasse könnte dann wie folgt aussehen:

```
    Optional opt = new Optional(new Integer(42));
```

```
    if (opt.isPresent()) {
      Integer n = (Integer) opt.get();
      System.out.println(n);
    }
```

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 `Optional` 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 *parametrischer 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 Optional<T> {

  private T value;
  private boolean present;

  public Optional() {
    present = false;
  }

  public Optional(T v) {
    value = v;
    present = true;
  }

  public boolean isPresent() {
    return present;
  }

  public T get() {
    if (present) {
      return value;
    }
    throw new NoSuchElementException();
  }
}
```

Die Verwendung der Klasse sieht dann so aus:

```
    Optional<Integer> opt = new Optional<Integer>(new Integer(42));

    if (opt.isPresent()) {
      Integer n = opt.get();
      System.out.println(n);
```

```
    }
```

Es sind also keine expliziten Typumwandlungen mehr erforderlich, und ein Ausdruck wie

```
opt = new Optional<Integer>(opt);
```

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;
  }
}
```

1.2 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 `Optional` so erweitern, dass sie auch das Interface `Comparable` implementiert:

```
public class Optional<T extends Comparable<T>>
  implements Comparable<Optional<T>> {
  ...
  @Override
  public int compareTo(Optional<T> o) {
    if (present) {
      return o.isPresent() ? value.compareTo(o.get()) : 1;
    } else {
      return o.isPresent() ? -1 : 0;
    }
  }
}
```

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 (`T`, `S` und `U`) sieht das z. B. wie folgt aus:

```
<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. `A` und `B` müssen hierbei Klassen oder Interfaces sein, also insbesondere keine Typvariablen.

1.3 Wildcards

Die Klasse `Integer` ist Unterklasse der Klasse `Number`. Somit sollte doch auch der folgende Code möglich sein:

```
Optional<Integer> oi = new Optional<Integer>(new Integer(42));
Optional<Number> on = oi;
```

Das Typsystem erlaubt dies aber nicht, weil es festlegt, dass `Optional<Integer>` **kein** Untertyp von `Optional<Number>` ist! Das mögliche Problem wird deutlich, wenn wir zur Klasse `Optional` noch eine Setter-Methode hinzufügen:

```
public void set(T v) {
  present = true;
  value = v;
}
```

Dann wäre auch Folgendes möglich:

```
on.set(new Float(42));
Integer i = oi.get();
```

Da `oi` und `on` 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

```
Optional<?> ox = oi;
```

einen `Optional` von unbekanntem Typ.

Hierbei wird "`?`" auch als *Wildcard*-Typ bezeichnet. `Optional<?>` repräsentiert somit alle anderen `Optional`-Instantiierungen, z. B. `Optional<Integer>`, `Optional<String>` oder auch `Optional<Optional<Object>>`.

Damit können aber nur Methoden verwendet werden, die für jeden Typ passen, also z. B.

```
Object o = ox.get();
```

Wir können auch die Methode `set` für `ox` verwenden, allerdings benötigen wir hier einen Wert, der zu allen Typen gehört: `null`.

```
ox.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 sogenannte *beschränkte Wildcards* (*bounded wildcards*) verwenden:

`<? extends A>` steht für *alle* Untertypen des Typs `A` *(Kovarianz)*

`<? super A>` steht für *alle* Obertypen von `A` *(Kontravarianz)*

Dabei ist zu beachten, dass ein Typ sowohl Ober- als auch Untertyp von sich selbst ist, d. h. für alle Typen T gilt `<T extends T>` und `<T super T>`.

Es folgen einige Beispiele für die Benutzung von beschränkten Wildcards.

Ausdruck	erlaubt?	Begründung
`Optional<? extends Number> on = oi;`		
`Integer i = on.get();`	nein!	`?` könnte auch `Float` sein
`Number n = on.get();`	ja	
`on.set(n);`	nein!	`?` könnte spezieller als `Number` sein
`on.set(new Integer(42));`	nein!	`?` könnte auch `Float` sein
`on.set(null);`	ja	
`Optional<? super Integer> ox = oi;`		
`Integer i = ox.get();`	nein!	`?` könnte auch `Number` oder `Object` sein
`Number n = ox.get();`	nein!	`?` könnte auch `Object` sein
`Object o = ox.get();`	ja	
`ox.set(o);`	nein!	`?` könnte auch `Number` sein
`ox.set(n);`	nein!	`n` könnte auch vom Typ `Float` sein
`ox.set(i);`	ja	

Abbildung 1.1: Ausdrücke mit Wildcards

Wir können also aus einem `Optional<? extends A>` Objekte vom Typ `A` herausholen aber nur `null` hineinstecken; in einen `Optional<? super A>` können wir Objekte vom Typ `A` hineinstecken, aber nur Objekte vom Typ `Object` herausholen.

1.4 Typinferenz

Bei der Initialisierung einer Variablen mit einem generischen Typparameter fällt auf, dass man den Typparameter zwei Mal angeben muss:

```
Optional<Integer>       o = new Optional<Integer>(new Integer(42));
List<Optional<Integer>> l = new ArrayList<Optional<Integer>>(o);
```

Seit Version 7 ist der Java-Compiler jedoch in der Lage, bei einer Objektinitialisierung mittels `new` in den meisten Fällen den generischen Typ zu berechnen. Daher muss der Typ dann nur noch bei der Variablendeklaration angegeben werden, beim Aufruf von `new` wird der Typ durch den „Diamant-Operator“ `<>` berechnet:

```
Optional<Integer> mv = new Optional<>(new Integer(42));
List<Optional<Integer>> l = new ArrayList<>(mv);
```

Dabei kann nur die gesamte Typangabe in den spitzen Klammern weggelassen werden, Angaben wie `<Optional<>>` sind nicht zulässig. Sofern der Compiler den Typ jedoch nicht inferieren kann, muss dieser nach wie vor manuell angegeben werden.

Neben der Schreiberleichterung erlaubt der Diamant-Operator es nun auch, generische Singleton-Objekte zu erstellen, was vorher nicht möglich war:

```
Optional<?> empty = new Optional<>();
```

Dies ist insbesondere sinnvoll, um von unveränderlichen Objekten nur eine Instanz im Speicher vorzuhalten.

2 Nebenläufige Programmierung in Java

2.1 Allgemeine Vorbemerkungen

2.1.1 Motivation

Eine Anwendung in der Informatik wird als *nebenläufig* bezeichnet, wenn es nicht einen streng sequenziellen Ablauf hat, sondern wenn die Anwendung aus mehreren Aktivitäten zusammengesetzt ist, die nebenläufig, d.h. (fast) gleichzeitig, ablaufen. Diese Aktivitäten werden häufig als Tasks, Threads oder auch Prozesse bezeichnet. Diese Tasks sind selbst eigene sequenziell ablaufende Programme. In einem nebenläufigen Programm gibt es also nicht einen Programmzähler, der die nächste abzuarbeitende Anweisung festlegt, sondern viele Programmzähler. Wie wir noch sehen werden, ist die Entwicklung nebenläufiger Software mit vielen Fallstricken verbunden. In diesem Kapitel wollen wir Methoden aufzeigen, wie man nebenläufige Software so entwickelt, damit diese möglichst zuverlässig abläuft. Darauf aufbauend werden wir später auch verteilte Software betrachten. Als Programmiersprache werden wir Java verwenden, aber die Konzepte kann man auch auf andere Sprachen übertragen.

Wozu benötigen wir nebenläufige Programmierung? 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, ...)

2.1.2 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.

2.1.3 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.

2.1.4 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:

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 sogenanten *green threads*.

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.

2.1.5 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 oder 2 erzeugen, je nachdem, wie der Scheduler die beiden nebenläufigen Prozesse ausführt. Hängt das Ergebnis eines Programmlaufs von der Reihenfolge des Schedulings ab, nennt man dies eine *Race-Condition*.

Neben den beiden Ergebnissen 1 und 2 ist aber auch noch ein weiteres Ergebnis, nämlich 0, möglich. Dies liegt daran, dass noch nicht klar spezifiziert wurde, welche Aktionen wirklich atomar ausgeführt werden. 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*.

2.1.6 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* haben einen internen Zustand (einen Wert `value` und eine Warteschlange `queue` für Prozesse) und stellen zwei *atomare* Operationen `P` und `V` zur Verfügung, die im Pseudocode ungefähr wie folgt implementiert sind:

```
P(s) {
  if   s.value >= 1
    then s.value = s.value - 1;
    else <suspend current thread and add it to s.queue>
}

V(s) {
  if <s.queue not empty>
    then <delete some thread from s.queue and wake it up>
    else s.value = s.value + 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 des Semaphors 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:

```
Buffer buffer = ...
Semaphore num = 0;
```

Code für den Producer:

```
while (true) {
  newproduct = produce();
  push(newproduct, 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:

```
Buffer buffer = ...
Semaphore num = 0;
Semaphore bufferAccess = 1;
```

Code für den Producer:

```
while (true) {
  newproduct = produce();
  P(bufferAccess);
  push(newproduct, buffer);
  V(bufferAccess);
  V(num);
}
```

Code für den Consumer:

```
while (true) {
  P(num);
  P(bufferAccess);
  prod = pull(buffer);
  V(bufferAccess);
  consume(prod);
}
```

Die Verwendung von Semaphoren bringt 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 einem binären Semaphor `s` bereits einen *Deadlock* erzeugen.

Eine Verbesserung bieten hier die *Monitore*, die wir bereits aus der Vorlesung „Betriebs- und Kommunikationssysteme“ kennen. In der Tat verwendet Java einen Mechanismus

ähnlich dieser Monitore zur Synchronisierung.

2.1.7 Dining Philosophers

Das Problem der dinierenden Philosophen (*dining philosophers*) mit n Philosophen lässt sich wie folgt mit Hilfe von Semaphoren modellieren:

```
Semaphore stick1 = 1;
Semaphore stick2 = 1;
Semaphore stick3 = 1;
Semaphore stick4 = 1;
Semaphore stick5 = 1;

par { phil(stick1, stick2); }
    { phil(stick2, stick3); }
    { phil(stick3, stick4); }
    { phil(stick4, stick5); }
    { phil(stick5, stick1); }
end par;
```

Code für Philosoph `i`:

```
phil(stickl,stickr) {
  while (true) {
    think();

    P(stickl);
    P(stickr);

    eat();

    V(stickl);
    V(stickr);
  }
}
```

Dabei kann jedoch ein Deadlock auftreten. falls alle Philosophen gleichzeitig ihr linkes Stäbchen nehmen. Diesen Deadlock können wir durch Zurücklegen vermeiden:

```
while (true) {
  think();

  P(stickl);

  if (lookup(stickr) == 0) {   # Zahlwert der Semaphore nachschauen
    V(stickl);
  } else {
    P(stickr);
```

```
      eat();

      V(stickl);
      V(stickr);
    }
  }
```

Hier bezeichnet `lookup(s)` eine Lookup-Funktion des abstrakten Datentyps Semaphore, die uns den Integer-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.

2.2 Threads in Java

2.2.1 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.

2.2.2 Das Interface `Runnable`

Java bietet keine Mehrfachvererbung. Deshalb ist eine Erweiterung der Klasse `Thread` häufig ungünstig. Eine Alternative bietet das Interface `Runnable`, welches nur die Methode `run()` besitzt, d.h. eine Klasse, die `Runnable` implementiert, muss die Methode `run()` implementieren:

```
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()`.

2.2.3 Eigenschaften von Thread-Objekten

Jedes Thread-Objekt in Java hat eine Reihe von Eigenschaften:

Name Jeder Thread besitzt einen Namen, wie z.B. `"main-Thread"`, `"Thread-0"` oder `"Thread-1"`. Der Zugriff auf den Namen eines Threads erfolgt über die Methoden `getName()` und `setName(String)`. Man kann diese Namen z.B. zum Debuggen verwenden.

Zustand Jeder Thread befindet sich stets in einem bestimmten Zustand. Eine Übersicht dieser Zustände und der Zustandsübergänge ist in Abbildung 2.1 dargestellt. Ein

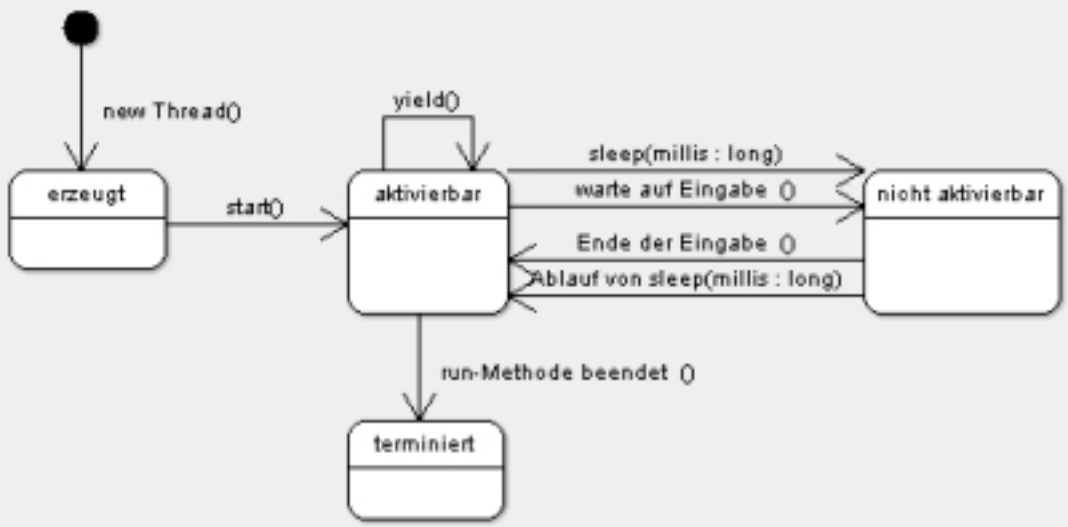


Abbildung 2.1: Zustände von Threads

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, 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.

Threadgruppe 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.

2.2.4 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 einer Klasse 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 Me-

thode, 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.

2.2.5 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 int balance;

  public Account(int initialDeposit) {
    balance = initialDeposit;
  }

  public synchronized int getBalance() {
    return balance;
  }

  public synchronized void deposit(int amount) {
    balance = 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(int)` 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 2.1.4). Interprozesskommunikation und Synchronisation). Mit Anwendung des Schlüsselwortes ist eine Ausgabe von `500` garantiert.

2.2.6 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 der Ausdruck `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. sequenzialisieren zu müssen:

```
class State {
  private double state;

  public void calc() {
    double res;
    // do some really expensive computation
    ...
    // save the result to an instance variable
    synchronized (this) {
      state = res;
    }
  }
}
```

Die Synchronisation einzelner Anweisungen ist auch nützlich, um auf andere Objekte zu synchronisieren. Wir betrachten als Beispiel eine einfache Implementierung einer synchronisierten 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);
  }
}
```

2.2.7 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 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:

```
// create an unsynchronized list:
List<Integer> unsyncList = new List<Integer>();
... // fill this list
List<Integer> list = Collections.synchronizedList(unsyncList);
```

Nun ist `list` eine synchronisierte Liste, d.h. alle Zugriffe über `list` auf die Liste werden synchronisiert. Wir können nun 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 können nicht garantieren, dass nicht ein anderer Thread die Collection zwischenzeitig verändert. Dies ist ein klassisches Beispiel für client-side synchronisation.

2.2.8 Kommunikation zwischen Threads

Threads kommunizieren über geteilte Objekte miteinander. Betrachten wir als Beispiel ein Objekt, bei dem ein Zustand z.B. durch einen Thread geändert wird und ein anderer Thread diese Änderung anzeigen soll. Wie kann ein Thread aber herausfinden, wann ein Zustand geändert wird? 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. Busy waiting bedeutet, dass ein Thread auf das Eintreffen eines Ereignisses wartet, dabei aber weiterrechnet und somit Ressourcen wie Prozessorzeit verbraucht. Um so etwas zu vermeiden, kann man Thread auch explizit suspendieren und später wieder aufwecken. Hierzu suspendiert man einen Thread mittels der Methode `wait()` des Objektes, und weckt ihn später mittels `notify()` oder `notifyAll()` wieder auf.

Das sieht zum Beispiel so aus:

```
public class PrintState {
  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 hat das `notify()` keine Wirkung, d.h. der beabsichtigte Effekt von `notify()` geht 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 drei 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()
  throws InterruptedException {
  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()
  throws InterruptedException {
  while (true) {
    if (!modified) {
      wait();
    }

    System.out.println(state);
    modified = false;
    notify();
  }
}

public synchronized void setValue(int v)
  throws InterruptedException {
  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()
  throws InterruptedException {
  while (true) {
    while (!modified) {
      wait();
    }

    System.out.println(state);
    modified = false;
```

```
    notify();
  }
}

public synchronized void setValue(int v)
  throws InterruptedException {
  while (modified) {
    wait();
  }

  state = v;
  notifyAll();
  modified = true;
  System.out.println("value set");
}
```

Es ist also wichtig, Bedingungen, die ein `wait` auslösen, nach dem Aufwecken wieder zu überprüfen, weshalb man diese Überprüfung immer in einer Schleife machen sollte! Dies sollte man auch machen, wenn es nur zwei Threads gibt, denn manche Thread-Implementierungen erzeugen sogenannte "spurious wakeups", d.h. ein wartender Prozess wird eventuall auch geweckt, ohne dass ein anderer Prozess ein `notify` gesendet hat!

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.

2.2.9 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 denn auch gezielt erwecken? Ja! Dazu verwenden wir spezielle Objekte zur Synchronisation der `take`- und `put`-Threads.

```
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` sehr wichtig! Ein anderer Thread, der die Methode von außen betritt, könnte sonst einen wartenden (und gerade aufgeweckten) Thread noch überholen!

2.2.10 Beenden und Unterbrechen von Threads

Java bietet mehrere Möglichkeiten, Threads zu beenden:

1. Beenden der `run()`-Methode
2. Abbruch der `run()`-Methode
3. Aufruf der `destroy()`-Methode (diese ist veraltet und daher 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.

Falls man Threads gestartet hat und warten möchte, bis diese beendet werden, so kann man hierzu die `Thread`-Methode

```
void join() throws InterruptedException
```

verwenden (es gibt auch Varianten von `join`, bei denen man noch Zeitlimits angeben kann). Wenn man also für einen Thread `th` die Methode `th.join()` aufruft, dann wartet die Programmausführung so lange, bis `th` beendet ist. Falls man während des Wartens einen Interrupt erhält (s.u.), dann wird eine `InterruptedException` ausgelöst.

Java sieht zusätzlich eine Möglichkeit zum Unterbrechen von Threads über *Interrupts* vor. Jeder Thread hat ein Flag, welches Interrupts anzeigt.

Die `Thread`-Methode `interrupt()` sendet einen Interrupt an einen Thread, das Flag wird gesetzt. Falls der Thread aufgrund eines Aufrufs von `sleep()`, `wait()` oder `join()` schläft, wird er erweckt und eine `InterruptedException` geworfen.

```
synchronized (o) {
  ...
  try {
    ...
    o.wait();
    ...
  } catch (InterruptedException e) {
    ...
  }
}
```

Bei einem 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 einen 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 setzen (mittels der Methode `public void interrupt()`)!

2.3 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. Damit die Daten über ein Netzwerk verschickt werden können, müssen sie (Argumente und Ergebnisse von Methodenaufrufen) in Byte-Folgen umgewandelt werden, was in der Regel als Serialisierung bezeichnet wird.

2.3.1 Serialisierung/Deserialisierung von Daten

Die Serialisierung eines Objektes `o` liefert eine Byte-Sequenz, die Deserialisierung der Byte-Sequenz liefert ein neues Objekt `o1`. Beide Objekte sollen bezüglich ihres Verhaltens gleich sein, haben aber unterschiedliche Objektidentitäten, d.h. `o1` ist eine Kopie von `o`.

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.

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

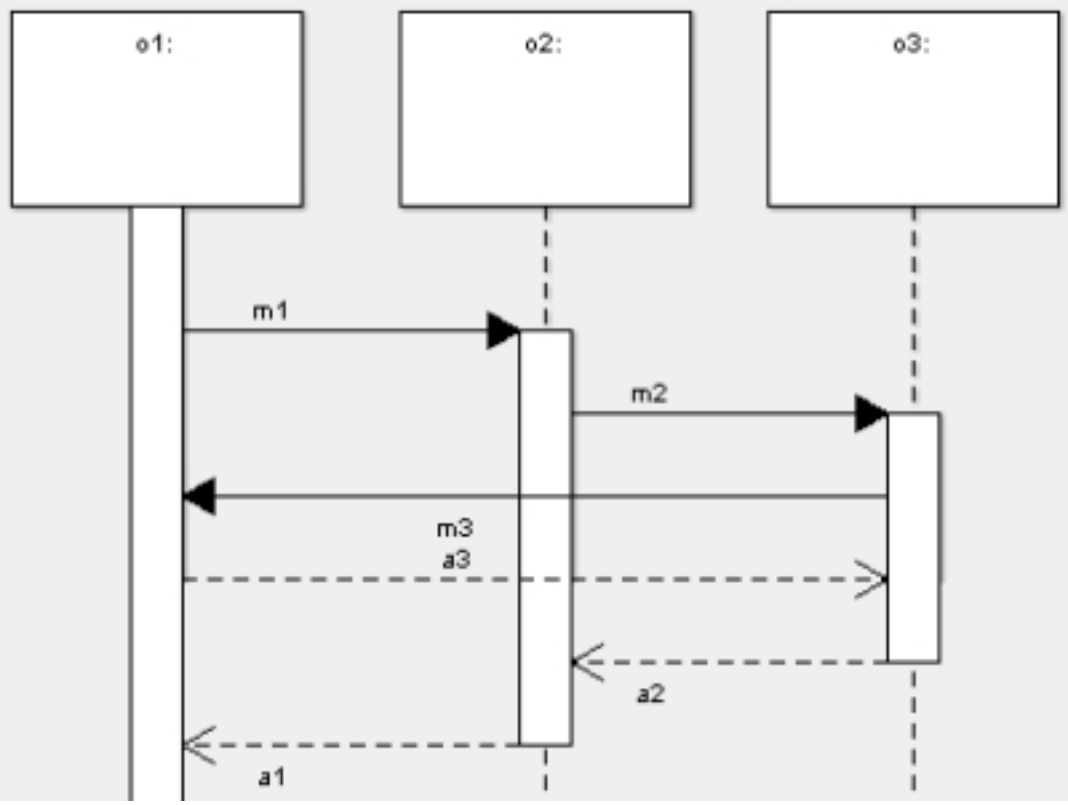


Abbildung 2.2: Remote Method Invocation in Java

```
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.

2.3.2 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 (Übertragung per 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:

```
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 (Remote-)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, das 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:

```
import java.rmi.*;
import java.rmi.server.*;

public class FlipServerImpl extends 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 dem RMI-Compiler `rmic` generieren - das ist jetzt nicht mehr notwendig. Die Klasse `UnicastRemoteObject` stellt die einfachste Art der Realisierung von Remoteobjekten dar. Alle notwendigen Übersetzungsschritte werden automatisch vorgenommen. In einzelnen Fällen, kann es auch notwendig sein, selber in den (De-)Serialisierungsprozess einzugreifen und Eigenschaften der `RemoteObjects` selber zu definieren, wozu noch weitere Schnittstellen angeboten werden.

2.3.3 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:

```
import java.rmi.*;
import java.net.*;

public class Server {
  public static void main(String[] args) {
    try {
      String host = args.length >= 1 ? args[0] : "localhost";
      String uri  = "rmi://" + host + "/FlipServer";
```

```
      FlipServerImpl server = new FlipServerImpl();
      Naming.rebind(uri, server);
    } catch (MalformedURLException|RemoteException 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:

```
import java.rmi.*;
import java.net.*;

public class Client {
  public static void main(String[] args) {
    try {
      String host = args.length >= 1 ? args[0] : "localhost";
      String uri = "rmi://" + host + "/FlipServer";

      FlipServer s = (FlipServer) Naming.lookup(uri);
      s.flip();
      System.out.println("State: " + s.getState());
    } catch (MalformedURLException|NotBoundException|RemoteException e) {
      ...
    }
  }
}
```

Damit stellt RMI stellt eine Fortsetzung der sequenziellen 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! Allerdings kann hier die Synchronisation schwieriger sein, wie das folgende Beispiel zeigt.

Wie wir oben gesehen haben, ist *client-side synchronisation* sinnvoll, um die atomare Ausführung mehrerer Methoden zu garantieren, die durchaus schon synchronisiert sein können. Zu diesem Zweck betrachten wir noch einmal den einfachen Flip-Server und einen Client, der die `flip`-Methode zweimal ohne Unterbrechung aufruft:

```
...
FlipServer s = (FlipServer) Naming.lookup(uri);
synchronized(s) {
    System.out.println("State1: " + s.getState());
    s.flip();
    Thread.sleep(2000);
    s.flip();
    System.out.println("State2: " + s.getState());
}
...
```

Da der Client die beiden Aufrufe auf dem Flip-Server `s` synchronisiert, sollte kein anderer währenddessen den Zustand des Flip-Servers verändern können, so dass die Ergebnisse beider Ausgaben immer gleich sind. Dies wäre im rein nebenläufigen Kontext auch der Fall, aber im verteilten Kontext gilt dies nicht mehr, da die Synchronisation nicht auf den Remote-Objekten, sondern den lokalen Stub-Objekten stattfindet!

Dynamisches Laden, Sicherheitskonzepte oder verteilte Speicherbereinigung (garbage collection) sind weitere Aspekte von Java RMI, die wir hier aber nicht betrachten.

3 Funktionale Programmierung

Ein Schwerpunkt dieser Vorlesung bildet dieses Kapitel zur funktionalen Programmierung. Die praktische Relevanz der funktionalen Programmierung wird z.B. von Thomas Ball und Benjamin Zorn von Microsoft in dem Artikel [3], der den vielsagenden Untertitel "Industry is ready and waiting for more graduates educated in the principles of programming languages" trägt, betont:

> Second, would-be programmers (CS majors or non-majors) should be exposed as early as possible to functional programming languages to gain experience in the declarative programming paradigm. The value of functional/declarative language abstractions is clear: they allow programmers to do more with less and enable compilation to more efficient code across a wide range of runtime targets.

Interessant ist auch die erste Empfehlung der Autoren:

> First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, its application in design formalisms, and experience the creation and debugging of formal specifications with automated tools...

Daher ist auch die Beschäftigung mit mathematischen Grundlagen und Logik in einem Informatikstudium wichtig, aber dies ist nicht Bestandteil dieser Vorlesung.

Die funktionale Programmierung bietet im Vergleich zur klassischen imperative Programmierung eine Reihe von Vorteilen:

- hohes Abstraktionsniveau, keine Manipulation von Speicherzellen
- keine Seiteneffekte, deshalb leichtere Codeoptimierung und bessere Verständlichkeit
- Programmierung über Eigenschaften, nicht über den 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

Für die praktische Programmierung sind Kenntnisse der funktionalen Programmierung wichtig, da funktionale Programmiertechniken und Sprachkonstrukte zu besser strukturierten Programmen führen, wie in dem Artikel [16] erläutert wird. Daher sind funktionale Konzepte auch in zum Teil eingeschränkter Form in vielen modernen Programmiersprachen zu finden. Wir werden aber zunächst einmal die rein funktionale Programmierung

vorstellen.

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 [5] zurück. Im Folgenden führen wir die rein funktionale Programmierung anhand der Programmiersprache Haskell [14] ein.

3.1 Ausdrücke und Funktionen

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

$$x = x + 1 \qquad \text{oder} \qquad x := x + 1$$

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

Damit sieht Haskell fast wie eine Skriptsprache aus. Im Gegensatz zu Skriptsprachen wie PHP, Ruby oder auch Python, hat Haskell alle Elemente, um auch große Softwaresysteme zu realisieren. Insbesondere ist Haskell im Gegensatz zu Skriptsprachen *streng getypt*, d.h. alle Werte und Ausdrücke haben einen Typ, der durch das Haskell-System geprüft wird,

bevor das Programm ausgeführt wird. Wir werden auf die verschiedenen Datentypen in Kapitel 3.2 genauer eingehen. Hier wollen wir aber schon bei allen Funktionen den Typ annotieren, um deutlich zu machen, wofür diese verwendet werden.[1] Ein einfacher Basisdatentyp ist `Int`, die Menge der ganzen Zahlen (bzw. eine endliche Teilmenge davon). Der Typ einer Funktion wird durch "`::`" annotiert, wobei mehrere Argumenttypen und der Ergebnistyp durch "$\rightarrow$" getrennt werden. Weiterhin sollte die intuitive Bedeutung von Funktionen durch einen *Kommentar* vor der Funktionsdefinition erläutert werden, wobei Kommentare durch zwei Minuszeichen eingeleitet werden und bis zum Zeilenende gehen.

Als Beispiel betrachten wir eine Funktion zur Quadratberechnung. Diese können wir in Haskell wie folgt definieren:

```
-- Explicit import of SimplePrelude
import SimplePrelude

-- Computes the square of a number.
square :: Int  → Int
square x = x * x
```

Anmerkung: Die `import`-Zeile ist normalerweise nicht notwendig. Wir benutzen diese hier, weil wir mit einer vereinfachten Version von Haskell arbeiten, die einige komplexere Konstrukte weglässt und damit den Einstieg erleichtern soll. Aus diesem Grund benutzen wir zunächst auch nicht den Haskell-Interpreter `ghci`, der zum Glasgow Haskell Compiler (GHC) gehört, sondern das Skript `fortprog-ghci` (siehe Web-Seiten zur Vorlesung).

Wenn diese Definition in der Datei `Square.hs` gespeichert ist, dann kann man mit dem Skript `fortprog-ghci` das interaktive Haskell-System starten, um die Definitionen zu laden und die Funktion `square` aufzurufen:

```
> fortprog-ghci
FortProg-GHCi, version 1.0
SimplePrelude> :load Square
[2 of 2] Compiling Main             ( Square.hs, interpreted )
Ok, two modules loaded.
*Main> square 3
9
*Main> square (3 + 1)
16
*Main> :q
```

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

[1] Haskell kann, im Gegensatz zu vielen anderen Sprachen, Funktionstypen auch inferieren, sodass man diese nicht unbedingt hinschreiben muss. Es ist aber ein besserer Programmierstil, Funktionstypen auch zur Programmdokumentation aufzuschreiben.

```haskell
-- Computes the mininum of two numbers.
min :: Int  → Int  → Int
min x y = if x <= y then x else y
```

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

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

In Haskell setzen wir diese Funktion so um:

```haskell
-- Computes the factorial of a non-negative number.
fac :: Int  → Int
fac n = if n == 0 then 1 else n * fac (n - 1)
```

3.1.1 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 3.1 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)
       =  if False then 1 else 2 * fac (2 - 1)
       =  2 * fac (2 - 1)
       =  2 * fac 1
       =  2 * (if 1 == 0 then 1 else 1 * fac (1 - 1))
       =  2 * (if False then 1 else 1 * fac (1 - 1))
       =  2 * 1 * fac (1 - 1)
       =  2 * 1 * fac 0
       =  2 * 1 * (if 0 == 0 then 1 else 0 * fac (0 - 1))
       =  2 * 1 * (if True then 1 else 0 * fac (0 - 1))
       =  2 * 1 * 1
       =  2 * 1
       =  2
```

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

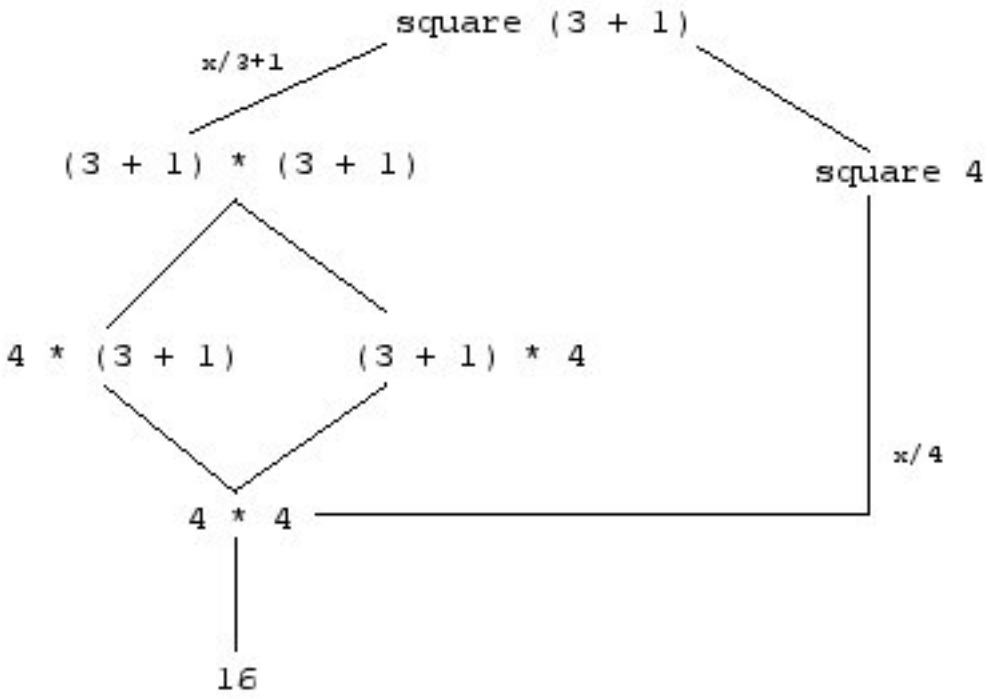


Abbildung 3.1: Mögliche Auswertungen von Funktionen

```
-- Computes the n-th Fibonacci number.
fib1 :: Int  → Int
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 ... $fib(n)$.

Die Realisierung dieser Idee ist etwas schwieriger, da wir hierzu die beiden vorherigen Zahlen stets als Parameter mitführen müssen:

```
-- An accumulator function to compute n-th Fibonacci number more efficiently.
fib2' :: Int  → Int  → Int  → Int
fib2' fibn fibnp1 n = if n == 0
                        then fibn
                        else fib2' fibnp1 (fibn + fibnp1) (n - 1)
```

```
-- Computes the n-th Fibonacci number.
fib2 :: Int  → Int
fib2 n = fib2' 0 1 n
```

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

Diese Programmiertechnik ist bekannt unter dem Namen *Akkumulatortechnik*, da wir hierbei Zwischenergebnisse (die bisher berechneten Fibonacci-Zahlen) akkumulieren und in Variablen (Parametern) mitführen. Um genauer zu verstehen, wie sich diese mitgeführten Werte bei der Berechnung entwickeln, ist es sinnvoll, die Berechnungsschritte zu skizzieren, wobei wir hier die Auswertung des `if-then-else` überspringen:

```
fib2 6  =  fib2' 0 1 6
        =  fib2' 1 1 5
        =  fib2' 1 2 4
        =  fib2' 2 3 3
        =  fib2' 3 5 2
        =  fib2' 5 8 1
        =  fib2' 8 13 0
        =  8
```

Wie wir sehen können, ist die korrekte Definition dieser Methode, aufwändiger als die direkte rekursive Definition.

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.

3.1.2 Lokale Definitionen

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

```
fib2 :: Int  → Int
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:

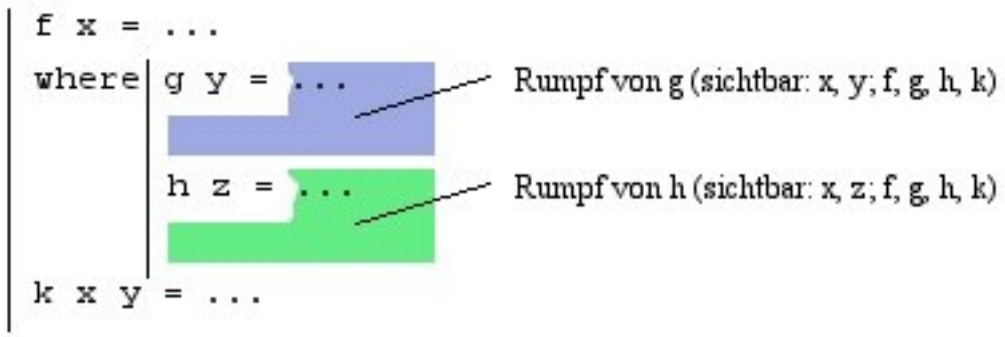


Abbildung 3.2: Layout-Regel in Haskell

```
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
- 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:

```
-- Checks whether a number is prime.
isPrim :: Int  → Bool
isPrim n = n /= 1 && checkDiv (div n 2)
  where checkDiv m =
          m == 1 || mod n m /= 0 && checkDiv (m - 1)
```

Hier drückt `/=` Ungleichheit auf ganzen Zahlen 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.

3.2 Datentypen

3.2.1 Basisdatentypen

Ganze Zahlen

Einen Basisdatentyp von Haskell haben wir oben bereits verwendet: ganze Zahlen. Diese werden in Haskell durch den folgenden Typ repräsentiert:

`Int`: Werte -2^{31} ... $2^{31} - 1$

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

Boolesche Werte

Ein weiterer Basisdatentyp sind die booleschen Werte:

`Bool`: `True False`

Operationen: `&& || not`

Gleitkommazahlen

Gleitkommazahlen sind ebenfalls ein Basisdatentyp:

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

Operationen: `+. -. *. /.`
Vergleiche: `<. >. <=. >=. ==. /=.`

Zur Unterscheidung der Operationen auf ganzen Zahlen wird ein Punkt hinter den Gleitkommaoperationen geschrieben. Dies ist wegen der Benutzung des Skriptes `fortprog-ghci` notwendig. Wenn wir später den vollen Sprachumgang von Haskell verwenden, muss man den Punkt weglassen.

Zeichen

Und auch Unicode-Zeichen sind ein Haskell-Basisdatentyp:

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

Operationen:[2]

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

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

3.2.2 Typannotationen

Wie schon erwähnt wurde, ist Haskell eine streng getypte Programmiersprache, d.h. alle Werte und Ausdrücke in Haskell haben einen Typ, welcher auch annotiert werden kann:

```
3 :: Int
```

[2] Diese Operationen stehen in der `SimplePrelude` direkt zur Verfügung. In normalen Haskell-Programmen muss man diese aus der Bibliothek `Data.Char` hinzuladen, indem man die Importdeklaration "`import Data.Char`" zu Beginn des Haskell-Programms angibt oder in der interaktiven GHCi-Umgebung den Befehl "`:module Data.Char`" ausführt.

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

```
(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 3.1), 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.

3.2.3 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 τ_11 ... τ_1n_1 | ... | C_k τ_k1 ... τ_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 :: τ_i1 → ··· → τ_in_i → τ
  ```

  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):

```
-- Add two complex numbers:
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 IntList = Nil | Cons Int IntList
```

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

```
-- Concatenate two lists of integer elements:
append :: IntList  → IntList  → IntList
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 `IntList`.

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 `SimplePrelude` 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 können diese wie „normale" Funktionen verwendet werden d.h. die Klammern deuten an, dass die Argumente nicht links und rechts stehen und die Argumente somit wie üblich nach dem Funktionsnamen folgen:

`(++) :: [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 ausgeben zu können. Hierzu kann man das Schlüsselwort `deriving` hinter der Datentypdefinition verwenden:

```
data MyType = ...
  deriving Show
```

3.3 Polymorphismus

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

```
-- Compute the length of a list of integers:
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:

```
-- Compute the last element of a list:
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 er-

zeugen 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)

-- Compute the height of a binary tree:
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 `12` 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 geschrieben 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!

3.4 Pattern Matching

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

```
f pat11 ... pat1n = e1
⋮
f patk1 ... patkn = ek
```

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.

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

3.4.2 Case-Ausdrücke

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

```
case e of pat_1  → e_1
          ⋮
          pat_n  → 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:

```
-- Breaks a string into a list of lines where a line is terminated at a
-- newline character. The resulting lines do not contain newline characters.
lines :: String  →  [String]
lines ""        = []
lines ('\n':cs) = "" : lines cs
lines (c:cs)    = case lines cs of
                    []     →  [[c]]
                    (l:ls) →  (c : l) : ls
```

3.4.3 Guards

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

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

`otherwise` ist hierbei kein Schlüsselwort, sondern eine Funktion, die immer zu `True` auswertet.

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

```
-- Returns prefix of length n.
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.

3.5 Funktionen höherer Ordnung

In Haskell werden Funktionen nicht nur zur Definition von Berechnungsverfahren eingesetzt, sondern Funktionen sind auch „Bürger erster Klasse“, denn sie können wie alle anderen Werte (z.B. Zahlen) verwendet werden. Dies bedeutet, dass Funktionen in Datenstrukturen und auch als Parameter oder Ergebnisse anderer Funktionen auftauchen können. In letzterem Fall spricht man auch von „Funktionen höherer Ordnung“.

Funktionen höherer Ordnung kann man z.B. zur

- generischen Programmierung
- Definition von Programmschemata (Kontrollstrukturen)

einsetzen. Hierdurch erreichen wir eine bessere Wiederverwendbarkeit und eine höhere Modularität des Programmcodes.

3.5.1 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:

```
-- Computes the derivation of a (continuous) function.
derive :: (Float → Float) → (Float → Float)
derive f = f'
 where f' :: Float → Float
       f' x = (f (x +. dx) -. f x) /. dx

dx :: Float
dx = 0.0001
```

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

3.5.2 Anonyme Funktionen (Lambda-Abstraktionen)

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

```
derive (\x → x *. x)
```

Das Argument entspricht der Funktion $x \mapsto x^2$. 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:

```
\ p1 ... pn → 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
```

Verwendet man diese Funktion, so können wir sehen, dass sie sich annähernd so wie die abgeleitete Funktion ($y \mapsto 2y$) verhält:

```
(derive (\x → x *. x)) 0.0 → 0.0001
(derive (\x → x *. x)) 2.0 → 4.0001
(derive (\x → x *. x)) 4.0 → 8.0001
```

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

```
add :: Float → Float → Float
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 `Float` nimmt und eine Funktion liefert, die einen weiteren `Float` nimmt und erst dann einen `Float` liefert.

Somit müssen die Typen `Float → Float → Float` und `Float → (Float → Float)` identisch sein. Tatsächlich ist der Typkonstruktor "→" als rechtsassoziativ definiert, so dass diese Bindung immer gilt, wenn keine Klammern geschrieben werden. Man sollte beachten, 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*. Die partielle Applikation wird syntaktisch durch Currying einfach möglich. Der Name *Currying* geht auf *Haskell B. Curry* zurück, der in den 1940er Jahren die folgende Isomorphie festgestellt hat:

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

Dies bedeutet, dass eine Funktion mit zwei Argumenten auch als Funktion mit einem Argument aufgefasst werden kann, die dann eine weitere Funktion für das zweite Argument liefert.

Eigentlich wurde dies schon viel früher im Jahr 1924 von *Moses Schönfinkel* festgestellt [17]. Weil aber dieser Artikel in Deutsch erschienen ist und "*Schönfinkeling*" sich englischsprachig nicht so gut aussprechen lässt, hat sich die Bezeichnung "Currying" durchgesetzt.

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`.
- `(/. 2.0)` steht für `\x -> x /. 2.0`.
- `(-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.

```
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 take) :: [a] → Int → [a]
(flip take) "Hello World!" :: Int → [Char]
```

3.5.3 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 | ord c == ord 'Z' = 'A'
       | ord c == ord 'z' = 'z'
       | 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, das wir kennen, 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 :: (b  → a  → b)  → b  → [a]  → b
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
```

3.5.4 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.

3.5.5 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 j
```

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.

3.5.6 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)
```

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
```

3.5.7 Funktionen höherer Ordnung in imperativen Sprachen

Auch moderne imperative Programmiersprachen bieten inzwischen die Möglichkeit, Funktionen als Parameter oder Rückgabewerte zu verwenden. Zwar bieten diese Programmiersprachen hierbei nicht die Möglichkeit der partiellen Applikation – wie in Haskell – und auch syntaktisch sind sie nicht ganz so elegant eingebunden, dennoch ermöglichen es Funktionen höherer Ordnung auch in imperativen Sprachen viele Algorithmen kompakter auszudrücken. Auch können sie genutzt werden, um zusätzliche Abstraktionsmöglichkeiten zu realisieren, da Methoden nun nicht mehr nur von Werten abstrahieren, sondern mit Hilfe von Funktionen auch von konkretem Verhalten abstrahiert werden kann.

Im Folgenden betrachten wir die Verwendung von funktionalen Parametern in Ruby und Java 8. Neben den konkreten syntaktischen Umsetzungen gehen wir insbesondere auf Probleme ein, welche sich prinzipiell aus dem Zusammenspiel zwischen Funktionen als

Werten und veränderbaren Variablen, Objekten bzw. Speicherzellen ergeben.

Funktionen höherer Ordnung in Ruby

Funktionen höherer Ordnung sind ein festes Sprachkonstrukt in der Programmiersprache Ruby.[3] Sie stehen in Form von „Blöcken“ zur Verfügung. Ein Block ist eine Anweisungssequenz, die zusätzlich auch noch parameterisiert werden kann. Einfachste Blöcke sind hierbei nicht parameterisierte Blöcke, wie sie z.B. als Schleifenrümpfe vorkommen. Solche Blöcke können aber auch Parameter (zwischen senkrechten Strichen notiert) haben, wodurch sie anonymen Funktionen entsprechen. Methoden und Funktionen können zusätzlich zu normalen Parametern auch noch einen Block erwarten und verwenden. Als einfachste Beispiele betrachten wir die Methoden `each` und `map`, welche z.B. für Arrays definiert sind:

```
a = [1,2,3,4,5]

b = a.map do |x| x + 1 end

b.each do |x| puts x end
```

Das Program gibt die Zahlen von 2 bis 6 auf dem Bildschirm mittels `puts` aus.[4].

Die `map`-Methode wendet den übergebenen einstelligen Block auf jedes Element des Arrays an. Hierbei wird das Array nicht mutiert, stattdessen wird ein neues Array gleicher Länge angelegt. Die `each`-Methode wendet ebenfalls den übergebenen einstelligen Block auf jedes Element des Arrays an, konstruiert aber kein neues Array.

Um zu verstehen, wie man Blöcke verwendet, wollen wir eine eigene Version von `map` definieren, welche aber im Gegensatz zu der oben verwendeten Methode das übergebene Array verändert (mutiert). Um nicht zu tief in die spezifischen Ruby-Details einzusteigen, definieren wir diese Funktion hier nicht als Methode, sondern als eigenständige Prozedur `map!`[5], welche das Array als weiteren Parameter verwendet:

```
def map!(a)
  for i in 0..a.size-1 do
    a[i] = yield(a[i])
  end
end

a = [1,2,3,4,5]

map!(a) do |x| x + 1 end
```

[3] `https://www.ruby-lang.org/`

[4] Blöcke können in Ruby alternativ auch mit geschweiften Klammern notiert werden, z.B. `b = a.map {|x| x+1}`.

[5] Das Ausrufezeichen im Name von Methoden und Funktionen ist eine Ruby-Konvention, die anzeigt, dass es sich um eine mutierende Methode handelt, also ein übergebenes Objekt verändert wird. Viele Methoden existieren in beiden Varianten.

```
a.each do |x| puts x end
```

Ein an diese Prozedur übergebener Block wird mit dem Schlüsselwort `yield` appliziert; im obigen Beispiel auf ein Argument (`a[i]`). Die Prozedur `map!` applizieren wir dann auf das Array `a` und übergeben zusätzlich noch die Increment-Funktion als Block.[6]

Blöcke entsprechen also funktionalen Parametern. Problematisch wird aber die Verwendung mehrere Blöcke als Argumente oder eines Blocks als Rückgabewert. Hierzu kann man in Ruby aus einem Block mit Hilfe eines `lambda` einen Wert der Klasse `Proc` machen, welcher dann wie jeder andere Wert verwendet werden kann, also als gewöhnlicher Parameter übergeben oder auch in Datenstrukturen gespeichert werden kann.

Die Klasse `Proc` stellt außerdem noch eine Methode `call` zur Verfügung, welche die Funktion, genauer den Block, auf Parameter anwendet. Als Beispiel definieren wir die Funktion `foldr` mit einem expliziten funktionalen Parameter anstelle eines Blocks:

```
def foldr(f,e,a)
  if a == [] then
    e
  else
    f.call(a[0],foldr(f,e,a[1,a.size-1]))
  end
end
```

Die Summe der Elemente eines Arrays können wir dann einfach wie folgt berechnen:

```
puts foldr(lambda do |x,y| x+y end,0,[1,2,3,4,5,6,7,8,9,10])
```

Als nächstes wollen wir ein Array von Funktionen definieren. Hierbei wollen wir an der `i`-ten Position im Array die Funktion ablegen, welche zu einem übergebenen Parameter den Wert `i` hinzuaddiert. Intuitiv könnten wir wie folgt vorgehen:

```
fa = [0,1,2,3,4,5,6,7,8,9]

for i in 0 .. fa.size-1 do  # Iteration über Array, Index beginnt bei 0
  fa[i] = lambda {|x| x+i } # Passende Inkrementfunktion an i-te Position
end                         # schreiben

puts fa[3].call(70)  # Anwenden der Funktion bei Index 3 auf den Wert 70.
```

Führt man dieses Programm aus, so wird aber nicht, wie erwartet der Wert `73` ausgegeben, sondern `79`. Dieses Verhalten liegt darin begründet, dass Variablen in Ruby Speicherzellen entsprechen und während der Programmausführung verändert werden. In diesem Programm addieren wir in der Funktion nicht den Wert, welchen `i` im Schleifenrumpf hat, sondern den Wert, welchen `i` bei der Anwendung der Funktion auf den Wert `70` hat und dies ist eben `9`. Konstruiert man also Funktionsrümpfe, so sollte man

[6] Ruby stellt auch eine vordefinierte Methode `map!` zur Verfügung.

in imperativen Sprachen darauf achten, dass innerhalb des Rumpfes keine veränderbaren Variablen vorkommen. Dies können wir z.B. dadurch erreichen, dass wir das Array von Funktionen mit Hilfe von `map!` konstruieren:

```
fa = [0,1,2,3,4,5,6,7,8,9]

fa.map! { |i| lambda {|x| x + i }}

puts fa[3].call(70)   # Hier erhalten wir nun 73.
```

Nun ist die Variable `i` keine Speicherzelle mehr, welche über die Zeit verändert werden kann. Vielmehr wird sie als Blockparameter bei jeder Anwendung des Blocks neu angelegt, was somit einer Art Scoping entspricht. Unser Programm liefert nun den erwarteten Wert 73.

Funktionen höherer Ordnung in Java

Auch Java bietet ab Version 8 die Möglichkeit, anonyme Funktionen mittels Lambda zu definieren. Hierbei war die Verwendung von funktionalen Parametern mittels anonymer innerer Klassen auch schon vorher möglich. Die spezielle Lambda-Schreibweise erhöht aber entscheidend die Lesbarkeit und lässt erst ein richtiges Gefühl funktionaler Programmierung aufkommen. Als Beispiel definieren wir eine Klasse `Higher`, welche eine nicht-mutierende `map`-Funktion für Listen zur Verfügung stellt:

```
import java.util.*;

class Higher {

  interface Fun<A,B> {  // Def. Interface für Objekte des funktionalen Typs,
    B call (A arg);     // welches eine call-Methode erfordert.
  }

  static <A,B> List<B> map (Fun<A,B> f, List<A> xs) {
    List<B> ys = new ArrayList<B> (xs.size());
    for (A x : xs) {
      ys.add(f.call(x));   // Hier wird das Interface benötigt.
    }
    return ys;
  }
}
```

Bei der Verwendung der `map`-Methode können wir nun einfach unter Verwendung der Lambda-Notation anonyme Implementierungen des Fun-Interfaces definieren, wie das folgende Hauptprogramm zeigt:

```
  public static void main (String[] args) {
    List<Integer> a = Arrays.asList(1,2,3,4,5);
```

```
    a = map(x -> x + 1, a);
    System.out.println(a);  // Ergebnis: [2,3,4,5,6]
  }
```

Hierbei stellt die Lambda-Funktion `x -> x + 1`[7] eine elegante Abkürzung für die Definition einer anonymen inneren Klasse dar, die das Interface `Fun` implementiert.

Als nächstes wollen wir untersuchen, wie sich das Problem der Verwendung von veränderbaren Variablen innerhalb von Funktionsrümpfen (wie oben in Ruby) in Java darstellt. Wir implementieren also wie oben eine Schleife, in welcher wir einer Liste nach und nach passende Lambda-Funktionen hinzufügen:

```
  public static void main (String[] args) {
    List<Fun<Integer,Integer>> fs = new ArrayList<>(10);
    for (int i = 0; i<10; i++) {
      fs.add(x -> x + i);
    }
    System.out.println(fs.get(3).call(70));
  }
```

Bei der Compilation des Programms erhalten wir aber bereits die folgende Fehlermeldung:

```
local variables referenced from a lambda expression must be final or
effectively final
      fs.add(x -> x + i);
```

Java erkennt also, dass die Variable `i` im weiteren Programmverlauf noch verändert wird und deshalb nicht im Rumpf eines Lambda-Ausdrucks verwendet werden darf. Somit ergibt sich für Java die einfache Lösung, eine `final`-Kopie bei jedem Schleifendurchlauf anzulegen und diesen im Funktionsrumpf zu verwenden:

```
  public static void main (String[] args) {
    List<Fun<Integer,Integer>> fs = new ArrayList<Fun<Integer,Integer>>();
    for (int i = 0; i<10; i++) {
      final int j = i;
      fs.add(x -> x + j);
    }
    System.out.println(fs.get(3).call(70));
  }
```

Nun lässt sich das Programm kompilieren und gibt, wie erwartet, `73` aus.

Moderne imperative Programmiersprachen bieten häufig auch die Möglichkeit der Verwendung von funktionalen Parametern und Werten an, wie wir an zwei Beispielen gesehen haben. Hierdurch ist es oft möglich, kürzere, verständlichere Programme zu schreiben und insbesondere auch vordefinierte Funktionen wie `map` und `fold` für Listen oder Arrays zu

[7] Bei mehrstelligen Funktionen, werden die Argumentvariablen durch Kommas getrennt und zusätzlich in Klammern gesetzt, z.B. `(x,y) -> x + y`. Außerdem sind Typanotationen möglich, z.B. `int x -> x + 1`.

verwenden. Der sich ergebende Programmierstil weicht oft stark vom gewohnten imperativen Stil ab, da nicht mehr die Kontrollstrukturen im Vordergrund stehen, sondern die Daten und die Funktionen, mit denen sie manipuliert werden, also der Kern der realisierten Algorithmen.

Wir haben auch den wesentlichen Fallstrick bei der Verwendung von Funktionen höherer Ordnung in imperativen Programmiersprchen kennen gelernt: die Verwendung veränderbarer Variablen in Funktionsrümpfen. Umgehen kann man dieses Problem häufig dadurch, dass man Variablen in diesem Kontext möglichst nicht verändert und einen funktionalen Programmierstil noch stärker pflegt. Speziell in Java stellt aber auch die Verwendung einer `final`-Kopie der veränderbaren Variablen eine Lösung dar.

Funktionen höherer Ordnung in JavaScript

Es gibt aber auch noch andere Fallstricke, die manchmal darin begründet sind, dass auf Grund fehlender strenger Typisierung Dinge implizit geregelt werden. Dies ist insbesondere bei Skriptsprachen der Fall. Betrachten wir z.B. die Sprache JavaScript, die ja in allen Web-Browsern läuft und genutzt wird, um Interaktion in Web-Seiten zu unterstützen. Als neuere Programmiersprache bietet JavaScript natürlich auch Funktionen höherer Ordnung an. So ist z.B. auf Feldern die Funktion `map` definiert, sodass wir alle Elemente eines Feldes wie folgt inkrementieren können:[8]

```
> function inc(x) { return x + 1; }
> [1,2,3].map(inc);
[2, 3, 4]
```

JavaScript unterstützt auch Lambda-Abstraktion mit der Notation "x `=>` e":

```
> [1,2,3].map(x => x + 1);
[2, 3, 4]
```

JavaScript hat viele vordefinierte Operationen, wie z.B. auch die Funktion `parseInt`, mit der man die String-Repräsentation einer Zahl in die Zahl umwandeln kann:

```
> parseInt("5");
5
> parseInt("7");
7
> parseInt("11");
11
```

Somit können wir `parseInt` auch auf alle Elemente eines Feldes von Strings anwenden:

```
> ["5","7","11"].map(parseInt);
[5, NaN, 3]
```

[8] In den meisten Web-Browsern kann man JavaScript-Programme direkt eintippen und ausführen, z.B. bei Firefox oder Chromium durch "`F12 > Console`".

Die Ursache für dieses merkwürdige Verhalten wollen wir hier nicht verraten. Es liegt aber nicht daran, dass der JavaScript-Interpreter kaputt ist, wie man vielleicht vermuten mag, denn die folgende Variante mit einer Lambda-Abstraktion liefert das gewünschte Ergebnis:

```
> ["5","7","11"].map(s => parseInt(s));
[5, 7, 11]
```

Wir haben somit in diesem Kapitel gesehen, dass eine ad-hoc Vermischung unterschiedlicher Konzepte manchmal zur Verwirrung führt. Aus diesem Grund fahren wir nun mit der rein funktionalen Programmierung in Haskell fort, um weitere interessante Programmierkonzepte kennenzulernen.

3.6 Typklassen und Überladung

Anmerkung: Von nun an verwenden wir den vollen Sprachumfang von Haskell. Dies bedeutet, dass wir das Haskell-System nun mit `ghci` (statt `fortprog-ghci`) aufrufen.

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 eine *Typeinschränkung* oder *Typconstraint*, der `a` einschränkt. Möchte man mehrere Typconstraints angeben, muss man diese durch Kommata getrennt in Klammern einfassen.

Konzeptuell bedeutet eine Typeinschränkung, dass bestimmte Funktionen, wie z.B. "`==`", für diesen Typ definiert sein müssen. Aus diesem Grund fasst man die zu einer Typeinschränkung gehörigen Funktionen zu einer Struktur zusammen. Diese Struktur wird *Typklasse* oder auch *Klasse* genannt und besteht aus einer Menge von Funktionsdeklarationen ohne Regeln, was mit einem Interface oder einer Schnittstelle vergleichbar ist.

In Haskell wird eine Typklasse mit dem Schlüsselwort `class` eingeleitet. Dahinter folgt der Name der Typklasse zusammen mit der Typvariablen, für die die Typeinschränkung definiert wird, und danach eine Liste von Funktionstypen. Zum Beispiel ist die Klasse `Eq` in Haskell wie folgt definiert:

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

Da die Funktionen `(==)` und `(/=)` zu der Typklasse `Eq` gehören, werden diese als lokale Funktionen deklariert. Die eigentliche Implementierung dieser Funktionen erfolgt individuell für jeden konkreten Typ, d.h. für jeden Typ kann eine andere Implementierung angegeben werden. Dies ist der wesentliche Unterschied zu dem in Kapitel 3.3 vorgestellten Polymorphismus. Bei polymorphen Funktionen wie in Kapitel 3.3 gibt es nur eine Implementierung für alle Typen, d.h. eine polymorphe Funktion verhält sich auf allen Typen gleich. Wenn wir allerdings für unterschiedliche Typen unterschiedliche Implementierungen haben, so spricht man bei Programmiersprachen auch von *Überladung* (*Overloading*). Letzteres bedeutet im Prinzip, dass wir den Namen einer Funktion mit unterschiedlichen Implementierungen oder Bedeutungen überladen.

Die Implementierung der Funktionen einer Typklasse für einen bestimmten Typ wird auch als *Instanz* der Klasse bezeichnet. Eine solche Instanz wird syntaktisch durch das Schlüsselwort `instance` eingeleitet, danach folgt die Typeinschränkung mit der konkreten Typangabe, und im Rumpf (nach dem Schlüsselwort `where`) folgen die Implementierungen der Funktionen der Typklasse, wobei man hier die Typsignaturen weglässt, weil diese ja schon in der Klassendefinition stehen.

Als Beispiel betrachten wir die Definition einer Instanz von `Eq` auf Binärbäumen für ganze Zahlen:

```
data IntTree = Empty | Node IntTree Int IntTree

instance Eq IntTree 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 `IntTree` verwendbar und wir können z.B. die obige Funktion `elem` auch auf Listen vom Elementtyp `IntTree` anwenden.

Man kann auch polymorphe 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>...
```

Interessant ist hier zu erwähnen, dass durch diese Instanzdefinition unendlich viele Typen zu Instanzen der Klasse `Eq` werden.

3.6.1 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 manchmal sogar muss:

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

Durch diese Vordefinitionen ist es ausreichend, in einer Instanz entweder `(==)` oder `(/=)` zu definieren, allerdings muss mindestens einer dieser Definitionen angegeben werden, weil sonst nichts Sinnvolles definiert wäre.

3.6.2 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.

Eine automatische Instanzdefinition von vordefinierten Klassen (außer `Num`) für selbst definierte Datentypen erreicht man mittels

```
  deriving (κ1, ..., κn)
```

hinter der Datentypdefinition. Hierbei generiert dann der Haskell-Compiler nach bestimmten Schemata automatisch die notwendigen Instanzdefinitionen. Für die Klasse `Eq` haben wir oben schon gesehen, nach welchem Schema diese generierbar sind. Für die Typklasse `Ord` gibt es dagegen mehrere Möglichkeiten. Haskell verwendet als Ordnung die Reihenfolge der Konstruktoren bei der `data`-Definition. Beim oben angegebenen Typ `Ordering` wird mit der Standardordnung `LT < EQ` zu `True` und `GT < EQ` zu `False` ausgewertet. Wenn man eine `deriving`-Annotation für Bäume angibt, also in der Form

```
data Tree a = Empty | Node (Tree a) a (Tree a)
 deriving (Eq, Ord)
```

dann hat dies den Effekt, dass die folgende Instanzdefinition generiert wird:

```
instance Ord a => Ord (Tree a) where
  Empty        <= Empty           = True
  Empty        <= Node _   _  _   = True
  Node _  _ _  <= Empty           = False
  Node tl n tr <= Node tl' n' tr' = tl < tl' ||
                                    (tl == tl' && n < n') ||
                                    (tl == tl' && n == n' && tr <= tr')
```

Hieraus ist ersichtlich, dass die Datenkonstruktoren entsprechend ihrer Definitionsreihenfolge und die Argumente in der gegebenen Reihenfolge geordnet werden.

Aufgabe: Überprüfen Sie die Typen aller in der Vorlesung definierten Funktionen auf ihren allgemeinsten Typ (hierzu müssen Sie bei selbst definierten Funktionen die Typsignatur löschen, wodurch Haskell dann automatisch den allgemeinsten Typ ableitet). Diese lauten zum Beispiel:

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

3.6.3 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'")]
```

3.7 Auswertungsstrategien und Lazy Evaluation

Wir betrachten das folgende Haskell-Programm:

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

Wie wird der Ausdruck `f h` ausgewertet? Da in diesem Ausdruck zwei Funktionsaufrufe vorhanden sind (`f` und `h`), stellt sich die Frage, welcher zuerst ausgewertet wird. Wenn man zunächst `h` auswertet, gerät man in eine Endlosschleife. Wenn man aber zuerst `f` auswertet, kann man die Gleichung für `f` anwenden und den Ausdruck `f h` durch `1` ersetzen. Somit spielt also die Reihenfolge, in der man Dinge auswertet, eine Rolle! Es kommen zwar nicht unterschiedliche Ergebnisse heraus (dank der Seiteneffektfreiheit der funktionalen Programmierung; dies ist z.B. in C anders), allerdings kann man bei einer Reihenfolge ein Ergebnis erhalten, während man bei einer anderen Reihenfolge kein Ergebnis erhält, sondern nicht terminiert.

Welche Auswertungsreihenfolge wird nun also genommen? Bei jeder Programmiersprache ist dies festgelegt. Z.B. würde bei Java zuerst `h` ausgerechnet werden, d.h. man erhält kein Ergebnis. Bei Haskell ist dies allerdings anders. Tatsächlich liefert Haskell den Wert `1`. Um diese Unterschiede präzise zu erfassen, werden wir in diesem Kapitel zunächst einmal den Begriff von Reduktionsstrategien definieren und dabei auch eine formale Definition funktionaler Berechnungen angeben. Danach werden wir dann sehen, welche Strategie Haskell verwendet.

Wie wir schon gesehen haben, werden bei einer Berechnung in Haskell Ausdrücke durch Ausdrücke ersetzt, die bezüglich der Programmregeln als „gleich“ angesehen werden. Um dies präzise zu definieren, benötigen wir einige Grundbegriffe, die wir im folgenden definieren. Damit dies nicht zu aufwändig wird, werden wir einige Vereinfachungen vornehmen. Insbesondere betrachten wir keine Typklassen und polymorphe Typen, wie aus folgender Definition zu ersehen ist.

Definition 3.1 (Programmsignatur) *Eine* Programmsignatur $\Sigma = (S, F)$ *besteht aus einer Menge von Sorten*[9] S *und einer Menge von Funktionssignaturen der Form*[10]

$$f :: s_1 \dots s_n \to s$$

wobei f ein Name ist, $n \geq 0$ eine natürliche Zahl und $s_1, \dots, s_n, s \in S$. Im Fall $n = 0$ lassen wir manchmal auch den Pfeil weg, d.h. wir schreiben einfach $f :: s$ statt $f :: \to s$. Weiterhin soll für jeden Namen f in F nur genau ein solches Element existieren.[11]

Falls $f :: s_1 \dots s_n \to s \in F$, dann heißt f ein n-stelliges Funktionssymbol. *c heißt* Konstante *falls $c :: s \in F$.*

[9] Sorten sind vergleichbar mit Datentypen. Der Begriff „Sorte“ wird allerdings eher in der Logik verwendet.

[10] Da wir hier auch keine Funktionen höherer Ordnung betrachten, schreiben wir Funktionssignaturen in einer nicht-curryfizierten Schreibweise.

[11] Dies bedeutet, dass wir Überladung, wie dies mit Typklassen möglich ist, ausschließen.

Programmsignaturen legen somit die Typen und Funktionssymbole fest, die wir in einem Programm verwenden können. Zum Beispiel ist $\Sigma = (S, F)$ mit $S = \{\texttt{Int}, \texttt{Bool}\}$ und

$$\begin{aligned} F \;=\; \{&\texttt{+} :: \texttt{Int Int} \to \texttt{Int},\ \texttt{*} :: \texttt{Int Int} \to \texttt{Int},\\ &\texttt{square} :: \texttt{Int} \to \texttt{Int},\ \texttt{True} :: \texttt{Bool},\ \texttt{False} :: \texttt{Bool},\ \ldots\} \end{aligned}$$

eine Programmsignatur. Wir nehmen an, dass alle Literale als Konstanten in der Programmsignatur vordefiniert sind, z.B. ist $0 :: \to \texttt{Int} \in F$.
Statt Ausdrücke verwenden wir auch oft den Begriff „Terme“, weil diese Begriffe auch aus dem Bereich der Termersetzung stammen. Terme müssen sinnvoll strukturiert sein, d.h. sie enthalten nur Symbole aus einer Programmsignatur und Variablen:

Definition 3.2 (Terme) *Sei $\Sigma = (S, F)$ eine Programmsignatur und X eine Menge von* Variablen *(der Form $x :: s$), welche disjunkt zu den Symbolen in Σ sind, d.h. die Variablennamen x sind verschieden von den Funktionssymbolen. Dann ist die Menge der* Terme *$T_\Sigma(X)_s$ der Sorte s über Σ und X die kleinste Menge, die folgende Bedingungen erfüllt:*

1. *Für alle Konstanten $c :: s \in F$ gilt $c \in T_\Sigma(X)_s$*
2. *Für alle Variablen $x :: s \in X$ gilt $x \in T_\Sigma(X)_s$*
3. *Für alle n-stelligen Funktionen $f :: s_1 \ldots s_n \to s \in F$ und $t_i \in T_\Sigma(X)_{s_i}$, $i = 1, \ldots, n$, ist $(f\ t_1 \ldots t_n) \in T_\Sigma(X)_s$*

Mit $T_\Sigma(X)$ bezeichnen wir die Menge aller Terme einer gegebenen Signatur, d.h.

$$T_\Sigma(X) = \{t \mid s \in S, t \in T_\Sigma(X)_s\}$$

Beachte:

- Es handelt sich hierbei um eine “induktive Definition”, d. h. komplexe Fälle werden induktiv durch Rückführung auf einfache Fälle definiert. Dies ist typisch in der Informatik, um komplexere Strukturen zu definieren.
- Intuitiv bedeutet diese Definition, dass Terme also Konstanten, Variablen oder Kombinationen mit passenden Argumenten sind.
- Obwohl bei dieser Definition das Funktionssymbol immer am Anfang steht, werden wir in konkreten Beispielen auch Infixsymbole wie in Haskell verwenden.

Beispiel: Sei Σ wie oben und $X = \{x :: \texttt{Int},\ y :: \texttt{Int},\ z :: \texttt{Int}\}$. Dann gilt:

`(square x)` $\in T_\Sigma(X)_{\texttt{Int}}$

`(1 + 2)` $\in T_\Sigma(X)_{\texttt{Int}}$

`(square (square (y + z)))` $\in T_\Sigma(X)_{\texttt{Int}}$

Damit können wir nun exakt definieren, was wir unter einem (funktionalen) Programm verstehen:

Definition 3.3 (Programm, Termersetzungssystem) *Ein* Programm *oder* Termersetzungssystem *ist ein Tripel* (Σ, X, R) *mit*

- Σ *ist eine Programmsignatur*
- X *ist eine Variablenmenge disjunkt zu* Σ
- R *ist eine Menge von Regeln der Form*

$$f\ t_1 \ldots t_n\ \texttt{=}\ r$$

wobei $(f\ t_1 \ldots t_n), r \in T_\Sigma(X)_s$ *für eine Sorte* s *(die Ergebnissorte der Funktion* f*). Hierbei dürfen in* r *nur Variablen vorkommen, die auch in* $(f\ t_1 \ldots t_n)$ *vorkommen.* $f\ t_1 \ldots t_n$ *wird auch als als linke Seite und* r *als rechte Seite der Regel bezeichnet.*

Formal sind die linke und rechte Seite einer Regel ein Terme, aber wir lassen bei der konkreten Notation die äußeren Klammern um die linke Seite weg, um die Ähnlichkeit zu Haskell herauszustellen. Ein Haskell-Programm kann also als ein Termersetzungssystem angesehen werden, wobei in Haskell noch mehr erlaubt ist.

Bei der Anwendung einer Regel werden die Variablen der linken Seite durch andere Terme ersetzt, so dass die Regel passt. Um dies genau zu beschreiben, benötigen wir den Begriff der Substitution:

Definition 3.4 (Substitution) *Sei* $\Sigma = (S, F)$ *eine Programmsignatur. Eine* Substitution σ *ist eine Abbildung* $\sigma : X \to T_\Sigma(X)$*, die jeder Variablen* $x :: s \in X$ *einen Term* $t \in T_\Sigma(X)_s$ *zuordnet.*
Die Anwendung einer Substitution σ *auf einen Term* $t \in T_\Sigma(X)$*, geschrieben* $t\sigma$*, ist wie folgt induktiv definiert:*

- *Falls* $t :: s \in X$*, dann ist* $t\sigma = \sigma(t)$*.*
- *Falls* $t :: s \in F$ *(also falls* t *eine Konstante ist), dann ist* $t\sigma = t$*.*
- *Falls* $t = (f\ t_1 \ldots t_n)$*, dann ist* $t\sigma = (f\ t_1\sigma \ldots t_n\sigma)$*.*

Somit ersetzt die Substitution nur Variablen – alles andere bleibt unverändert. Üblicherweise fordert man, dass eine Substitution σ nur endlich viele Variablen verändert, d.h. die Menge

$$\{x \mid x :: s \in X \text{ mit } \sigma(x) \neq x\}$$

endlich ist. Dann kann σ auch als endliche Menge von Paaren der Form

$$\{x \mapsto \sigma(x) \mid x :: s \in X \text{ mit } \sigma(x) \neq x\}$$

dargestellt werden.

Beispiel: Sei $\sigma = \{\texttt{x} \mapsto 2,\ \texttt{y} \mapsto \texttt{(square z)}\}$ eine Substitution. Dann ist

$$\texttt{(x * (y + z))}\sigma = \texttt{2 * ((square z) + z)}$$

Zum Ausrechnen eines Terms werden bestimmte Ersetzungen an bestimmten Positionen im Term vorgenommen. Daher müssen wir noch definieren, wie Teilterme eines Terms an bestimmten Stellen ersetzt (ausgerechnet) werden. Dazu benötigen wir den Begriff einer Position:

Definition 3.5 (Position) *Eine Position identifiziert eindeutig eine Stelle in einem Term. Ist t ein Term, dann ist $Pos(t)$ die Menge aller Positionen in einem Term t. Üblicherweise werden die Positionen als Folgen natürlicher Zahlen angegeben, wobei die leere Folge $\langle\rangle$ als die Position der Wurzel definiert ist, also als der Einstiegspunkt des Terms.*

Beispiel: Sei $t = (x \ * \ (y \ + \ z))$ und $Pos(t) = \{\langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 2,1\rangle, \langle 2,2\rangle\}$, dann entspricht $\langle\rangle$ dem Einstiegspunkt des Terms. $\langle 1\rangle$ entspricht dem ersten Argument, also dem x. $\langle 2\rangle$ entspricht dem Teilterm $(y \ + \ z)$ und $\langle 2,1\rangle$ wiederum dem y innerhalb des Teilterms. Das z innerhalb des Teilterms ist mit $\langle 2,2\rangle$ indiziert.

Definition 3.6 (Teilterm) *Ist $p \in Pos(t)$, dann heißt $t|_p$ Teilterm von t an der Position p und ist wie folgt definiert:*

- *Falls $p = \langle\rangle$, dann ist $t|_{\langle\rangle} = t$.*
- *Falls $p = \langle p_1, \ldots, p_k\rangle$ und $t = (f\ t_1 \ldots t_n)$, dann ist $t|_{\langle p_1,\ldots,p_k\rangle} = t_{p_1}|_{\langle p_2,\ldots,p_k\rangle}$.*

Beispiel: Sei $t = (x \ * \ (y \ + \ z))$ und es sei $t|_{\langle 2,2\rangle}$ zu bestimmen. $t|_{\langle 2\rangle}$ wäre $(y \ + \ z)$ und $t|_{\langle 2,2\rangle}$ entsprechend dann nur noch z.

Definition 3.7 (Ersetzung) *Ist $p \in Pos(t)$ und t' ein Term, dann heißt $t[t']_p$* Ersetzung *von $t|_p$ durch t' an der Stelle p und ist wie folgt definiert:*

- *Falls $p = \langle\rangle$, dann ist $t[t']_{\langle\rangle} = t'$.*
- *Falls $p = \langle p_1, \ldots, p_k\rangle$ und $t = (f\ t_1 \ldots t_n)$, dann ist*

$$t[t']_p = (f\ t_1 \ldots t_{p_1-1}\ t_{p_1}[t']_{\langle p_2 \ldots p_k\rangle}\ t_{p_1+1} \ldots t_n)$$

Beispiel: Sei $t = (x \ * \ (y \ + \ z))$ und $t' = (z \ + \ y)$, dann ist $t[t']_{\langle 2\rangle} = (x \ * \ (z \ + \ y))$.

Damit haben wir jetzt alles Grundlagen, um Berechnungen im Substitutionsmodell formal zu beschreiben:

Definition 3.8 (Reduktionsschritt) *Sei $P = (\Sigma, X, R)$ ein Programm. Dann ist ein* Reduktionsschritt *$t_1 \Rightarrow t_2$ definiert falls $p \in Pos(t_1)$, $l \ = \ r \in R$ und eine Substitution σ existiert mit der Eigenschaft*

- *$l\sigma = t_1|_p$ (σ ersetzt formale Parameter durch die aktuellen)*
- *$t_2 = t_1[r\sigma]_p$ (ersetze Teilterm durch rechte Regelseite nach Substitution der formalen Parameter)*

Eine Reduktion *oder* Berechnung $t_1 \Rightarrow^* t_2$ *ist eine (eventuell leere) Folge von Schritten*

$$s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n$$

mit $s_1 = t_1$ *und* $s_n = t_2$.

Beispiel: R enthalte die Regel

```
square x = (x * x)
```

Dann gilt:

```
(square (1 + 2)) ⇒ ((1 + 2) * (1 + 2))
```

wobei hier $p = \langle\rangle$ und $\sigma(\texttt{x}) = \texttt{(1 + 2)}$ ist. Weiterhin gilt:

```
(3 * (square 4)) ⇒ (3 * (4 * 4))
```

wobei hier $p = \langle 2 \rangle$ und $\sigma(\texttt{x}) = 4$ ist.

Auswertung vordefinierter Funktionen: Die Auswertung vordefinierter Funktionen wie "+" oder "*" kann man konzeptionell definieren durch eine unendliche Menge von Regeln:

```
0 + 0 = 0          0 * 0 = 1
0 + 1 = 1          ...
0 + 2 = 2          1 * 2 = 2
...                ...
1 + 2 = 3          3 * 3 = 9
...                ...
```

Somit kann `(square (1 + 2))` auf folgende Weise ausgewertet werden:

```
(square (1 + 2))
⇒ ((1 + 2) * (1 + 2))
⇒ (3 * (1 + 2))
⇒ (3 * 3)
⇒ 9
```

Es ist jedoch auch möglich, den Term auf folgende Weise auszuwerten:

```
(square (1 + 2))
⇒ (square 3)
⇒ (3 * 3)
⇒ 9
```

Wie wir sehen, tauchen bei den Reduktionen unterschiedliche Zwischenergebnisse auf, aber das Endergebnis ist identisch. Daher wollen wir zunächst einmal festlegen, was ein Endergebnis ist:

Definition 3.9 (Normalform) *Ein Term t heißt in* Normalform*, falls es keinen Term t' gibt mit $t \Rightarrow t'$, d. h. falls kein weiterer Reduktionsschritt möglich ist.*

Somit entsprechen Normalformen den Endergebnissen beziehungsweise den *Werten* von Ausdrücken. Das letzte Beispiel hat gezeigt, dass es unterschiedliche Strategien gibt, wie man Reduktionen durchführen kann. Wir betrachten im Folgenden zwei Strategien etwas genauer.

Wertaufruf/innermost/call-by-value/strikte Auswertung

Informell bedeutet der Wertaufruf (call-by-value oder auch strikte Auswertung), dass die Argumente eines Funktionsaufrufs *vor* der Funktionsanwendung ausgewertet werden. Formal können wir dies wie folgt definieren:

Definition 3.10 *Ein Reduktionsschritt $t_1 \Rightarrow t_2$ mit $t_2 = t_1[r\sigma]_p$ heißt* Wertaufruf *(*innermost*,* call-by-value*,* strikte *Auswertung oder auch Aufruf in* applikativer Ordnung*), falls $t_1|_p = (f\ s_1 \ldots s_n)$ und jedes s_i in Normalform ist.*

Namensaufruf/outermost/call-by-name/nicht-strikte Auswertung

Informell bedeutet der Namensaufruf (call-by-name oder nicht-strikte Auswertung), dass zunächst die äußeren Prozeduraufrufe vor den Aufrufen in den Argumenten ersetzt werden, bis nur noch Aufrufe elementarer Prozeduren vorhanden sind, die dann ausgewertet werden. Formal können wir dies wie folgt festlegen:

Definition 3.11 *Ein Reduktionsschritt $t_1 \Rightarrow t_2$ mit $t_2 = t_1[r\sigma]_p$ heißt* Namensaufruf *(*outermost*,* call-by-name*,* nicht-strikte *Auswertung oder auch Aufruf in* Normalordnung*), falls $t_1|_p$ ein äußerster Teilterm ist, an dem ein Reduktionsschritt möglich ist. Das heißt falls $p = \langle p_1 \ldots p_k \rangle$, dann ist ein Reduktionsschritt an $\langle p_1 \ldots p_i \rangle$ mit $i < k$ nicht möglich.*

Im obigen Beispiel war die zweite Auswertung strikt (call-by-value), während die erste Auswertungsfolge nicht-strikt (call-by-name) ist:

```
(square (1 + 2))
⇒ ((1 + 2) * (1 + 2))
⇒ (3 * (1 + 2))
⇒ (3 * 3)
⇒ 9
```

Hier sieht man, dass zwar eine Doppelauswertung von `(1 + 2)` stattfindet, aber das Endergebnis identisch zur strikten Auswertung ist. Allerdings kann eine nicht-strikte Auswertung für einen Ausdruck ein Ergebnis liefern, bei dem die strikte Auswertung nicht terminiert. Die nicht-strikte Auswertung liefert immer eine Normalform zurück, sofern eine existiert. Dafür ist die strikte Auswertung häufig effizienter implementierbar. Betrachten wir dazu noch einmal unser Eingangsbeispiel:

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

Der Ausdruck `f h` wird mit nicht-strikter Auswertung zu `1` ausgewertet, während eine strikte Auswertung nicht terminiert.

Bei dem Ausdruck `((square 2) + (square 5))` gibt es übrigens mehrere innermost Positionen. Daher zeichnet man manchmal noch solche Strategien aus, die z.B. das linkeste innerste Funktionssymbol auswerten, d.h. man spricht dann von

- *leftmost-innermost* (LI)
- *rightmost-innermost* (RI)
- *leftmost-outermost* (LO)
- *rightmost-outermost* (RO)

Strategien. Im Allgemeinen ist eine *Reduktionsstrategie* eine (partielle) Abbildung von Termen in Positionen, die festlegt, an welchen Positionen die nächsten Reduktionsschritte stattfinden. Hierbei kann es manchmal auch sinnvoll sein, an mehreren Positionen parallel zu reduzieren. Z.B. könnte man bei

```
((square 2) + (square 5))
```

gleichzeitig an den Position $\langle 1 \rangle$ und $\langle 2 \rangle$ reduzieren. Man spricht dann auch von *parallel innermost* (PI) und *parallel outermost* (PO) Reduktion.

Es gibt sowohl strikte (LI) funktionale Sprachen (Lisp, Scheme, Erlang, Standard ML) als auch nicht-strikte (LO) funktionale Sprachen (Haskell). Allgemein sind nicht-strikte Sprachen berechnungsstärker, wie wir oben gesehen haben. Allerdings kann diese Berechnungsstärke einen Effizienzgewinn wie auch einen Effizienzverlust mit sich bringen. Letzteres kann passieren, wenn Argumente bei einem Reduktionsschritt verdoppelt werden. Haskell vermeidet dies durch Lazy Auswertung, wie wir noch sehen werden.

Erwähnenswert ist noch, dass es keine rein strikte Programmiersprache gibt. Zumindest muss immer die Fallunterscheidung (if-then-else) nicht strikt ausgewertet werden: zuerst wird nur die Bedingung ausgewertet und die Auswertung der übrigen Teile erfolgt, je nach Wert der Bedingung, erst später.

Ein Vorteil der outermost Reduktion liegt darin, dass sie *berechnungsvollständig* ist: Alles, was irgendwie berechnet werden kann, wird durch eine PO-Strategie berechnet (meistens auch durch eine LO-Strategie, aber es gibt auch Ausnahmen). Darüberhinaus hat eine nicht-strikte Auswertung auch praktische Vorteile: Sie unterstützt

- die Vermeidung überflüssiger (ggf. unendlicher) Berechnungen, und
- das Rechnen mit unendlichen Datenstrukturen.

Zum Beispiel definiert die folgende Funktion `from` die aufsteigende, unendliche Liste der natürlichen Zahlen ab der Zahl `n`:

```
from :: Int → [Int]
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 zu `[1]` ausgewertet, 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 Fibonacci-Funktion. Um die n-te Fibonacci-Zahl zu erhalten, erzeugen wir die Liste aller Fibonacci-Zahlen 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`.

Weil das Rechnen mit unendlichen Strukturen oft nützlich ist, gibt es in Haskell auch einige vordefinierte Funktionen zum Erzeugen unendlicher Listen. Z.B. liefert `repeat` eine unendliche Liste mit identischen Elementen:

```
repeat :: a → [a]
repeat x = x : repeat x
```

Somit liefert `take 70 (repeat '-')` eine textuelle Linie.
Eine unendliche Liste mit wiederholter Anwendung einer Funktion kann mit der folgenden Funktion erzeugt werden:

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

Zur Übung sollte man sich überlegen, was `iterate (+1) 0` liefert.
Wie wir an der Funktion `from` gesehen haben, ist es in Haskell recht einfach, unendliche arithmetische Sequenzen zu berechnen. Tatsächlich können wir mit folgenden Funktionen *arithmetische Sequenzen* und *Intervalle* mit der Schrittweite `1` und auch mit beliebiger Schrittweite definieren:

```
from :: Int → [Int]
from n = n : from (n + 1)

fromThen :: Int → Int → [Int]
fromThen n1 n2 = let d = n2-n1 in n1 : fromThen (n1+d) (n2+d)

fromTo :: Int → Int → [Int]
fromTo n m = if n>m then [] else n : fromTo (n + 1) m

fromThenTo :: Int → Int → Int → [Int]
fromThenTo n1 n2 m =
  let d = n2-n1
  in if d>=0 && n1>m || d<0 && n1<m
       then []
       else n1 : fromThenTo (n1+d) (n2+d) m
```

Weil diese Funktionen[12] ganz praktisch sind, hat Haskell auch eine spezielle Syntax dafür:

```
[n1 .. ]        steht für   from n
[n1,n2 .. ]     steht für   fromThen n1 n2
```

[12] In Haskell haben diese Funktionen noch den Namenspräfix `enum`.

```
[n .. m]        steht für   fromTo n m
[n1,n2 .. m]    steht für   fromThenTo n1 n2 m
```

Somit gelten z.B. folgende Gleichheiten:

```
[1..4]          == [1,2,3,4]
take 5 [2..]    == [2,3,4,5,6]
take 5 [2,4..] == [2,4,6,8,10]
[1,3..10]       == [1,3,5,7,9]
take 5 [3,1..] == [3,1,-1,-3,-5]
```

Tatsächlich kann man nicht nur ganze Zahlen aufzählen, sondern man kann auch Gleitkommazahlen und Zeichen aufzählen:

```
> [1,1.5 .. 10]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5,7.0,7.5,8.0,8.5,9.0,9.5,10.0]
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
> take 20 ['A' ..]
"ABCDEFGHIJKLMNOPQRST"
```

Damit man die Notation für Sequenzen für alle diese verschiedenen Typen nutzen kann, enthält die Prelude von Haskell die Typklasse

```
class Enum a where
  succ, pred :: a  → a
  toEnum :: Int  → a
  fromEnum :: a  → Int
  enumFrom :: a  → [a] -- [n..]
  enumFromThen :: a  → a  → [a] -- [n1,n2..]
  enumFromTo :: a  → a  → [a] -- [n..m]
  enumFromThenTo :: a  → a  → a  → [a] -- [n1,n2..m]
```

Damit kann man die Notation für Sequenzen auf allen Instanzen dieser Typklasse verwendet werden. Z.B. sind die Typen `Int`, `Float`, `Char`, `Bool` und auch `Ordering` Instanzen von `Enum`, sodass gilt:

```
> [LT ..]
[LT,EQ,GT]
```

Bei Datentypen mit endlich vielen Werten sind die Instanzen so definiert, dass auch nur endliche Sequenzen entstehen.
In diesem Zusammenhang ist auch die in der Prelude definierte Typklasse `Bounded` interessant, die Datentypen mit minimalen und und maximalen Werten beschreibt:

```
class Bounded a where
  minBound, maxBound :: a
```

Z.B. sind die Typen `Int`, `Char`, `Bool` und auch `Ordering` Instanzen dieser Typklasse:

```
> minBound :: Bool
False
> maxBound :: Bool
True
> maxBound :: Int
9223372036854775807
```

Man kann Instanzen von `Enum` und `Bounded` auch automatisch ableiten lassen, wenn der Datentyp eine Aufzählung von Konstanten ist:

```
data Color = Red | Blue | Yellow
 deriving (Show, Enum, Bounded)
```

Hier werden dann die Konstruktoren in ihrer textuellen Reihenfolge geordnet:

```
> maxBound :: Color
Yellow
> [Red ..]
[Red,Blue,Yellow]
```

Arithmetische Sequenzen können oft verwendet werden. Z.B. können wir die Fakultätsfunktion auch wie folgt definieren:

```
fac n = foldr (*) 1 [1 .. n]
```

Wir können auch eine Liste von Objekten durchnummerieren, indem wir aus den Elementen Paare mit einem Index machen:

```
> zip [1..] "abcde"
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
```

Dies können wir natürlich auch mit anderen bekannten Funktionen kombinieren:

```
> take 5 (map (uncurry (++)) (zip (map show [1..]) (repeat ". Zeile")))
["1. Zeile","2. Zeile","3. Zeile","4. Zeile","5. Zeile"]
```

Sharing

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 Verbesserung 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 wie folgt aus:

```
double x = x + x

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

Dann verläuft die Auswertung so, wie auf Abbildung 3.3 dargestellt. Die schwarzen Linien zeigen dabei jeweils einen Reduktionsschritt an, blaue Linien sind Zeiger auf Ausdrücke.

Die Programmiersprache Haskell basiert auf Lazy-Auswertung, also der Kombination von LO-Auswertung mit Sharing.[13] Formalisiert wurde diese Strategie im Jahre 1993 von Launchbury [10]. 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.

Ein weiterer Vorteil der Lazy-Auswertung ist die schöne Komponierbarkeit von Funktionen: Angenommen, wir hätten eine Generator-Funktion, z.B. vom Typ `gen :: `$\alpha \to \beta$, und eine Konsumenten-Funktion, z.B. vom Typ `con :: `$\beta \to \gamma$. Wenn wir beide Funktionen komponieren durch `con . gen`, dann entstehen durch die Lazy-Auswertung keine großen Zwischendatenstrukturen, sondern es stehen nur die Teile, welche zur gegebenen Zeit benötigt werden, im Speicher und diese können danach direkt wieder freigegeben werden. Wenn wir z.B. die 100. Fibonacci-Zahl durch den Ausdruck `fibs!!99` ausrechnen, dann müssen prinzipiell während der Berechnung zu einem Zeitpunkt immer nur zwei Zahlen im Speicher gehalten werden. Dies gilt genauso, wenn wir große Dateien schrittweise verarbeiten.

[13] Dies ist etwas ungenau: Haskell übersetzt die Muster in den linken Regelseiten zunächst in einfachere standardisierte Muster, auf die dann die LO-Auswertung angewendet wird.

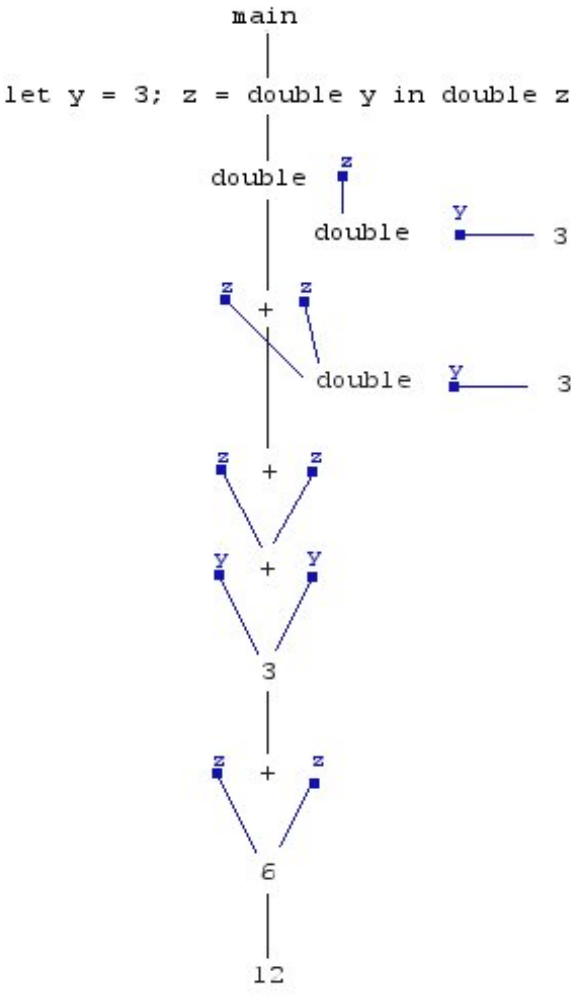


Abbildung 3.3: Sharing bei lazy evaluation

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

```
ones = 1 : ones
```

Diese kann sogar mit konstantem Speicherbedarf dargestellt werden, wie dies in der Abbildung 3.4 skizziert ist.

Abschließend wollen wir noch darauf hinweisen, dass die Lazy Auswertung durch Sharing nur die mehrfache Auswertung von Ausdrücken vermeidet, die durch mögliche Duplikation von Argumenten auf der rechten Seite (vgl. `double`) entsteht. Wenn dagegen Ausdrücke mehrfach im Quellcode stehen, wie z.B. in

```
fib100_plus = fib 100 + fib 100
```

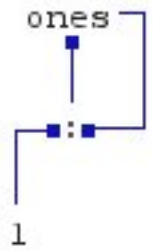


Abbildung 3.4: Zyklische Liste `ones`

dann werden diese unabhängig voneinander ausgewertet. Somit würde also bei der Berechnung von `fib100_plus` einmal das linke Argument `fib 100` und einmal das rechte Argument `fib 100` ausgewertet werden. Wenn wir dagegen die Definition

```
fib100_double = double (fib 100)
```

betrachten, dann wird bei der Berechnung von `fib100_double` der Teilausdruck `fib 100` durch die lazy Strategie nur einmal berechnet. Dies wird auch durch die folgende Eigenschaft der Lazy Auswertung beschrieben:

Bei Lazy Auswertung wird jeder Teilausdruck *höchstens einmal* ausgewertet.

3.8 List Comprehensions

Wir wir schon in Kapitel 3.7 auf Seite 74 gesehen haben, stellt Haskell mit den arithmetischen Sequenzen ein wenig syntaktischen Zucker für Listendefinitionen zur Verfügung. Darüberhinaus können wir sogar Listen mit einer an die Mathematik angelehnten Notation beschreiben, die auch als *list comprehension* bezeichnet wird. So liefert der Ausdruck

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

die Liste `[(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4)]`. Und der Ausdruck

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

liefert die unendliche Liste aller endlichen Anfangsfolgen der Menge der natürlichen Zahlen.

Erlaubt sind in List Comprehensions neben *Generatoren* wie `i <- [1..3]` oder `n <- [0..]` auch *boolesche Bedingungen* wie `i /= j` und auch lokale Definitionen mittels `let`, wobei man dann kein "`in`" schreiben muss. Der folgende Ausdruck zeigt ein einfaches Beispiel für die Verwendung von Generatoren, lokalen Definitionen und Bedingungen:

```
[ (x,y,z) | x <- [1 .. 4], y <- [2 .. 6], let z = x+y, x /= y ]
```

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]
```

Eine praktische Anwendung, bei der die Verwendung von list comprehensions sehr angenehm ist, ist das Parsen von Werten.

Als Beispiel betrachten wir CSV-Dateien (comma separated values), mit der wir eine Liste von Listen von Werten in eine Datei speichern können.

```
data CSV a = CSV [[a]]
```

Wir geben zunächst eine Instanz für `Show` an:

```
instance Show a => Show (CSV a) where
  show (CSV xss) = concat (map (++"\n") (map separate xss))
    where
      separate []     = ""
      separate [x]    = show x
      separate (x:xs) = show x ++ "," ++ separate xs
```

Es folgt die Instanz für `Read`. Hierbei verwenden wir wieder die im Kapitel zu Klassen vorgestellte Idee des Parsens. Die Funktion

```
readsPrec :: Read a => Int  → String  → [(a, String)]
```

verwendet einen zusätzlichen Parameter, mit welchem Präzedenzen bei der Klammerung notiert werden können. Diesen ignorieren wir hier einfach bzw. leiten ihn einfach unverändert an die Unterparser weiter:

```
instance Read a => Read (CSV a) where
  readsPrec p s = case s of
    []       → [ (CSV []          , []) ]
    '\n':s1  → [ (CSV ([]:xs)     , s2)
                 | (CSV xs        , s2) <- readsPrec p s1 ]
    ',' :s1  → [ (CSV ((x:xs):xss), s3)
                 | (x             , s2) <- readsPrec p s1
                 , (CSV (xs:xss), s3) <- readsPrec p s2 ]
    _        → [ (CSV ((x:xs):xss), s2)
                 | (x             , s1) <- readsPrec p s
                 , (CSV (xs:xss), s2) <- readsPrec p s1 ]
```

Da wir den Präzedenzparameter `p` nicht berücksichtigen wollen, reichen wir ihn bei rekursiven Aufrufen einfach mittels `readsPrec p s` weiter. Alternativ hätten wir auch kürzer `reads s` schreiben können, was eine Belegung des Präzedenzparameters mit dem initialen Wert zur Folge hätte.

3.9 Ein- und Ausgabe

Haskell ist eine rein funktionale Sprache, d.h. Funktionen haben keine Seiteneffekte. Wie kann Ein- und Ausgabe in so einer 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
```

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 der Lazy-Auswertung 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
```

Hier soll das Semikolon "`;`" bewirken, dass `Hi` zweimal hintereinander ausgegeben wird. Das Problem dabei ist jedoch, dass wegen der Lazy-Auswertung `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 Lazy-Auswertung die gewünschte Eingabereihenfolge. All diese Probleme werden in Haskell mit Monaden gelöst.

3.9.1 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
```

Hierbei wird durch `repeat (putStr "Hi")` eine unendliche Liste von Ausgabeaktionen erzeugt.

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.

3.9.2 `do`-Notation

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

```
do a1
   ⋮
   an
```

(man beachte die Layout-Regel nach dem `do`!) steht eine alternative, „imperative“ Notation zur Verfügung. Hierbei steht

```
do p <- e1
   e2
```

für den Bind-Aufruf

```
e1 >>= \p  → e2
```

und die Anweisungssequenz

```
do e1
   e2
```

steht für den Aufruf des Sequenzoperators

```
e1 >> e2
```

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)
```

3.9.3 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`.

3.9.4 Lesen und Schreiben von Dateien

Die Bibliotheken von Haskell bieten, basierend auf dem monadischen I/O-Konzept, viele Möglichkeiten, um mit der Umgebung zu kommunizieren, z.B. Lesen und Schreiben von Dateien, Datenbanken oder auch Socketverbindungen für Netzwerkanwendungen. Eine sehr einfache zu benutzende Methode zum Lesen und Schreiben von Dateien ist mit den I/O-Aktionen

```
readFile  :: FilePath  → IO String        -- reads a file with a given name
writeFile :: FilePath  → String  → IO ()  -- writes a file with some content
```

schon vordefiniert. Hierbei ist `FilePath` ein Typsynonym für `String`, durch das dokumentiert werden soll, dass das erste Argument ein Dateiname ist. Z.B. können wir durch

```
copyFile :: FilePath → FilePath → IO ()
copyFile fromfile tofile = readFile fromfile >>= writeFile tofile
```

eine Operation definieren, mit der man eine Datei kopieren kann. Weil die Lese- und Schreiboperationen auf Dateien lazy ausgeführt werden, können auch große Dateien ohne nennenswerten Speicherverbrauch kopiert werden.

Mit den uns schon bekannten Funktionen können wir Dateien einfach analysieren. Die Größe einer Datei können wir wie folgt ausgeben:

```
> readFile "FILE" >>= print . length
```

Die Anzahl der Zeilen in einer Datei können wir so berechnen:

```
> readFile "FILE" >>= print . length . lines
```

Hier können wir sehen, wie man die Funktionskomposition und den damit verbundenen Kombinatorstil gut anwenden kann. Als weiteres Beispiel wollen wir die Anzahl der Leerzeilen in einer Datei zählen:

```
> readFile "FILE" >>= print . length . filter (all (==' ')) . lines
```

Hierbei prüft die vordefinierte Funktion `all`, ob alle Elemente einer Liste ein gegebenes Prädikat erfüllen (man mache sich klar, warum diese Definition genau dies ausdrückt):

```
all :: (a → Bool) → [a] → Bool
all p = foldr (&&) True . map p
```

Abschließend wollen wir noch sehen, wie einfach man durch Anwendung von Funktionen höherer Ordnung und unendlichen Datenstrukturen die Zeilen einer Datei nummeriert ausgeben kann. Hierzu definieren wir uns eine Funktion, die einen Text zeilenweise nummeriert:

```
enumerateLines :: String → String
enumerateLines = concat . map (++"\n")
                        . map (uncurry (++))
                        . zip (map (\n → show n ++ ": ") [1..])
                        . lines
```

Wenn nun die Datei `FILE` den Inhalt

```
Dies ist
eine Datei
mit drei Zeilen.
```

enthält, dann wird diese wie folgt durchnummeriert:

```
> readFile "FILE" >>= putStrLn . enumerateLines
1: Dies ist
2: eine Datei
3: mit drei Zeilen.
```

Durch Verwendung einiger vordefinierter Funktionen können wird den Code noch verbessern. Da z.B. die Kombination von `concat` und `map` oft benutzt wird, ist diese Kombination wie folgt vordefiniert:

```
-- Maps a function from elements to lists and merges the result into one list.
concatMap :: (a → [b]) → [a] → [b]
concatMap f = concat . map f
```

Damit können wir die Definition von `enumerateLines` wie folgt vereinfachen:

```
enumerateLines :: String → String
enumerateLines = concatMap (++"\n")
                         . map (uncurry (++))
                         . zip (map (\n → show n ++ ": ") [1..])
                         . lines
```

Schaut man sich die erste Funktion genauer an, dann erkennt man, dass diese eine Liste von Strings zu einem String zeilenweise zusammenfügt, indem Zeilenumbrüche zwischen den einzelnen Strings gesetzt werden. Da auch dies öfters benutzt wird, ist diese Kombination vordefiniert:

```
-- Concatenates a list of strings with terminating newlines.
unlines    :: [String] → String
unlines ls = concatMap (++"\n") ls
```

Damit können wir die Definition von `enumerateLines` noch einmal vereinfachen:

```
enumerateLines :: String → String
enumerateLines = unlines . map (uncurry (++))
                         . zip (map (\n → show n ++ ": ") [1..])
                         . lines
```

3.10 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.

In größeren Anwendungen möchte man die Funktionalitäten anderer Module in eigenen Modulen verwenden. Hierzu kann man eine `import`-Deklaration angeben, durch die von dem importierten Modul exportierten Namen in dem aktuellen Modul sichtbar gemacht werden, wie z.B. in dem folgenden Hauptmodul:

```
module Main where

import Nats

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

Bei `import`-Deklarationen 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))
```

Darüber hinaus 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.

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.[14] Dann kann man ein ausführbares Maschinenprogramm `myexecutable` z.B. durch

```
ghc -o myexecutable Main.hs
```

erzeugen. Wenn wir dies mit unserem obigen Hauptprogramm durchführen, erhalten wir eine ausführbare Datei `myexecutable`, die wir dann direkt ausführen können:

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

3.11 Funktoren und Monaden

Wir haben den Begriff „Monade" im Zusammenhang mit der Ein/Ausgabe kurz gesehen. In diesem Kapitel wollen wir dies etwas vertiefen, weil Monaden nicht nur für die Ein/-Ausgabe nützlich sind, sondern allgemein verwendet werden können, um Berechnungen besser zu strukturieren.

Tatsächlich kann man Berechnungen auf verschiedene Arten strukturieren und solche generellen Strukturen in Haskell gut beschreiben, wie wir sehen werden. In diesem Kapitel wollen wir insbesondere lernen, wie man ähnliche Programmiermuster in Haskell allgemein beschreiben kann. Hierzu werden wir Typklassen intensiv verwenden.

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

3.11.1 Funktoren (Functors)

Wie wir in Kapitel 3.5 gesehen haben, können wir mittels Funktionen höherer Ordnung allgemeine Programmiermuster beschreiben und in verschiedenen Kontexten verwenden. Z.B. können wir mit der Funktion `map` sehr einfach jedes Element einer Liste bezüglich einer gegebenen Funktion transformieren:

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

So können wir z.B. mittels `map (2*)` alle Werte einer Zahlenliste verdoppeln.

Wir haben in den Übungen gesehen, dass wir dies auch ähnlich in anderen Strukturen machen können. Z.B. können wir in einem blattbeschrifteten Binärbaum der Form

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

durch folgende Funktion alle Blätter transformieren:

```
mapTree :: (a → b) → Tree a → Tree b
mapTree f (Leaf x)     = Leaf (f x)
mapTree f (Node tl tr) = Node (mapTree f tl) (mapTree f tr)
```

Wir können Ähnliches natürlich auch mit `Maybe`-Daten machen:

```
mapMaybe :: (a → b) → Maybe a → Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just x) = Just (f x)
```

Obwohl alle diese Funktionen von der Struktur sehr ähnlich sind, können wir diese nicht zu einem Schema verallgemeinern. Mittels Typklassen können wir allerdings ausdrücken, dass bestimmte Datentypen so eine Transformation bereitstellen:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Im Unterschied zu den Typklassen, die wir bisher kennengelernt haben, ist die Variable `f` in der Typklasse `Functor` kein Platzhalter für Typen, sondern für einstellige Typkonstruktoren, wie beispielsweise `Tree`, `Maybe` oder `[]`. Daher werden solche Typklassen auch *Typkonstruktorklassen* genannt. Somit hat also jeder Typkonstruktor f, der "`Functor` f" erfüllt, eine Transformation `fmap`.

Instanzen der Typklasse `Functor` können wie üblich definiert werden:

```
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree
```

```
instance Functor Maybe where
  fmap = mapMaybe
```

Durch die Typklasse `Functor` sind wir nun in der Lage, generelle Transformationen zu definieren, die auf verschiedene Datentypen anwendbar sind, wie z.B. das Inkrementieren aller Zahlen in einem Datentyp:

```
incAll :: Functor f => f Int  → f Int
incAll = fmap (+1)
```

Die Funktion `incAll` können wir auf alle `Functor`-Instanzen anwenden:

```
> incAll [1,2,3]
[2,3,4]

> incAll (Just 5)
Just 6

> incAll (Node (Leaf 1) (Leaf 2))
Node (Leaf 2) (Leaf 3)
```

Wir können `Functor`-Instanzen nicht nur für Container-artige Typkonstruktoren definieren, sondern auch für andere einstellige Typkonstruktoren, wie z.B. `IO`:

```
instance Functor IO where
  fmap f a = do x <- a
                return (f x)
```

Hierbei wird durch `fmap` eine IO-Aktion in eine andere IO-Aktion umgewandelt, die die übergebene Funktion auf das Ergebnis anwendet. Z.B. wird durch die folgende IO-Aktion eine Zeile eingelesen und deren Länge auf die Kommandozeile ausgegeben:

```
getLineLength = do x <- fmap length getLine
                   print x
```

Wir betrachten noch einen Beispielaufruf:

```
> getLineLength
abcde
5
```

Wenn wir `fmap` anwenden, woher wissen wir eigentlich, dass es sich wie eine sinnvolle Transformation verhält? Dies wird dadurch sichergestellt, dass jede Instanz von `Functor` die folgenden Gesetze erfüllen muss:

```
fmap id      = id                          -- Identity
fmap (f . g) = fmap f . fmap g             -- Composition
```

Da Haskell nicht überprüft, ob diese Gesetze erfüllt sind, obliegt es dem Entwickler einer `Functor`-Instanz sicherzustellen, dass dies der Fall ist. Z.B. erfüllt die folgende Instanz diese Gesetze nicht:

```
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = fmap f xs ++ [f x]
```

Zur Übung sollte man sich überlegen, warum die Gesetze hier nicht erfüllt sind, d.h. man sollte geeignete Gegenbeispiele finden.

3.11.2 Applicatives

Funktoren beschreiben das generelle Konzept, eine einstellige Funktion auf alle Elemente einer Struktur anzuwenden. Nehmen wir nun an, wir möchten Funktionen beliebiger Stelligkeit auf entsprechend vielen Strukturen anwenden. Hierzu könnte man für jede Stelligkeit eine Funktion angeben:

```
fmap0 :: a → f a

fmap1 :: (a → b) → f a → f b

fmap2 :: (a → b → c) → f a → f b → f c

fmap3 :: (a → b → c → d) → f a → f b → f c → f d
...
```

Hierbei entspricht `fmap1` der bisherigen Funktion `fmap` und `fmap0` ist der degenerierte Fall, wenn die anzuwendende Funktion kein Argument hat.

Damit könnten wir z.B. die Addition auf zwei `Maybe`-Werte anwenden:

```
> fmap2 (+) (Just 1) (Just 2)
Just 3
```

Dies ist allerdings etwas unbefriedigend, da wir viele ähnliche Funktionen angeben müssen und es auch nicht klar ist, wieviele dann ausreichend sind.

Auf der Ebene von Funktionen in Haskell ist die einfache Applikation ausreichend, da wir mehrere Argumente durch Curryfizierung darauf zurückführen können. Z.B. ist

```
apply :: (a → b) → a → b
apply f x = f x
```

die einfache Applikation einer Funktion `f` auf ein Argument `x` und wir erhalten bei der Funktion

```
add x y = x + y
```

die Anwendung mehrerer Argumente durch Rückführung auf `apply`:

```
> (add `apply` 1) `apply` 2
3
```

Leider funktioniert dies auf der `Functor` Ebene nicht so einfach. Wenn wir allerdings die anzuwendenden Funktionen auch in die `Functor`-Struktur einbetten, dann können wir dies erreichen. Nehmen wir dazu an, dass es die folgenden Basisfunktionen gibt:

```
pure :: a → f a

(<*>) :: f (a → b) → f a → f b
```

`pure` bettet also einen einzelnen Wert in eine `f`-Struktur ein und `<*>` ist die Applikation auf der `f`-Struktur.

Damit können wir nun die mehrstelligen Varianten von `fmap` auf diese Basisfunktionen zurückführen. Z.B. können wir

```
g <*> x <*> y
```

interpretieren als

```
(g <*> x) <*> y
```

wenn der Operator `<*>` linksassoziativ ist. Man beachte hierbei die Ähnlichkeit zum obigen `apply`!

Im Allgemeinen können wir also eine n-stellige Funktion g wie folgt auf n Strukturen anwenden:

```
pure g <*> x1 <*> x2 <*> ··· <*> xn
```

Damit könnten wir die obigen `fmap`-Funktionen alleine durch `pure` und `<*>` definieren:

```
fmap0 :: a → f a
fmap0 = pure

fmap1 :: (a → b) → f a → f b
fmap1 g x = pure g <*> x

fmap2 :: (a → b → c) → f a → f b → f c
fmap2 g x y = pure g <*> x <*> y

fmap3 :: (a → b → c → d) → f a → f b → f c → f d
fmap3 g x y z = pure g <*> x <*> y <*> z
...
```

Somit ist es nun nicht mehr nötig, eine bestimmte Anzahl von `fmap`-Funktionen vorzuhalten, sondern wir können jede Applikation mit Hilfe der Kombinatoren `pure` und `<*>` selbst definieren. Dies nennt man auch einen *applikativen Stil*. Strukturen, die dieses Verhalten bereitstellen, werden in der Typklasse `Applicative` [12] zusammengefasst:

```
class Functor f => Applicative f where
  pure  :: a  → f a
  (<*>) :: f (a  → b)  → f a  → f b
```

Z.B. gibt es für den Typkonstruktor `Maybe` eine Instanz von `Applicative`:

```
instance Applicative Maybe where
  pure                  = Just

  Nothing  <*> _        = Nothing
  (Just f) <*> Nothing  = Nothing
  (Just f) <*> (Just x) = Just (f x)
```

Da `Maybe` auch eine Instanz von `Functor` ist, könnten wir die Definition von "`<*>`" in der `Applicative`-Instanz auch noch kürzer schreiben:

```
instance Applicative Maybe where
  pure                  = Just

  Nothing  <*> _  = Nothing
  (Just f) <*> mx = fmap f mx
```

Somit können wir also beliebige Funktionen auf mehreren `Maybe`-Werten anwenden:

```
> pure (+1) <*> Just 1
Just 2

> pure (+) <*> Just 1 <*> Just 4
Just 5

> pure (+) <*> Just 1 <*> Nothing
Nothing
```

Den applikativen Stil kann man beispielsweise gut einsetzen, wenn man mit Fehlerwerten rechnen möchte. Dazu betrachten wir als Beispiel die folgende Darstellung arithmetischer Ausdrücke in Haskell:

```
data Exp = Num Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
 deriving Show
```

Z.B. könnte der Ausdruck `3 + (4 * 2)` wie folgt dargestellt werden:

```
Add (Num 3) (Mul (Num 4) (Num 2))
```

Wir wollen nun eine Funktion zum Auswerten solcher Ausdrücke definieren, also eine

Funktion

```
eval :: Exp → Int
```

Das Problem ist allerdings, dass manche Ausdrücke keinen Wert haben, wenn dort z.B. durch `0` diviert wird, wie bei dem Ausdruck `3 + (4 / (2 - 2))` repräsentiert durch

```
Add (Num 3) (Div (Num 4) (Sub (Num 2) (Num 2)))
```

Damit unser Auswerter nicht mit einem Laufzeitfehler abstürzt, sondern immer einen sinnvollen Wert zurückliefert, sollte `eval` den Typ

```
eval :: Exp → Maybe Int
```

haben. Somit werden `Maybe`-Werte zur Darstellung von Werten verwendet, welche auch fehlerhaft sein könnten. Wenn wir nun versuchen, die `eval`-Funktion zu definieren, müssten wir bei jedem Teilterm prüfen, ob dieser eventuell den Wert `Nothing` hat, um dann `Nothing` als Ergebnis zu liefern, und ansonsten die `Just`-Werte mit dem entsprechenden Operator zu verknüpfen. Z.B. würden die Regeln für die Addition wie folgt aussehen:

```
eval (Add e1 e2) = madd (eval e1) (eval e2)
 where
  madd Nothing  _        = Nothing
  madd (Just _) Nothing  = Nothing
  madd (Just m) (Just n) = Just (m + n)
```

Wie man sieht, ist der resultierende Code für `eval` recht umständlich und unleserlich, obwohl die Hilfsfunktion `madd` eigentlich nichts anderes macht, als etwaige Fehler, die bei der Auswertung der Argumente auftreten, zu propagieren, bzw. anderfalls die Werte dieser Argumente zu addieren. Mittels der `Applicative`-Kombinatoren können wir die Funktion `eval` wesentlich eleganter und lesbarer aufschreiben:

```
eval :: Exp → Maybe Int
eval (Num     n) = Just n
eval (Add e1 e2) = pure (+) <*> eval e1 <*> eval e2
eval (Sub e1 e2) = pure (-) <*> eval e1 <*> eval e2
eval (Mul e1 e2) = pure (*) <*> eval e1 <*> eval e2
eval (Div e1 e2) = mdiv (eval e1) (eval e2)
 where
  mdiv (Just x) (Just y) = if y == 0 then Nothing
                                     else Just (x `div` y)
  mdiv (Just _) Nothing  = Nothing
  mdiv Nothing  _        = Nothing
```

Wir können also unsere Standardfunktionen, wie "+", einfach durch Einbettung in die `Maybe`-Struktur, anwenden. Statt uns selbst um die Propagation von Fehlern während der Auswertung kümmern zu müssen, nutzen wir aus, dass der `<*>`-Kombinator auf `Maybe`-Strukturen eine Funktion nur anwendet, wenn sein Argument ein `Just`-Wert ist.

Andernfalls liefert er `Nothing` zurück. Somit müssen wir lediglich bei der Division noch explizit auf den `Maybe`-Werten arbeiten. In allen anderen Fällen können wir dank der `Applicative`-Kombinatoren davon abstrahieren.
Damit können wir nun Ausdrücke ohne Laufzeitfehler auswerten:

```
> eval (Add (Num 3) (Mul (Num 4) (Num 2)))
Just 11

> eval (Add (Num 3) (Div (Num 4) (Sub (Num 2) (Num 2))))
Nothing
```

Übrigens gibt es auch einen vordefinierten Operator `<$>`, der wie folgt definiert ist:

```
(<$>) :: Applicative f => (a → b) → f a → f b
g <$> x = pure g <*> x
```

Damit können wir unseren Auswerter noch einfacher schreiben (wir ersetzen in der ersten Regel auch noch `Just` durch `pure`, damit der Code möglichst allgemein wird):

```
eval :: Exp → Maybe Int
eval (Num     n) = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Sub e1 e2) = (-) <$> eval e1 <*> eval e2
eval (Mul e1 e2) = (*) <$> eval e1 <*> eval e2
eval (Div e1 e2) = mdiv (eval e1) (eval e2)
 where
  mdiv (Just x) (Just y) = if y == 0 then Nothing
                                     else Just (x `div` y)
  mdiv (Just _) Nothing  = Nothing
  mdiv Nothing  _        = Nothing
```

Wie `Functor`-Instanzen müssen auch Instanzen von `Applicative` bestimmte Gesetze erfüllen, die aber etwas komplexer sind:

```
pure id <*> v      = v                            -- Identity
pure f  <*> pure x = pure (f x)                   -- Homomorphism
u       <*> pure y = pure (\g → g y) <*> u        -- Interchange
(pure (.) <*> u <*> v) <*> w = u <*> (v <*> w)    -- Composition
```

Wie wir gesehen haben, ist `Applicative` eine Erweiterung von `Functor`. Wie sieht nun die Listeninstanz für `Applicative` aus? Diese ist wie folgt definiert:

```
instance Applicative [] where
  -- pure :: a → [a]
  pure x = [x]

  -- (<*>) :: [a → b] → [a] → [b]
  gs <*> xs = [ g x | g <- gs, x <- xs ]
```

Somit macht `pure` aus einem Wert eine einelementige Liste mit genau diesem Wert und `<*>` wendet alle Funktionen aus der im ersten Argument gegebenen Liste auf alle Werte des zweiten Argumentes an.

Hier sind einige Beispiele:

```
> pure (+1) <*> [1,2,3]
[2,3,4]

> pure (*) <*> [2] <*> [3]
[6]

> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

Was ist nun eine geeignete Intuition, um diese Beispiele zu verstehen? Hierzu sollte man den Typ `[a]` als Verallgemeinerung von `Maybe a` auffassen. Ein Wert aus `Maybe a` ist entweder leer (z.B. ein Fehler) oder enthält einen Wert. Dagegen kann ein Wert aus `[a]` leer sein, einen Wert oder viele Werte enthalten. Man kann sich also `[a]` als Ergebnis von Berechnungen vorstellen, die nichtdeterministisch Werte zurückliefern (nichtdeterministische Berechnungen werden wir insbesondere noch im Rahmen der Logikprogrammierung genauer erläutern). Damit können wir das letzte Beispiel so auffassen, dass alle Ergebnisse der ersten Berechnung (`[1,2]`) mit den Ergebnissen der zweiten Berechnung (`[3,4]`) kombiniert werden.

Berechnungen können z.B. auch I/O-Aktionen sein. Tatsächlich können wir auch einfach eine `Applicative`-Instanz für den Typkonstruktor `IO` angeben:

```
instance Applicative IO where
  -- pure :: a → IO a
  pure = return

  -- (<*>) :: IO (a → b) → IO a → IO b
  mg <*> mx = do g <- mg
                 x <- mx
                 return (g x)
```

Eigentlich haben wir `Applicative` als Erweiterung von `Functor` eingeführt, um Funktionen mit beliebig vielen Argumenten auf entsprechende Strukturen anzuwenden. Diese Interpretation passt nur bedingt auf die `IO`-Instanz. Es gibt aber noch eine weitere Interpretation von `Applicative`, die hierzu gut passt und für ein besseres Verständnis nützlich ist.

Die I/O-Monade wurde eingeführt, um rein funktionale Berechnungen von (Seiten-)Effektbehafteten Berechnungen zu unterscheiden. Eine *effektvolle Berechnung* (oder auch effektbehaftete Berechnung) berechnet einen Wert und hat nebenbei noch einen Effekt: im Falle von `eval` war der Effekt ein möglicher Fehler. Somit beschreibt der Ausdruck

```
pure f <*> e1 <*> e2
```

dass die Werte und Effekte von `e1` und `e2` berechnet werden und dann durch `f` kombiniert werden. Somit dient `Applicative` dazu, effektvolle Berechnungen elegant zu kombinieren, wohingegen man ohne `Applicative` die Teileffekte jeweils abfragen müsste, um dann die Effekte weiterzuleiten und die Ergebnisse zu kombinieren (vgl. die erste Variante von `eval`).

Im nächsten Kapitel werden wir eine weitere wichtige Strukturierung effektvoller Berechnungen kennenlernen.

3.11.3 Monaden (Monads)

Wir haben gesehen, wie wir mit `Applicative` effektvolle Berechnungen strukturieren können und dies an einem Evaluator für arithmetische Ausdrücke demonstriert. Mittels des Operators `<*>` können wir Funktionen auf effektvolle Berechnungen anwenden, allerdings müssen diese Funktionen selbst Funktionen ohne Effekte sein, wie aus dem Typ

```
(<*>) :: Applicative f => f (a → b) → f a → f b
```

ersichtlich ist. Dies hatte zur Folge, dass die Definitionen der Division

```
eval (Div e1 e2) = mdiv (eval e1) (eval e2)
 where
  mdiv (Just x) (Just y) = if y == 0 then Nothing
                                     else Just (x `div` y)
  mdiv (Just _) Nothing  = Nothing
  mdiv Nothing  _        = Nothing
```

etwas umständlich ist, weil hier die Funktion `mdiv` selbst die Effekte der Argumente überprüfen muss. Schöner wäre es, wenn man eine separate sichere Divisionsfunktion definieren könnte, die auf Zahlen arbeitet aber einen Fehlereffekt zurückliefert:

```
safediv :: Int → Int → Maybe Int
safediv x y | y == 0    = Nothing
            | otherwise = Just (x `div` y)
```

Damit könnten wir die Definition der Division etwas vereinfachen:

```
eval (Div e1 e2) = mdiv (eval e1) (eval e2)
 where
  mdiv (Just x) (Just y) = safediv x y
  mdiv (Just _) Nothing  = Nothing
  mdiv Nothing  _        = Nothing
```

Trotzdem ist es noch unbefriedigend, dass wir hier den Effekt der Argumente prüfen müssen und in den anderen Fällen nicht. Die Verbesserung beruht auf der Idee, eine Funktion zu definieren, die genau diese Überprüfung macht, d.h. die das Ergebnis einer effektbehafteten Berechnung überprüft und entsprechend weitermacht. Da wir im obigen Beispiel zwei Argumente haben, müsste so eine Funktion zweimal angewendet werden.

Z.B. könnte die Funktion wie folgt aussehen:

```
processMaybe :: Maybe a  → (a  → Maybe b)  → Maybe b
processMaybe Nothing  fm = Nothing
processMaybe (Just x) fm = fm x
```

Die Funktion rechnet also das erste Argument aus und macht mit dem zweiten Argument weiter, wenn das erste Argument nicht `Nothing` ist. Damit könnten wir die Division wie folgt definieren:

```
eval (Div e1 e2) = processMaybe (eval e1)
                                (\x1  → processMaybe (eval e2)
                                                     (\x2  → safediv x1 x2))
```

Hierdurch wird nun die Berechnungsstruktur etwas klarer: werte erst `e1` aus, danach `e2`, und dann verknüpfe beide Ergebnisse.

Interessanterweise hat `processMaybe` einen ähnlichen Typ wie der Bind-Operator der I/O-Monade:

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

Dieser diente ja auch dazu, eine Sequenz von Berechnungen zu strukturieren. Tatsächlich kann man den Bind-Operator verallgemeinern für andere Strukturen, die man dann *Monaden* nennt [18]. Eine Monade kann als Erweiterung der `Applicative`-Struktur angesehen werden, die zusätzlich noch einen Bind-Operator für sequenzielle Berechnungen besitzt. In Haskell ist daher allgemein die Klasse `Monad` wie folgt definiert [18]:

```
class Applicative m => Monad m where
  return :: a  → m a

  (>>=) :: m a  → (a  → m b)  → m b

  return = pure

  (>>) :: m a  → m b  → m b
  p >> q = p >>= \_  → q
```

Monaden haben also immer die Basisoperationen `return` und "`>>=`", die wir schon aus der I/O-Monade kennen. Zusätzlich ist "`>>`" eine abgeleitete Operation. Eine Monade stellt also mit dem "bind"-Operator "`>>=`" eine sequenzielle Verknüpfung von Effekten bereit. Der Vorteil der Benutzung von Monaden in Haskell ist, dass wir die gut lesbare `do`-Notation für alle Monaden anwenden können.

Da wir schon gesehen haben, wie eine `Applicative`-Instanz für `Maybe` aussieht, können wir nun auch eine `Monad`-Instanz für `Maybe` recht einfach definieren (dies ist die oben eingeführte Funktion `processMaybe`):

```
instance Monad Maybe where
  mx >>= f = case mx of Nothing  → Nothing
```

```
                        Just x   → f x
```

Das erste Argument von "»=" ist hier eine Berechnung, die fehlschlagen kann. Wenn diese fehlschlägt (`Nothing`), ist die gesamte Berechnung fehlgeschlagen, ansonsten wird die Berechnung mit dem Ergebnis der ersten Berechnung fortgeführt.

Die `Maybe`-Monade können wir nun verwenden, um den Fall der Division zu vereinfachen, indem wir dem Bind-Operator die Überprüfung der Argumenteffekte überlassen:

```
eval (Div e1 e2) = eval e1 >>= \x  →
                   eval e2 >>= \y  →
                   safediv x y
```

Wenn wir nun noch die `do`-Notation verwenden, erhalten wir die folgende Definition:

```
eval (Div e1 e2) = do x <- eval e1
                      y <- eval e2
                      safediv x y
```

Ähnlich wie bei `Applicative` können wir auch eine Listeninstanz von `Monad` defineren:

```
instance Monad [] where
  -- (>>=) :: [a]  →  (a  →  [b])  →  [b]
  xs >>= f = [ y | x <- xs, y <- f x ]
```

Somit wird durch den Ausdruck `xs >>= f` die Funktion `f` auf alle Werte der Liste `xs` angewendet und diese Ergebnisse werden aufgesammelt. Anders ausgedrückt: die nichtdeterministischen Werte von `xs` werden mit allen nichtdeterministischen Resultaten von `f` kombiniert. Dies wird an folgendem Beispiel deutlich:

```
pairs :: [a]  →  [b]  →  [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)
```

Mit der oben beschriebenen `Monad`-Instanz ergibt sich:

```
> pairs [1,2,3] [4,5]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Es sollte noch erwähnt werden, dass auch `Monad`-Instanzen nicht beliebig definiert werden sollen, sondern so, dass die Instanzen analog zu `Functor` und `Applicative` auch bestimmte Gesetze erfüllen:

```
return x >>= f           = f x                -- Left identity
m >>= return             = m                  -- Right identity
m >>= (\x  →  f x >>= g) = (m >>= f) >>= g    -- Associativity
```

Außerdem sollen die jeweiligen Instanzen von `Monad` und `Applicative` zusammenpassen, was durch folgende Forderung ausgedrückt wird:

```
pure      = return
m1 <*> m2 = m1 >>= (\x1 → m2 >>= (\x2 → return (x1 x2)))
```

Der monadische Programmierstil wird typischerweise zur Sequenzialisierung effektbehafteter Berechnungen benutzt oder anders formuliert: Er kommt zum Einsatz, wenn es wichtig ist, in welcher Reihenfolge diese Effekte passieren. Die Benutzung von `Monad`-Instanzen für verschiedene Datentypen erlaubt es, verschiedene Effekte voneinander zu trennen und so mehr Programmiersicherheit zu erreichen. Daher werden Monaden in vielen Anwendungen verwendet, wie z.B. auch beim automatisierten Testen, was wir uns als Nächstes anschauen. Darüberhinaus gibt es noch weitere Typkonstruktorklassen, die verschiedene Aspekte von Datentypen abstrahieren.[15]

3.12 Automatisiertes Testen

Tests sind ein wichtiges Hilfsmittel um Programmierfehler aufzudecken. Sie können zwar niemals die Abwesenheit von Fehlern zeigen (dazu braucht man Beweise), zeigen jedoch häufig deren Anwesenheit und sind oft einfacher zu erstellen als Korrektheitsbeweise. Diese QuickSort-Implementierung

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

können wir manuell im GHCi testen:

```
ghci> qsort [9,1,4]
[1,4,9]
```

Alternativ könnten wir eine Reihe solcher Tests in eine Datei schreiben und diese jedesmal ausführen, wenn wir die Implementierung ändern (Regressionstests).

3.12.1 Eigenschaftsbasiertes Testen

Regressionstests zu schreiben ist langweilig. Programmierer vernachlässigen diese Aufgabe deshalb oft und schreiben nur wenige Tests, die nicht alle interessanten Eigenschaften überprüfen. Besser wäre es, viele Tests mit möglichst vielen oder sogar beliebigen Testdaten durchzuführen. Aber wie sollen wir die Tests aufschreiben, wenn die Daten beliebig sind und wir sie nicht konkret kennen? Die Antwort ist ganz einfach: statt Funktionen auf bestimmten Daten zu testen, sollten wir besser allgemeine Eigenschaften von Funktionen überprüfen! Dies können spezielle Eigenschaften sein („eine Sortierfunktion liefert immer eine Permutation der Eingabeliste“) oder auch eine generelle Eigenschaft wie „die Implementierung A liefert die gleichen Ergebnisse wie die Implementierung B“ (Regressionstest).

[15] Ein Überblick findet sich unter `https://wiki.haskell.org/Typeclassopedia`.

In Haskell kann man sehr einfach Eigenschaften von Funktionen schreiben und diese mit automatisch generierten Eingaben testen. Dazu verwenden wir das Tool *QuickCheck* [6], das man mit

```
bash# cabal install QuickCheck
```

installieren kann.

Als erstes Beispiel betrachten wir das folgende Haskell-Prädikat, welches spezifiziert, dass `qsort` idempotent sein soll. Hierbei berücksichtigen wir die Konvention von QuickCheck, dass zu testende Eigenschaften immer mit `prop_` anfangen:

```
import Test.QuickCheck

prop_idempotence :: [Int]  → Bool
prop_idempotence xs = qsort (qsort xs) == qsort xs
```

Wir können dieses Prädikat von Hand mit Beispieleingaben aufrufen.

```
ghci> prop_idempotence [1,2,3]
True
```

Wir können aber auch die Funktion `quickCheck` verwenden, die zufällig Listen vom Typ `[Int]` generiert und prüft, ob das Prädikat für diese erfüllt ist:

```
ghci> quickCheck prop_idempotence
*** Failed! Falsifiable (after 4 tests and 4 shrinks):
[1,0,0]
ghci> quickCheck prop_idempotence
*** Failed! Falsifiable (after 6 tests and 10 shrinks):
[0,-1,-1]
```

Nachdem QuickCheck einige Tests durchgeführt hat, findet es ein Gegenbeispiel und versucht anschließend das Beispiel so zu vereinfachen (shrink), dass die Testdaten möglichst klein sind. Weil die Testeingaben zufällig generiert werden, erhalten wir auch unterschiedliche Gegenbeispiele.

Wir haben bei der Implementierung von `qsort` einen Fehler gemacht. Um diesen zu finden testen wir `qsort` für das von `quickCheck` gefundene Gegenbeispiel:

```
ghci> qsort [1,0,0]
[0,0,1]
ghci> qsort [0,0,1]
[0,1]
```

Im Ergebnis ist ein Wert verloren gegangen, weil beide Listenwerte identisch sind. Wir passen die Definition daher wie folgt an und schreiben `(>=x)` anstelle von `(>x)`:

```
qsort :: Ord a => [a]  → [a]
qsort []     = []
```

```
qsort (x:xs) = filter (<x) xs ++ x : filter (>=x) xs
```

Zumindest für die obige Eingabe funktioniert die Implementierung nun.

```
ghci> prop_idempotence [1,0,0]
True
```

Wir verwenden wieder `quickCheck` um weitere Fehler zu suchen.

```
ghci> quickCheck prop_idempotence
quickCheck prop_idempotence
*** Failed! Falsifiable (after 10 tests and 8 shrinks):
[0,-1,-2]
```

Noch immer enthält unsere Implementierung einen Fehler:

```
ghci> qsort [0,-1,-2]
[-1,-2,0]
```

Wir haben die rekursiven Aufrufe vergessen:

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

Diese Implementierung ist augenscheinlich idempotent. Zumindest findet QuickCheck kein Gegenbeispiel mehr.

```
ghci> quickCheck prop_idempotence
OK, passed 100 tests.
```

Idempotenz ist eine notwendige Eigenschaft einer Sortierfunktion, aber nicht hinreichend. Zum Beispiel wäre die Definition

```
qsort _ = []
```

auch idempotent.

Als weitere Eigenschaft spezifizieren wir daher, dass alle Elemente aus dem Argument von `qsort` auch im Ergebnis vorkommen müssen und umgekehrt.

```
prop_preservation :: [Int]  → Bool
prop_preservation xs =
  null (xs \\ qsort xs) && null (qsort xs \\ xs)
```

Wir verwenden dazu die Funktion `(\\)` zur Berechnung der Differenz zweier Listen aus dem Modul `Data.List`.

Auch diese Eigenschaft ist augenscheinlich erfüllt:

```
ghci> quickCheck prop_preservation
```

```
OK, passed 100 tests.
```

Jede Funktion, die eine Permutation ihrer Eingabe liefert, erfüllt die obige Eigenschaft. Wir könnten daher zusätzlich testen, ob das erste Element einer sortierten Liste das kleinste ist, um zu testen, ob die Funktion richtig herum sortiert.

```
prop_smallest_first :: [Int] → Bool
prop_smallest_first xs =
  head (qsort xs) == minimum xs
```

Die Funktion `minimum` berechnet das kleinste Element einer Liste und ist ebenfalls im Modul `Data.List` vordefiniert.

Wenn wir diese Eigenschaft mit `quickCheck` testen, erhalten wir einen Fehler.

```
ghci> quickCheck prop_smallest_first
*** Exception: Prelude.head: empty list
```

Diese Eigenschaft ist nur für nicht-leere Listen sinnvoll und liefert mit der leeren Liste als Eingabe einen Fehler beim Pattern-Matching.

QuickCheck stellt eine Funktion `(==>)` zur Spezifikation von Vorbedingungen bereit, die wir verwenden um die obige Eigenschaft anzupassen.

```
prop_smallest_first :: [Int] → Property
prop_smallest_first xs =
  not (null xs) ==>
    head (qsort xs) == minimum xs
```

Durch die Verwendung von `(==>)` ändert sich der Ergebnistyp der Eigenschaft von `Bool` zu `Property`. Wir können sie aber weiterhin mit `quickCheck` testen.

```
ghci> quickCheck prop_smallest_first
OK, passed 100 tests.
```

Häufig verwendet man statt einzelner Eigenschaften eine Referenz- oder Prototyp-Implementierung. Wenn man zum Beispiel eine offensichtlich korrekte, aber ineffiziente Variante einer Funktion programmieren kann, kann man diese benutzen um eine effiziente Implementierung dagegen zu testen. Beispielhaft testen wir unsere `qsort`-Funktion gegen die vordefinierte `sort`-Funktion aus dem `Data.List`-Modul.

```
prop_reference :: [Int] → Bool
prop_reference xs = qsort xs == sort xs
```

Für 100 von `quickCheck` generierte Eingaben vom Typ `[Int]` berechnet `qsort` dasselbe Ergebnis wie die vordefinierte Sortierfunktion, was zu einigem Vertrauen in die Implementierung von `qsort` berechtigt.

```
ghci> quickCheck prop_reference
OK, passed 100 tests.
```

3.12.2 Automatisierte Testausführung

Man kann mit QuickCheck auch sehr einfach alle Eigenschaften in einem Modul, die mit dem Präfix `prop_` beginnen, automatisch testen. Hierzu muss man im Modulkopf die Spracherweiterung

```
{-# LANGUAGE TemplateHaskell #-}
```

angeben und die Definitionen

```
return []
runTests = $quickCheckAll
```

am Ende des Programms hinzufügen (die merkwürdige erste Zeile hängt mit der Spracherweiterung zusammen). Danach kann man dann mit

```
ghci> runTests
```

alle Tests ablaufen lassen.

Dies können wir auch zum einfachen Testen von Funktionen verwenden, d.h. wenn wir keine Eigenschaften formulieren, sondern nur einfache Testfälle mit konkreten Testdaten ablaufen lassen wollen (dies wird in anderen Programmiersprachen traditionell unter dem Begriff „Unit Testing“ verstanden). Z.B. könnten wir den obigen einfachen Test (vgl. Beginn von Kapitel 3.12) als Eigenschaft ohne Parameter in unserem Programm aufschreiben:

```
prop_qsort914 = qsort [9,1,4] == [1,4,9]
```

Wenn wir nun wieder mit

```
ghci> runTests
```

alle Tests ablaufen lassen, wird auch dieser Test ausgeführt.

3.12.3 Klassifikation der Testeingabe

Es ist zwar nett, dass QuickCheck für uns die Testdaten erzeugt, aber es ist etwas unbefriedigend, dass wir nicht wissen, welche dies sind. Aus diesem Grund unterstützt QuickCheck auch Methoden, um Informationen über die ausgeführten Tests zu erhalten. Zum Beispiel gibt die Funktion `verboseCheck` alle Tests der Reihe nach aus, was allerdings meistens zu unübersichtlich ist. Deshalb gibt es auch Funktionen, mit denen man Tests klassifizieren kann.

Mit der Funktion `classify` kann man ein Prädikat angeben, um Tests zu klassifizieren. Darüberhinaus kann man einen String angeben, der die dazugehörigen Tests charakterisiert. Wenn wir also das Referenz-Prädikat so implementieren

```
prop_reference :: [Int]  → Property
```

```
prop_reference xs =
  classify (length xs < 5) "small" $
    qsort xs == sort xs
```

erzeugt `quickCheck` die folgende Ausgabe:

```
ghci> quickCheck prop_reference
+++ OK, passed 100 tests (15% small).
ghci> quickCheck prop_reference
+++ OK, passed 100 tests (24% small).
ghci> quickCheck prop_reference
+++ OK, passed 100 tests (16% small).
```

Da `classify p s` aus einer Eigenschaft eine neue Eigenschaft (mit Klassifikation) macht, können wir dies auch schachteln:

```
prop_reference :: [Int]  → Property
prop_reference xs =
  classify (length xs < 5) "small" $
  classify (length xs > 20) "big" $
    qsort xs == sort xs
```

Damit erhalten wir mehrere Klassifikationen der Testdaten:

```
ghci> quickCheck prop_reference
+++ OK, passed 100 tests:
44% big
24% small
```

Schließlich können wir auch die Längen selbst verwenden, um Tests noch feiner zu gruppieren:

```
prop_reference :: [Int]  → Property
prop_reference xs =
  collect (length xs) $
    qsort xs == sort xs
```

Danach gibt `quickCheck` aus wie viele Listen wie lang waren.

```
ghci> quickCheck prop_reference
OK, passed 100 tests.
+++ OK, passed 100 tests:
 5% 9
 5% 1
 5% 0
 4% 5
 4% 33
 4% 24
 4% 2
 4% 11
```

```
3% 7
3% 60
3% 17
2% 6
2% 53
2% 46
2% 44
2% 43
2% 4
2% 37
2% 3
2% 29
2% 27
2% 25
2% 20
2% 19
2% 16
2% 15
1% 83
1% 82
1% 8
1% 72
1% 66
1% 64
1% 57
1% 55
1% 54
1% 52
1% 51
1% 47
1% 45
1% 42
1% 36
1% 35
1% 32
1% 30
1% 28
1% 26
1% 22
1% 21
1% 18
1% 14
1% 12
1% 10
```

Es zeigt sich, dass durchaus auch sehr lange Listen mit über 80 Elementen generiert werden. Dies macht die Stärke der automatisierten Tests aus, denn solche Testfälle würde man selten von Hand aufschreiben.

3.12.4 Eingabe-Generatoren

Weil QuickCheck die Eingabedaten zufällig erzeugt, variieren diese in unterschiedlichen Läufen von QuickCheck. Wie werden aber diese Daten erzeugt? Zum Glück muss man als Benutzerin oder Benutzer wenig davon wissen, allerdings sind doch einige Details notwendig, wenn man Funktionen mit selbst definierten Daten testen will.

In QuickCheck legt die Klasse `Arbitrary` fest, wie für einen bestimmten Typ Testdaten erzeugt werden.

```
class Arbitrary a where
  arbitrary :: Gen a
```

Hierbei ist `Gen` eine ähnliche Struktur wie `IO` (eine Monade), die aber anstelle von Ein- und Ausgabeaktionen das Treffen von Zufallsentscheidungen erlaubt. Wichtig ist insbesondere, dass wir auch die `do`-Notation für `Gen` verwenden können.

Wir definieren einen Datentyp für kleine natürliche Zahlen, welchen wir verwenden wollen, um unsere Implementierung mit QuickSort zu testen.

```
data Digit = Digit Int
 deriving (Eq, Ord)
```

Damit QuickCheck Gegenbeispiele anzeigen kann, müssen wir eine `Show`-Instanz definieren. Anders als eine mittels `deriving` erzeugte Instanz lässt unsere aber den Konstruktor weg:

```
instance Show Digit where
  show (Digit d) = show d
```

Wir passen den Typ unserer Eigenschaft an, um nur noch Listen von kleinen natürlichen Zahlen zu sortieren.

```
prop_reference :: [Digit]  → Property
prop_reference xs =
  classify (length xs < 5) "small" $
  classify (length xs > 20) "big" $
    qsort xs == sort xs
```

Wenn wir nun versuchen, QuickCheck mit der angepassten Eigenschaft aufzurufen, erhalten wie einen Typfehler.

```
ghci> quickCheck prop_reference
No instance for (Arbitrary Digit) ...
```

QuickCheck weiß nicht, wie man Werte vom Typ `Digit` generiert. Wir müssen dies durch eine `Arbitrary`-Instanz festlegen.

```
instance Arbitrary Digit where
  arbitrary = do d <- elements [0..9]
```

```
                  return (Digit d)
```

Die Definition von `arbitrary` für den Typ `Digit` wählt mit der Funktion

```
elements :: [a]  → Gen a
```

eine kleine natürliche Zahl zufällig aus und gibt diese als `Digit`-Wert zurück. Alternativ kann man auch mit der Funktion

```
choose :: (a,a)  → Gen a
```

zufällig Werte aus einem Intervall wählen:

```
instance Arbitrary Digit where
  arbitrary = do d <- choose (0,9)
                 return (Digit d)
```

Als weitere Alternative kann man mit der Funktion

```
oneof :: [Gen a]  → Gen a
```

aus einer Liste von Generatoren zufällig Generatoren auswählen. Dies ist nützlich, wenn wir aus Generatoren für Datentypalternativen zufällig etwas auswählen wollen. Z.B. kann eine `Arbitrary`-Instanz für `Either` wie folgt definiert werden:

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (Either a b) where
  arbitrary = do x <- arbitrary
                 y <- arbitrary
                 oneof [return (Left x), return (Right y)]
```

Die vordefinierte `Arbitrary`-Instanz für Listen verwendet unsere Instanz für `Digit`, um Werte vom Typ `[Digit]` zu erzeugen. Sie ist wie folgt definiert:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized $ \n  →
   do k <- choose (0,n)
      makeArbitraries k
```

Die Funktion

```
sized :: (Int  → Gen a)  → Gen a
```

kann man verwenden, um einen Generator zu definieren, der Werte bis zu einer gewissen Größe erzeugt. In diesem Fall wird der Größen-Parameter `n` verwendet, um die Länge der erzeugten Liste zu beschränken. Mit `choose` wird eine Länge zwischen `0` und `n` gewählt und dann mit `makeArbitraries` eine zufällige Liste entsprechender Länge erzeugt. Die Elemente der Liste werden in `makeArbitraries` durch einen Aufruf der Funktion `arbitrary` für den Element-Typ generiert:

```
makeArbitraries :: Arbitrary a => Int  → Gen [a]
makeArbitraries n | n <= 0    = return []
                  | otherwise = do x  <- arbitrary
                                   xs <- makeArbitraries (n - 1)
                                   return (x:xs)
```

Diese Funktion ist übrigens auch bereits in QuickCheck vordefiniert, allerdings unter dem Namen `vector`. Mit diesen Funktionen (es gibt natürlich noch mehr spezielle Funktionen) kann man dann Generatoren für selbst definierte Datentypen sehr kompakt definieren.

Fallstudie: Arithmetik auf Peano-Zahlen

Als Beispiel betrachten wir einmal eine Darstellung natürlicher Zahlen in der sogenannten *Peano-Darstellung*. Dies ist eine induktive Definition der Menge $\mathbb{N}$ der natürlichen Zahlen mit den Konstruktoren `Z` (**Z**ero, Null) und `S` (**S**uccessor, Nachfolger). Mit dieser Darstellung können wir arithmetische Operationen wie Addition und Multiplikation einfach selbst definieren:

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

data Nat = Z | S Nat
 deriving (Eq,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)
```

Wir möchten einige Eigenschaften unserer selbst definierten Arithmetik testen. Zunächst einmal sollte die Addition kommutativ sein:

```
prop_add_comm :: Nat  → Nat  → Bool
prop_add_comm m n = add m n == add n m
```

Wenn wir versuchen, diese Eigenschaft zu testen, erhalten wir die Mitteilung, dass QuickCheck nicht in der Lage ist, Testdaten zu generieren:

```
ghci> quickCheck prop_add_comm
No instance for (Arbitrary Nat) ...
```

Wie können wir Testdaten generieren? Die einfachste Methode ist die Umwandlung von `Int`-Zahlen, wofür es Testdatengeneratoren gibt, in `Nat`-Zahlen mittels der Hilfsfunktion

```
toNat :: Int  → Nat
toNat n | n==0 = Z
        | n>0  = S (toNat (n-1))
```

Damit könnten wir die Eigenschaft auf `Int`-Zahlen umdefinieren:

```
prop_add_comm :: Int  → Int  → Bool
prop_add_comm x y =
  let m = toNat x
      n = toNat y
  in add m n == add n m
```

Wenn wir die Eigenschaft allerdings austesten, erhalten wir einen Fehler

```
ghci> quickCheck prop_add_comm
*** Failed! (after 5 tests and 4 shrinks):
Exception: Non-exhaustive patterns in function toNat
0
-1
```

weil wir negative Zahlen nicht in `Nat`-Zahlen konvertieren können. Daher schränken wir die Generierung auf positive Zahlen ein:

```
prop_add_comm :: Int  → Int  → Property
prop_add_comm x y = x>=0 && y>=0 ==>
  let m = toNat x
      n = toNat y
  in add m n == add n m
```

Damit funktioniert nun unser Test erfolgreich:

```
ghci> quickCheck prop_add_comm
+++ OK, passed 100 tests.
```

Unschön ist allerdings, dass wir nun bei jedem Test diese Datenkonvertierung explizit einbauen müssen. Besser ist es, eine `Arbitrary`-Instanz für `Nat` zu definieren. Dies können wir durch Ausnutzung der `Arbitrary`-Instanz für `Int` und Konvertierung wie folgt machen:

```
instance Arbitrary Nat where
  arbitrary = do x <- arbitrary `suchThat` (>=0)
                 return (toNat x)
```

Die Funktion

```
suchThat :: Gen a  → (a  → Bool)  → Gen a
```

schränkt dabei die generierten Daten auf ein Prädikat ein.

Eine andere Möglichkeit ist die direkte Definition des Generators ohne Benutzung von `toNat`. Hierbei gehen wir ähnlich wie in der `Arbitrary`-Instanz für `Either` vor:

```
instance Arbitrary Nat where
  arbitrary = do s <- arbitrary
```

```
                  oneof [return Z, return (S s)]
```

Da `oneof` recht gleichmäßig zwischen den Alternativen wählt, erhalten wir überwiegend kleine Zahlen, wie man z.B. mit `verboseCheck` sehen kann. Als Verbesserung kann man die Funktion

```
frequency :: [(Int, Gen a)]  → Gen a
```

an Stelle von `oneof` verwenden, bei der man den einzelnen Generatoren noch eine gewichtete Häufigkeitsverteilung mitgeben kann:

```
instance Arbitrary Nat where
  arbitrary = do s <- arbitrary
                 frequency [(1,return Z), (4,return (S s))]
```

Damit können wir nun bequem verschiedene Eigenschaften testen:

```
-- Kommutativitaet von add:
prop_add_comm :: Nat  → Nat  → Bool
prop_add_comm m n = add m n == add n m

-- Assoziativitaet von add:
prop_add_assoc :: Nat  → Nat  → Nat  → Bool
prop_add_assoc x y z = add (add x y) z == add x (add y z)

-- Kommutativitaet von mul:
prop_mul_comm :: Nat  → Nat  → Bool
prop_mul_comm m n = mul m n == mul n m

-- Assoziativitaet von mul:
prop_mul_assoc :: Nat  → Nat  → Nat  → Bool
prop_mul_assoc x y z = mul (mul x y) z == mul x (mul y z)

-- Distributivitaet von mul und add:
prop_mul_distr :: Nat  → Nat  → Nat  → Bool
prop_mul_distr x y z = mul (add x y) z == add (mul x z) (mul y z)
```

Als kleine Anmerkung sei noch erwähnt, dass diese Gesetze eigentlich für alle Zahlen gelten sollten, aber bei Gleitkommazahlen auf Grund von Rundungsfehlern nicht immer erfüllt sind. Tatsächlich kann man dies mit QuickCheck herausfinden. Z.B. wird die Eigenschaft

```
prop_int_assoc :: Int  → Int  → Int  → Bool
prop_int_assoc x y z = (x*y)*z == x*(y*z)
```

erfolgreich getestet, während die gleiche Eigenschaft auf Gleitkommazahlen

```
prop_float_assoc :: Float  → Float  → Float  → Bool
prop_float_assoc x y z = (x*y)*z == x*(y*z)
```

einen Fehler aufzeigt:

```
ghci> quickCheck prop_float_assoc
*** Failed! Falsifiable (after 3 tests and 184 shrinks):
1.0e-45
0.2537598
1.9703689
```

Wenn man ohne Rundungsfehler mit belieber Genaugkeit rechnen möchte, kann man in Haskell auch den Datentyp `Rational` (rationale Zahlen) benutzen.

3.13 Datenabstraktion und abstrakte Datentypen

Wie wir am Beispiel der ganzen Zahlen gesehen haben, gibt es verschiedene Möglichkeiten, einen Datentyp zu implementieren. Tatsächlich ist eine zentrale Idee der Programmierung mit Daten die

> **Datenabstraktion**: Es ist nicht wichtig, wie die Daten intern dargestellt sind, sondern es ist nur wichtig, wie man die Daten benutzt, d.h. welche Operationen hierfür zur Verfügung stehen.

Z.B. ist es in höheren Programmiersprachen uninteressant, in welcher Bitreihenfolge Integer-Werte intern repräsentiert werden. Wichtig ist nur, dass die Addition und Multiplikation „wie üblich“ funktioniert. Dasselbe gilt auch für andere Daten. Z.B. haben wir in Kapitel 3.5.5 gesehen, dass wir auch Felder anders implementieren können, als man dies aus imperativen Programmiersprachen kennt. Trotzdem liefern die Feldoperationen die richtigen Ergebnisse. Da die Datenabstraktion eine wichtige Programmertechnik (in allen Programmiersprachen!) ist, wollen wir dies nachfolgend etwas genauer erläutern.

Damit man von der internen Struktur oder Darstellung der Daten abstrahieren kann, muss ein Anwender der Daten diese nicht kennen. Stattdessen muss man nur die Schnittstelle, manchmal auch API (Application Programming Interface) genannt, kennen, d.h. eine Menge von Operationen, die man verwenden kann, um mit den Daten zu arbeiten. Um der Schnittstelle etwas mehr Struktur zu geben, klassifiziert man die Schnittstellenoperationen wie folgt:

Konstruktoren: Operationen zur Konstruktion von Daten.

Selektoren: Operationen zur Extraktion von Teilinformationen aus den Daten.

Operatoren: Operationen zur Verknüpfung von Daten.

Wir betrachten als einfaches Beispiel rationale Zahlen, die aus einem Zähler (numerator) und einem Nenner (denominator) bestehen. Ein konkreter Datentyp ist hierfür sofort definiert:

```
data Rat = Rat Int Int
  deriving Eq
```

Für eine „natürliche“ Anzeige definieren wir eine eigene Show-Instanz:

```
instance Show Rat where
  show (Rat n d) = show n ++ "/" ++ show d
```

Ein *Konstruktor* für rationale Zahlen ist `Rat`. Wenn wir allerdings die eigentliche Implementierung verstecken wollen, damit wir sie später verändern können, dann ist es besser, an Stelle dieses Konstruktors eine selbst definierte Konstruktoroperation zu verwenden:

```
rat :: Int  → Int  → Rat
rat n d = Rat n d
```

Darüberhinaus benötigen wir auch *Selektor*operationen, wenn wir auf die Komponenten einer rationalen Zahl zugreifen wollen:

```
numerator :: Rat  → Int
numerator (Rat n _) = n

denominator :: Rat  → Int
denominator (Rat _ n) = n
```

Ein *Operator* auf rationalen Zahlen ist z.B. eine arithmetische Verknüpfung wie Addition oder Multiplikation. Diese sind mit dem Schulwissen zur Bruchrechnung recht einfach definierbar:

```
addR :: Rat  → Rat  → Rat
addR (Rat n1 d1) (Rat n2 d2) = rat (n1*d2 + n2*d1) (d1*d2)

mulR :: Rat  → Rat  → Rat
mulR (Rat n1 d1) (Rat n2 d2) = rat (n1*n2) (d1*d2)
```

Damit der Anwender nicht in die Details der Implementierung schaut und so Funktionen definieren könnte, die genau von dieser Implementierung abhängig sind, verbergen wir diese, indem nur der Typname `Rat` exportiert wird. Unser Modul `Rat` sieht also wie folgt aus:

```
module Rat(Rat,rat,numerator,denominator,addR,mulR) where

data Rat = Rat Int Int
  deriving Eq

instance Show Rat where
  show (Rat n d) = show n ++ "/" ++ show d

rat :: Int  → Int  → Rat
rat n d = Rat n d

numerator :: Rat  → Int
numerator (Rat n _) = n

denominator :: Rat  → Int
```

```
denominator (Rat _ n) = n

addR :: Rat  → Rat  → Rat
addR (Rat n1 d1) (Rat n2 d2) = rat (n1*d2 + n2*d1) (d1*d2)

mulR :: Rat  → Rat  → Rat
mulR (Rat n1 d1) (Rat n2 d2) = rat (n1*n2) (d1*d2)
```

Wir können nun rationale Zahlen verwenden. Z.B. können wir Brüche definieren und diese verknüpfen:

```
oneThird :: Rat
oneThird =  rat 1 3

twoThird :: Rat
twoThird = addR oneThird oneThird
```

Wenn wir den Wert von `twoThird` ausgeben, erhalten wir

```
ghci> twoThird
6/9
```

Dies ist zwar nicht falsch, aber eigentlich hätten wir die Ausgabe `2/3` erwartet. Intuitiv sollen rationale Zahlen immer gekürzt werden. Wir erreichen dies durch eine einfache Änderung: bei der Konstruktion rationaler Zahlen werden Zähler und Nenner immer mit dem größten gemeinsamen Teiler gekürzt. Wir ändern daher einfach die Definition des Konstruktors:

```
rat :: Int  → Int  → Rat
rat n d = let g = gcd n d in Rat (div n g) (div d g)

ghci> twoThird
2/3
```

Ein Nachteil verbleibt allerdings noch: bei negativen Nennern ist die Darstellung nicht eindeutig. Z.B. ergibt

```
ghci> rat (-3) (-2)
-3/-2
```

obwohl dies der gleiche Wert wie `3/2` ist. Damit die Darstellung eindeutig und damit möglichst einfach wird, vermeiden wir die Konstruktion rationaler Zahlen mit negativen Nennern durch folgende Neudefinition:

```
rat :: Int  → Int  → Rat
rat n d = let g  = gcd n d in posDenom (div n g) (div d g)
 where
  posDenom n d = if d<0 then Rat (0 - n) (abs d)
                        else Rat n d
```

```
ghci> rat (-3) (-2)
3/2
```

Hier sehen wir schon einen wichtigen Vorteil der Datenabstraktion. Wir haben nur eine Operation verbessert und können die restlichen Operationen mit der gleichen Schnittstelle weiter verwenden.

Wenn aber die Implementierung eines Datentyps verborgen ist, woher wissen wir dann, wie sich die Operationen verhalten oder was für Ergebnisse diese liefern? Hier kommt ein weiteres Prinzip der Datenabstraktion ins Spiel: Das Verhalten der Operationen wird durch bestimmte Gesetze beschrieben, die jede Implementierung erfüllen muss. Somit können wir für einen Datentyp unterschiedliche Implementierungen angeben, aber uns trotzdem darauf verlassen, dass diese in einer bestimmten Art und Weise funktionieren. Im Prinzip könnten diese Gesetze beliebige logische Formeln sein, aber in der Regel schränkt man sich auf Gleichungen wie "`addR x y == addR y x`" ein. Eine solche Beschreibung eines Datentyps nennt man dann auch *abstrakten Datentyp*:

Definition 3.12 (Abstrakter Datentyp) *Ein* abstrakter Datentyp *(*ADT*) ist ein Tupel* (Σ, X, E)*, das aus den folgenden Komponenten besteht:*

- *eine Programmsignatur* Σ *(diese enthält die Operationen des Datentyps),*
- *eine Menge* X *von Variablen, die disjunkt zu* Σ *sind, und*
- *eine Menge* E *von Gleichungen der Form* $t = t'$*, wobei* $t, t' \in T_\Sigma(X)_s$ *Terme der gleichen Sorte* s *sind.*

Betrachten wir als Beispiel noch einmal die rationalen Zahlen. Der ADT für rationale Zahlen ist[16] (Σ, X, E) mit

- $\Sigma = (S, F)$ mit
  $S = \{\texttt{Rat}, \texttt{Int}\}$
  $F = \{\texttt{rat} :: \texttt{Int Int} \to \texttt{Rat}, \texttt{numerator} :: \texttt{Rat} \to \texttt{Int}, \texttt{denominator} :: \texttt{Rat} \to \texttt{Int}\}$
- $X = \{n :: \texttt{Int}, d :: \texttt{Int}\}$
- $E = \left\{ \dfrac{\texttt{numerator}\ \ (\texttt{rat}\ n\ d)}{\texttt{denominator}\ (\texttt{rat}\ n\ d)} = \dfrac{n}{d} \right\}$

Typischerweise besitzt ein ADT eine ausgezeichnete Signatur (hier: `Rat`), die die Elemente des Datentyps beschreibt (und weitere Signaturen, die in den ADT-Operationen benutzt werden). Häufig kann man aus der Benutzung dieser ADT-Signatur auch erkennen, zu welcher Kategorie die ADT-Operationen gehören:

- *Konstruktoren* haben die ADT-Signatur als Ergebnissignatur (hier: `Rat`).
- *Selektoren* haben die ADT-Signatur als Argumentsignatur (hier: `numerator` und `denominator`).

[16] Eigentlich gehört zu diesem ADT auch der ADT für ganze Zahlen, da wir diesen ja benutzen. Im Folgenden lassen wir aber ADTs, die schon bekannt sind und die wir nur verwenden, weg. Formal müsste man diese aber importieren.

- *Operatoren* haben die ADT-Signatur als Argument- und Ergebnissignatur (z.B. "`addR` :: `Rat`, `Rat` → `Rat`", die hier aber nicht weiter spezifiziert ist).

Häufig ist es so, dass Selektoren und Konstruktoren in einer eindeutigen Beziehung stehen, d.h. dass der Selektor genau die Daten zurückliefert, die man mit einem Konstruktor konstruiert hat. In unserem Fall wären allerdings die Gleichungen

$$\texttt{numerator}\ (\texttt{rat}\ n\ d) = n$$

und

$$\texttt{denominator}\ (\texttt{rat}\ n\ d) = d$$

nicht korrekt, da wir die Brüche in gekürzter Form darstellen. Die Gleichung im ADT spezifiziert daher, dass es unwesentlich ist, wie rationale Zahlen genau dargestellt werden. Wichtig ist nur, dass das Verhältnis von Zähler und Nenner immer gleich ist.

Eine *Implementierung eines ADT* ist ein Programm, das alle Funktionen in der Programmsignatur so implementiert, dass die Gleichungen für alle Werte an Stelle der Variablen immer erfüllt sind. Mit QuickCheck können wir zumindest automatische Korrektheitstests durchführen. Z.B. können wir die ADT-Gleichung und weitere Gleichungen für Eigenschaften arithmetischer Funktionen wie folgt überprüfen:

```
-- ADT-Gleichung:
prop_adt_eq :: Int  → Int  → Property
prop_adt_eq n d =
  d/=0 ==> numerator (rat n d) * d == denominator (rat n d) * n

-- Kommutativitaet von addR:
prop_add_comm :: Rat  → Rat  → Bool
prop_add_comm m n = addR m n == addR n m

-- Assoziativitaet von addR:
prop_add_assoc :: Rat  → Rat  → Rat  → Bool
prop_add_assoc x y z = addR (addR x y) z == addR x (addR y z)

-- Kommutativitaet von mulR:
prop_mul_comm :: Rat  → Rat  → Bool
prop_mul_comm m n = mulR m n == mulR n m

-- Assoziativitaet von mulR:
prop_mul_assoc :: Rat  → Rat  → Rat  → Bool
prop_mul_assoc x y z = mulR (mulR x y) z == mulR x (mulR y z)

-- Distributivitaet von mulR und addR:
prop_mul_distr :: Rat  → Rat  → Rat  → Bool
prop_mul_distr x y z = mulR (addR x y) z == addR (mulR x z) (mulR y z)
```

Damit wir dies mit QuickCheck prüfen können, benötigen wir natürlich noch einen Generator für `Rat`:

```
instance Arbitrary Rat where
```

```
  arbitrary = do n <- arbitrary
                 d <- arbitrary `suchThat` (/=0)
                 return (rat n d)
```

Viele wichtige Datenstrukturen lassen sich als abstrakte Datentypen spezifizieren. Bei Listenstrukturen haben wir z.B. folgende ADT-Komponenten:

- Konstruktoren: `[]` und `(:)`
- Selektoren: `head` und `tail`
- Operatoren: `(++)`
- Gleichungen:

$$\begin{aligned} \texttt{head}\ (x : xs) &= x \\ \texttt{tail}\ (x : xs) &= xs \end{aligned}$$

  (und weitere Gleichungen für `(++)`)

Fallstudie: Mengenimplementierungen

Um zu zeigen, wie die Datenabstraktion hilft, die Implementierung von Datentypen auszutauschen, ohne dass sich für den Anwender etwas ändert, betrachten als Beispiel Mengen. Mengen sind Sammlungen von Objekten, bei denen es keine Reihenfolge und keine doppelten Elemente gibt. Für Mengen kann man viele Operationen definieren. Hier betrachten wir nur die folgenden Funktionen:

- `empty`: leere Menge
- `insert x s`: Element `x` zur Menge `s` hinzufügen
- `isElem x s`: ist `x` in der Menge `s` enthalten?
- `union s1 s2`: Vereinigung der Mengen `s1` und `s2`

Unabhängig von einer konkreten Implementierung sollten die folgenden Gesetze immer gelten:

$$\begin{aligned} \texttt{isElem}\ x\ \texttt{empty} &= \texttt{False} \\ \texttt{isElem}\ x\ (\texttt{insert}\ x\ s) &= \texttt{True} \\ \texttt{isElem}\ x\ (\texttt{insert}\ y\ (\texttt{insert}\ x\ s)) &= \texttt{True} \\ \texttt{isElem}\ x\ (\texttt{union}\ s_1\ s_2) &= \texttt{isElem}\ x\ s_1\ \texttt{||}\ \texttt{isElem}\ x\ s_2 \end{aligned}$$

Damit wir diese für verschiedene Implementierungen testen können, formulieren wir die Gesetze als QuickCheck-Eigenschaften und schreiben sie in ein eigenes Modul:

```
module TestSets where

import Sets
import Test.QuickCheck
```

```
type Elem = Int

prop_empty_elem :: Elem  → Bool
prop_empty_elem x = not (isElem x empty)

prop_insert_elem :: Elem  → Set Elem  → Bool
prop_insert_elem x s = isElem x (insert x s)

prop_insert_elem2 :: Elem  → Elem  → Set Elem  → Bool
prop_insert_elem2 x y s = isElem x (insert y (insert x s))

prop_elem_union :: Elem  → Set Elem  → Set Elem  → Bool
prop_elem_union x s1 s2 =
  isElem x (union s1 s2) == (isElem x s1 || isElem x s2)
```

Hierbei definieren wir ein Typsynonym für den zu wählenden Elementtyp bei den Tests, um z.B. die Tests auch mit einer kleinen Auswahl an Elementen (z.B. `Digit`) durchzuführen.

Damit wir die Mengengesetze mit QuickCheck testen können, muss es eine `Arbitrary`-Instanz für `Set` geben. Diese können wir einfach durch Erzeugung einer Liste von Elementen und Umwandlung diese Liste in eine Menge definieren:[17]

```
instance (Eq a, Arbitrary a) => Arbitrary (Set a) where
  arbitrary = do
    s <- arbitrary
    return ((foldr :: (a  → b  → b)  → b  → [a]  → b) insert empty xs)
```

Eine erste einfache und klare Referenzimplementierung von Mengen erhalten wir durch die Darstellung von Mengen als charakteristische Funktion, wie dies schon in den Übungen gemacht wurde:

```
module Sets(Set, empty, insert, isElem, union) where

data Set a = Set (a  → Bool)

empty :: Set a
empty = Set (const False)

insert :: Eq a => a  → Set a  → Set a
insert x (Set s) = Set (\y  → x == y || s y)

isElem :: Eq a => a  → Set a  → Bool
isElem x (Set s) = s x

union :: Eq a => Set a  → Set a  → Set a
```

[17] Die Typannotation bei `foldr` ist notwendig, weil in Haskell die Funktion `foldr` nicht nur auf Listen definiert ist, sondern ähnlich wie `fmap` auf „faltbaren Strukturen“ überladen ist.

```
union (Set s1) (Set s2) = Set (\x → s1 x || s2 x)
```

Damit wir die Mengengesetze mit QuickCheck testen können, muss es eine `Show`-Instanz für `Set` geben. Da wir die charakteristischen Funktionen nicht direkt anzeigen können, geben wir eine „Dummy“-Implementierung für `Show` an.

```
instance Show a => Show (Set a) where
  show _ = "some set"
```

Damit können wir erfolgreich alle Mengengesetze testen.

Eine alternative Implementierung von Mengen erhalten wir, wenn wir Mengen z.B. als Listen darstellen, in denen keine doppelten Elemente vorkommen. Mit dieser Darstellung ist die Implementierung auch recht einfach:

```
module SetsAsLists(Set, empty, insert, isElem, union) where

data Set a = Set [a]
 deriving Show

empty :: Set a
empty = Set []

insert :: Eq a => a → Set a → Set a
insert x (Set xs) = Set (if x `elem` xs then xs else x:xs)

isElem :: Eq a => a → Set a → Bool
isElem x (Set xs) = x `elem` xs

union :: Eq a => Set a → Set a → Set a
union (Set [])     s2 = s2
union (Set (x:xs)) s2 = insert x (union (Set xs) s2)
```

Man beachte, dass diese Implementierung exakt die gleiche Schnittstelle wie die Referenzimplementierung und der ADT hat. Dies bedeutet, dass wir in einem Programm die Mengenimplementierung austauschen können, indem wir lediglich `import Sets` durch `import SetsAsLists` ersetzen. Hier sehen wir einen wesentlichen Vorteil der Datenabstraktion: Wir können die Implementierung austauschen, ohne etwas bei den Anwendungsprogrammen zu ändern (bis auf den Namen des importierten Moduls).

Eine etwas effizientere Implementierung von Mengen können wir realisieren, wenn wir **Mengen als geordnete Listen** darstellen, indem wir z.B. beim Einfügen darauf achten, dass das Element an die richtige Position eingefügt wird:

```
insert :: (Eq a, Ord a) => a → Set a → Set a
insert x (Set xs) = Set (oinsert xs)
 where
  oinsert []     = [x]
  oinsert (y:ys) | x==y      = y:ys
                 | x<y       = x:y:ys
```

```
                  | otherwise = y : oinsert ys
```

Dies hat den Vorteil, dass wir zur Prüfung, ob ein Element in einer Menge vorhanden ist, im Durchschnitt nur noch die Hälfte der Elemente vergleichen müssen:

```
isElem :: (Eq a, Ord a) => a  → Set a  → Bool
isElem x (Set xs) = oelem xs
 where
  oelem []     = False
  oelem (y:ys) | x==y      = True
               | x<y       = False
               | otherwise = oelem ys
```

Dies ist aber noch keine Komplexitätsverbesserung, d.h. die Laufzeit zum Auffinden eines Elements ist immer noch linear in der Anzahl der Elemente. Was wir allerdings deutlich verbessern können, ist die Vereinigungsoperation. Während die bisherige Mengenvereinigung quadratisch in der Anzahl der Elemente beider Listen ist (wegen des wiederholten Aufrufs von `insert` und damit `elem`), können wir dies bei geordneten Listen auf eine lineare Laufzeit reduzieren, indem wir beide Listen gleichzeitig durchlaufen:

```
union :: (Eq a, Ord a) => Set a  → Set a  → Set a
union (Set s1) (Set s2) = Set (ounion s1 s2)
 where
  ounion []       ys     = ys
  ounion xs@(_:_) []     = xs
  ounion (x:xs)   (y:ys) | x==y = x : ounion xs ys
                         | x<y  = x : ounion xs (y:ys)
                         | x>y  = y : ounion (x:xs) ys
```

Auch diese effizientere, aber komplexere Implementierung können wir wieder erfolgreich mit QuickCheck testen. Leider sind die (manchmal wichtigsten) Operationen `insert` und `isElem` noch linear in der Anzahl der Mengenelemente. Dies könnten wir auf eine logarithmische Komplexität reduzieren, in dem wir ausgewogene Suchbäume als Darstellung verwenden. Dadurch wird zwar die Implementierung aufwändiger, aber diese können wir wieder mit den vorhandenen ADT-Eigenschaften testen, denn die Schnittstelle der Implementierung bleibt gleich.

4 Einführung in die Logikprogrammierung

4.1 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?“

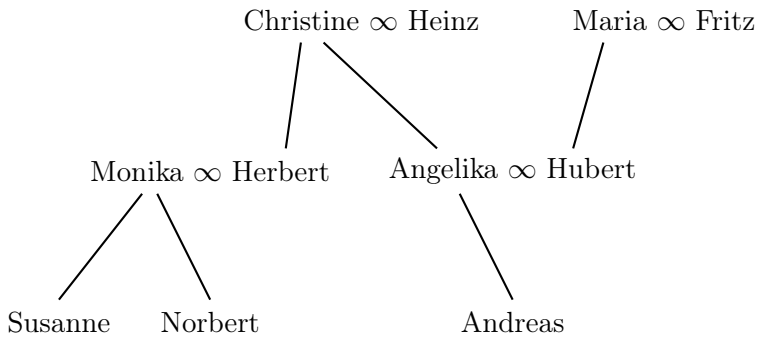
„Ist Monika die Tante von Andreas?“

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 | Fritz | Monika
            | Herbert | Angelika | Hubert
            | Susanne | Norbert | Andreas
  deriving (Eq,Show)
```

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* 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 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 Regel für die Vaterbeziehung formulieren:

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

Ebenso können wir Regeln für die Grossvaterbeziehungen angeben:

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

Weil sowohl der Vater des Vaters als auch der Vater der Mutter ein Großvater ist, geben wir hier zwei Regeln an. Man beachte, dass die linken Seiten beider Regeln identisch sind, weil beide Regeln ja echte Alternativen für die Großvaterbeziehung ausdrücken. Im Gegensatz zu Haskell, wo nur die erste anwendbare Regel verwendet wird, werden in Prolog alle Regeln ausprobiert. Regeln mit gleichen linken Seiten drücken also eine Disjunktion (logisches „Oder“) aus.

Variablen in Regeln haben die folgende Bedeutung: die Regeln sind korrekte 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. Dasselbe gilt auch für Prolog, d.h. auch Prolog-Programme können eventuell nicht terminieren. Dies hängt von der Auswertungsstrategie ab, die wir aber erst später vorstellen.

4.2 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 fac fac` entspricht `fac(fac(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 die *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
```

4.3 Elementare Programmiertechniken

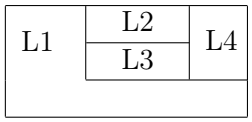
In diesem Kapitel zeigen wir einige grundlegende logische Programmiertechniken.

4.3.1 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, L4.
- 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:

```
color(red).
color(yellow).
color(green).
color(blue).
```

2. Jedes Land hat eine dieser Farben:

```
coloring(L1,L2,L3,L4) :- color(L1), color(L2), color(L3), color(L4).
```

3. Wann sind zwei Farben verschieden?

```
different(red,yellow).
different(red,green).
different(red,blue).
...
different(green,blue).
```

4. Korrekte Lösung: Aneinandergrenzende Länder haben verschiedene Farben:

```
correctColoring(L1,L2,L3,L4) :-
    different(L1,L2),
    different(L1,L3),
    different(L1,L4),
```

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

5. Gesamtlösung des Problems:

```
?- coloring(L1,L2,L3,L4), correctColoring(L1,L2,L3,L4).
L1 = red
L2 = yellow
L3 = green
L4 = blue
```

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

```
solution(S) :- potentialSolution(S), correctSolution(S).
```

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, wie wir in einem nachfolgenden Beispiel sehen werden.

Exkurs: Ungleichheit

Die Definition von `different` mit vielen Fakten erscheint sehr aufwändig. Tatsächlich würde dies auch kein erfahrener Prolog-Programmierer so machen, aber wir wollten zunächst einmal zeigen, wie man auch die Ungleichheit von Daten konstruktiv definieren kann.
Prolog bietet noch eine andere Möglichkeit, bei deren Benutzung man allerdings sorgfältig vorgehen muss. Wie wir ja wissen, kann man mit t_1 `=` t_2 ausdrücken, dass die Terme t_1 und t_2 syntaktisch gleich sein sollen. Tatsächlich kann man in Prolog auch mit

```
t1 \= t2
```

ausdrücken, dass die Terme t_1 und t_2 *syntaktisch verschieden* sein sollen. Insofern hätten wir die Definition von `different` weglassen und die Regel für `correctColoring` so schreiben können:

```
correctColoring(L1,L2,L3,L4) :-
    L1 \= L2, L1 \= L3, L1 \= L4, L2 \= L3, L2 \= L4, L3 \= L4.
```

Hierbei ist allerdings zu beachten, dass die Ungleichheit im Gegensatz zur Gleichheit keine Variablen belegt, sondern einfach nur ausprobiert, ob die Gleichheit *nicht beweisbar* ist:

```
?- X = a.
X = a
?- X \= a.
no
```

Rein logisch hätte man der letzten Anfrage erwarten können, dass eine Antwort wie `X = b` herauskommt. Allerdings ist diese Anfrage ist nicht beweisbar, weil die Umkehrung `X=a` beweisbar ist. Somit muss man darauf achten, dass die Ungleichheit nur verwendet wird, wenn die beteiligten Terme keine Variablen enthalten. Bei unserem Beispiel ist dies dadurch sichergestellt, weil mit `coloring(L1,L2,L3,L4)` alle Variablen mit einer Farbe belegt werden, bevor diese mit `correctColoring` weiter getestet werden.

Eine genaue Diskussion, wie Prolog mit negativen Aussagen umgeht, erfolgt später in Kapitel 4.5.

Sortieren von Zahlen

Um einzusehen, dass die Komplexität auch sehr groß werden kann, betrachten wir als nächstes Beispiel das Sortieren von Zahlen, `sort(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):

   ```
   sorted([]).
   sorted([_]).
   sorted([E1,E2|L]) :- E1 =< E2, sorted([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]) :- remove(E,L1,R1), perm(R1,R2).
   ```

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

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

3. Nach dem Schema erhalten wir folgende Gesamtlösung:

```
sort(UL,SL) :-
    perm(UL,SL),  % moegliche Loesung
    sorted(SL). % korrekte Loesung
```

4. Diese logische Spezifikation ist ausführbar:

```
?- sort([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.

4.3.2 Musterorientierte Wissensrepräsentation

Ein typisches Beispiel für eine 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` definieren:

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.

4.3.3 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 $\mathtt{X}_i$ 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:

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

Letztes Element einer Liste:

```
last(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:

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

Ist eine Liste Teil einer anderen?

```
sublist(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.

4.3.4 Peano-Zahlen

Als weiteres Beispiel für die logische Programmierung betrachten wir die *Peano-Darstellung* natürlicher Zahlen, die wir schon in Kapitel 3.12.4 gesehen haben. In der Peano-Darstellung ist eine natürliche Zahl wie folgt definiert:

- 0 ist eine natürliche Zahl, welche wir in Prolog mit dem Funktor `o` repräsentieren.
- Falls n eine natürliche Zahl ist, so ist auch $s(n)$ (der Nachfolger) eine natürliche Zahl. Dies stellen wir in Prolog durch `s(`n`)` dar.

Wir können also zunächst ein einstelliges Prädikat `isPeano` wie folgt definieren:

```
isPeano(o).
isPeano(s(N)) :- isPeano(N).
```

Dieses kann zum einen verwendet werden, um zu prüfen, ob ein bestimmter Wert eine Peano-Zahl ist. In der Logikprogrammierung können wir damit aber auch alle möglichen Peano-Zahlen aufzählen:

```
?- isPeano(N).
X = o;
X = s(o);
X = s(s(o));
X = s(s(s(o)))
...
```

Als nächstes definieren wir arithmetische Operationen für Peano-Zahlen. Wir beginnen zunächst mit der Nachfolger- und der partiellen Vorgängerfunktion:

```
succ(X   ,s(X)).  % Bestimmung des Nachfolgers
pred(s(X),X   ).  % Bestimmung des Vorgängers
```

Die Addition zweier Peano-Zahlen ist ebenfalls einfach:

```
add(o   ,Y,Y   ) .
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

und funktioniert genau wie die Konkatenation von Listen. Die Addition kann auch umgekehrt verwendet werden, um z.B. zu ermitteln, welche Zahlen addiert werden können, um einen gegebenen Wert zu erhalten:

```
?- add(X,Y,s(s(o))).
```

```
X = o, Y = s(s(o));
X = s(o), Y = s(o);
X = s(s(o)), Y = o;
false
```

Somit können wir auch einfach die Subtraktion unter Verwendung der Addition definieren:

```
sub(X,Y,Z) :- add(Y,Z,X).
```

Im nächsten Schritt können wir dann auch die Multiplikation und die natürliche Ordnung („kleiner oder gleich“) definieren:

```
mult(o   , _, o).
mult(s(N), M, K) :- mult(N,M,O), add(O,M,K).

leq(o   , _   ).
leq(s(N), s(M)) :- leq(N,M).
```

Als Anfragen erhalten wir:

```
?- mult(s(s(o)),s(s(o)),V).
V = s(s(s(s(o))));
false
?- leq(s(s(o)),s(o)).
false
?- leq(N,s(s(o))).
N = o ;
N = s(o) ;
N = s(s(o)) ;
false
?- mult(X,Y,s(s(o))).
X = s(o), Y = s(s(o));
X = s(s(o)), Y = s(o);
```

Danach terminiert das System leider nicht, da Prolog aus der Definition der Multiplikation nicht erkennen kann, wann es nicht mehr sinnvoll ist, weitere Werte für `X` und `Y` zu probieren. Wie es hierzu genau kommt und welche Aussagen Prolog genau heraus finden kann, betrachten wir im nächsten Kapitel.

4.4 Rechnen in der Logikprogrammierung

Rechnen in Prolog entspricht im Wesentlichen dem Beweisen von Aussagen. Aber wie beweist Prolog Aussagen? Um das zu verstehen, betrachten wir zunächst eine vereinfachte Technik: das einfache 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* 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:

> *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.

Die Ersetzung von Variablen durch andere Terme haben wir schon durch den Begriff der *Substitution* in Definition 3.4 (Kapitel 3.7) präzisiert. Substitutionen verwenden wir auch in der Logikprogrammierung, wobei wir hier Terme in der Prolog-Schreibweise notieren und mit $\sigma(t)$ die Anwendung einer Substitution σ auf einen Term t notieren.
In unserem Beispiel ist die Substitution σ wie folgt definiert:

$$\sigma = \{\texttt{K} \mapsto \texttt{susanne}, \texttt{V} \mapsto \texttt{herbert}\}$$

Die Anwendung dieser Substitution sieht dann so aus:

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

Im Fall der Logikprogrammierung müssen wir Variablen nicht nur in Regeln, sondern auch in Anfragen wie

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

ersetzen, damit wir Antworten zu Anfragen ausrechnen können. Hierzu verwendet man die *Unifikation*, was den Prozess der Ersetzung von Variablen in Termen bezeichnet, sodass die Terme syntaktisch gleich werden. Für die Terme `datum(Tag,Monat,83)` und `datum(3,M,J)` gibt es mehrere mögliche Substitutionen, die diese gleich machen:

$\sigma_1 = \{\texttt{Tag} \mapsto 3, \texttt{Monat} \mapsto 4, \texttt{M} \mapsto 4, \texttt{J} \mapsto 83\}$

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

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

Definition 4.1 (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 $\sigma' = \phi \circ \sigma$.*

Intuitiv bedeutet dies, dass man jeden Unifikator σ' durch Spezialisierung, d.h. Anwendung einer weiteren Substitution ϕ, eines allgemeinsten Unifikators σ erhalten kann.

In dieser Definition wird die *Komposition von Funktionen* $\circ$ verwendet, die wie folgt definiert ist:

$$\phi \circ \sigma(t) = \phi(\sigma(t)) \qquad \text{für alle Terme } t$$

Die Komposition ist also einfach die Hintereinanderanwendung zweier Funktionen. Wenn man Substitutionen in der Mengenschreibweise wie in Kapitel 3.7 notiert, dann ist zu

beachten, dass die Komposition nicht einfach die Vereinigung dieser Mengen ist, sondern dies wirklich als Hintereinanderausführung der dadurch beschriebenen Substitionen zu interpretieren ist. Somit gelten z.B. die folgenden Gleichheiten:

$$\begin{aligned}
\{Y \mapsto X\} \circ \{Z \mapsto 1\} &= \{Y \mapsto X, Z \mapsto 1\} \\
\{Y \mapsto X\} \circ \{Z \mapsto Y\} &= \{Y \mapsto X, Z \mapsto X\} \\
\{Z \mapsto Y\} \circ \{Y \mapsto X\} &= \{Z \mapsto Y, Y \mapsto X\} \\
\{Z \mapsto Y\} \circ \{Y \mapsto X\} &= \{Y \mapsto X, Z \mapsto Y\} \\
\{Y \mapsto X\} \circ \{Y \mapsto 2\} &= \{Y \mapsto 2\}
\end{aligned}$$

Zur Übung sollte man sich einmal überlegen, warum jede dieser Gleichheiten gilt und wie ein allgemeines Verfahren (Algorithmus) aussieht, um aus den Substitutionen in dieser Mengendarstellung die Komposition in der Mengendarstellung zu berechnen!

Beim Rechnen in der Logikprogrammierung ist es wichtig, mit mgus anstatt mit spezielleren Unifikatoren zu arbeiten, weil es dadurch weniger Beweismöglichkeiten gibt und damit auch weniger zu suchen ist. Es stellt sich also die Frage: Gibt es immer mgus und wie kann man diese berechnen?
Die Antwort hat *Robinson* im Jahr 1965 [15] gefunden: es gibt immer mgus für unifizierbare Terme. Für ihre Berechnung definieren wir den Begriff der *Unstimmigkeitsmenge von Termen*:

Definition 4.2 *Sind t, t' Terme, dann ist die* Unstimmigkeitsmenge *(*disagreement set*) $ds(t, t')$ definiert durch:*

1. *Falls $t = t'$: $ds(t, t') = \varnothing$*
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 innersten Position, an denen t und t' verschieden sind.
Daraus ergibt sich unmittelbar der folgende *Unifikationsalgorithmus*.

Unifikationsalgorithmus:

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

1. k `:= 0;` σ_0 `:=` $\{\}$
2. Falls $ds(\sigma_k(t_0), \sigma_k(t_1)) = \varnothing$, dann ist σ_k ein 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“

Die Bedingung im 2. Fall ($ds(\sigma_k(t_0), \sigma_k(t_1)) = \varnothing$) ist gleichbedeutend mit der Tatsache, dass $\sigma_k(t_0) = \sigma_k(t_1)$, d.h. beide Terme sind unter Berücksichtigung der aktuellen Substitution identisch. Man beachte im 3. Fall, dass $\{x, t\}$ eine Menge ist, d.h. dies ist dasselbe wie $\{t, x\}$.

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`$\}$
   - $ds(\sigma_2(t_0), \sigma_2(t_1)) = \varnothing$

   $\Longrightarrow$ σ_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)`$\}$
   - $\sigma_1 = \{$`Y` $\mapsto$ `f(1)`$\}$
   - $ds(\sigma_1(t_0), \sigma_1(t_1)) = \{$`g(X), 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 = `X`, t_1 = `f(X)`
   - $ds(t_0, t_1) = \{$`X, 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.

Für den Unifikationsalgorithmus gilt der folgende Satz:

Satz 4.1 (Unifikationssatz von Robinson [15]) *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:

   Für all $k \geq 0$ existiert eine 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

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

   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:

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

   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.

Bezüglich der Laufzeit des Algorithmus gibt es noch folgende Anmerkungen:

1. Auch ohne Vorkommenstest hat der Algorithmus eine exponentielle Laufzeit, da die exponentiell wachsenden Terme aufgebaut werden müssen.
2. Eine bessere Laufzeit ergibt sich wenn man auf die explizite Darstellung der Terme verzichtet und z. B. Graphen verwendet. Dadurch sind Laufzeitverbesserungen bis hin zu linearen Algorithmen möglich (siehe z. B. [8, 11, 13]).
3. Ein exponentielles Wachstum der Terme ist in der Praxis äußerst selten. Daher ist oftmals der klassische Algorithmus ausreichend in Verbindung mit dem „Sharing“ von Variablen, d.h. Variablen werden nicht direkt durch Terme ersetzt sondern durch Referenzen auf Terme (vgl. Sharing in Haskell).

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

```
?- A1,...,Ai-1,Ai,Ai+1,...,Am.
```

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

```
L :- L1,...,Ln.
```

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

```
?- σ(A1,...,Ai-1,L1,...,Ln,Ai+1,...,Am).
```

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

```
?- t0 = t1.
```

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(X1).
```

durchführen (zur Umbenennung setzen wir häufig einfach einen neuen Index an alle Regelvariablen), ist dies erfolgreich möglich, weil `p(X1)` und `p(f(X))` unifizierbar sind.

Man beachte, dass die Umbenennung von Regelvariablen auch notwendig ist, um Konflikte von Anfragevariablen mit lokalen Variablen in Regeln zu vermeiden. Betrachten wir z.B. die Regel

```
p :- X=a.
```

und die Anfrage

```
?- p, X=b.
```

Wenn die lokale Regelvariable `X` im ersten Resolutionsschritt z.B. zu `X1` umbenannt wird, dann ist diese Anfrage beweisbar:

```
?- p, X=b.
   ⊢
?- X1=a, X=b.
   ⊢ {X1↦a}
?- X=b.
   ⊢ {X↦b}
?- .
```

Wenn dagegen diese Umbenennung nicht erfolgt, dann wäre die Anfrage nicht beweisbar:

```
?- p, X=b.
   ⊢
?- X=a, X=b.
   ⊢ {X↦a}
?- a=b.
Fehlschlag!
```

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,G).`
Beweis nach dem Resolutionsprinzip:

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

   ⊢ {X1↦hans, Z1↦G}

?- vater(hans,Y1), vater(Y1,G).

   ⊢ {Y1↦peter}

?- vater(peter,G).

   ⊢ {G↦frank}

?- .
```

Antwort: `G = frank`

Auswertungsstrategie 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 4.4 (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 N 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 N genau die Kinder $N_1, \ldots, N_n$, 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).
```

Der SLD-Baum für dieses Programm und die Anfrage "`?- p(S,b).`" ist in Abbildung 4.1 dargestellt. Aus diesem SLD-Baum ist ersichtlich, dass es drei verschiedene SLD-Ableitungen für die ursprüngliche Anfrage gibt. In dieser Abbildung haben wir an die Kanten die Nummer der jeweils verwendeten Regel und den Unifikator geschrieben, wobei die Regeln durch Anhängen von Indizes an die Variablen umbenannt werden.

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.

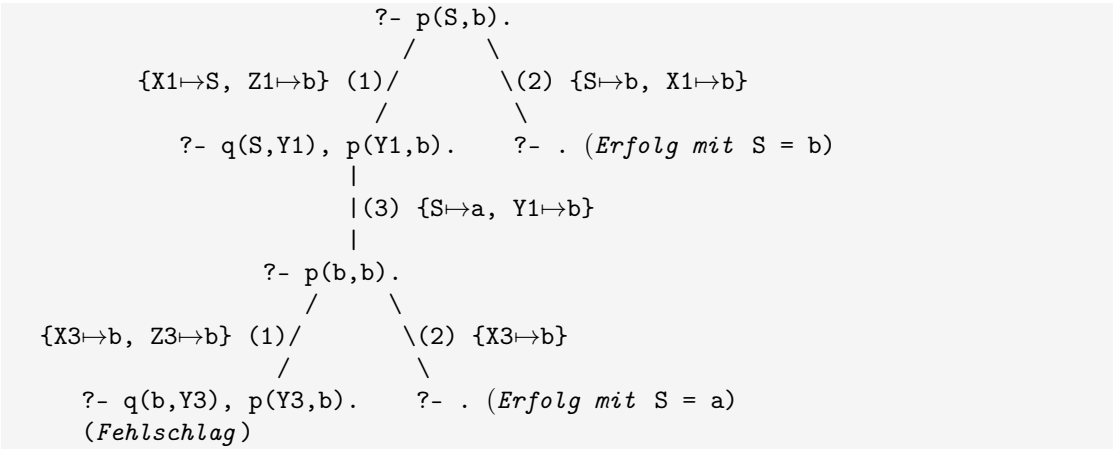


Abbildung 4.1: SLD-Baum

Auswertungsstrategie 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 (von links nach rechts) 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 letzten 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 *Auswertungsstrategie 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]}
   ?- .
```

Nachdem wir nun verstanden haben, wie Prolog versucht Aussagen zu beweisen, können wir weitergehende Eigenschaften von Prolog betrachten, welche für die Programmierung in Prolog sehr nützlich sind.

4.5 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 [7] 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 sind und fehlgeschlagen, dann ist `\+ p` beweisbar.

Die NAF-Regel ist effektiv implementierbar, indem der gesamte SDL-Baum des negierten Prädikats überprüft wird und das Ergebnis (`true` oder `false`) anschließend negiert wird.

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 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.

Wie schon früher erwähnt wurde, ist die *Ungleichheit* von Termen als Negation der Gleichheit in Prolog vordefiniert:

```
X \= Y :- \+ X = Y
```

Damit könnten wir die letzte Klausel auch wie folgt schreiben:

```
geschwister(S,P) :- mutter(S, M), mutter(P, M), S \= P.
```

Es muss also auch bei der Benutzung der Ungleichheit darauf geachtet werden, dass die beteiligten Terme variablenfrei sind, wenn dies bewiesen werden soll.

Um diese Bedingung nach Variablenfreiheit einfach sicherzustellen, bieten manche Prolog-Systeme die Möglichkeit der *verzögerten Negation*. Die Idee ist hierbei, den Beweis negativer Literale so lange zu verzögern, bis sie keine Variablen enthalten, um dadurch eine logisch sichere Negation zu erhalten.

Falls die Negation 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-System 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. Somit wäre die folgende Definition von `geschwister` auch möglich:

```
geschwister(S,P) :- when(ground([S,P], \+ S = P)), mutter(S,M), mutter(P,M).
```

4.6 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`, später)
4. Vermeidung von nicht-terminierenden Suchpfaden

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:

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

ab(a).
ab(b).

?- yes.
```

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 den Cut also sehr 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.
```

Außerdem können wir mit Hilfe des Cuts auch die Negation aus dem letzten Kapitel

```
p :- \+ q.
```

formulieren als:

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

Hierbei ist `fail` ein nie beweisbares Prädikat.

4.7 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.

4.7.1 Arithmetik in Prolog

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

1. `Y` zum Zeitpunkt des Beweises ein 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. Prolog bietet für Vergleiche die folgenden Operatoren an:

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:

```
fac(0,1).
fac(N,F) :- N > 0,
            N1 is N - 1,
            fac(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[1] ausführbar sind. Eine mögliche Lösung für dieses Problem ist die nachfolgend diskutierte Constraint-Programmierung (*Constraint Logic Programming*, CLP).

[1] Der Modus eines Prädikats gibt an, für welchen formalen Parameter welche Art von aktuellem Parameter (Variable, Grundterm, ...) erlaubt ist.

4.7.2 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 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,U,I)`: Diese steht für einen Schaltkreis `S` mit Spannung `U` 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),U,I) :- {U = I * R}.

% KIRCHHOFFsche Maschenregel
sk(reihe(S1,S2),U,I) :-
    {I = I1, I = I2, U = U1 + U2},
    sk(S1,U1,I1),
```

```
    sk(S2,U2,I2).

% KIRCHHOFFsche Knotenregel
sk(parallel(S1,S2),U,I) :-
    {I = I1 + I2, U = U1, U = U2},
    sk(S1,U1,I1),
    sk(S2,U2,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))), U, 5).
{ U = 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 eine feste monatliche Rückzahlung vorgeben:

```
?- 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 folglich eine Erweiterung der Logikprogrammierung. Es ersetzt Terme durch Constraint-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
- Constraints: 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 den Hardwareentwurf und die Hardwareverifikation
- unendliche zyklische Bäume
- Listen
- *endliche Bereiche*, welche zahlreiche Anwendungen im 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 Bereiche (finite domains) für die Praxis die wichtigsten Anwendungen hat, betrachten wir die Sprache CLP(FD) nun genauer.

4.7.3 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 finden

sich im Bereich der 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").

FD-Constraints in Prolog

Die FD-Constraints in Prolog

- sind kein Standard, aber häufig als Bibliotheken vorhanden
- haben bei verschiedenen Systemen einen unterschiedlichen Umfang ($\leadsto$ 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 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.

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:
  - `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. Prolog-Manual).

– `domain(Vs, Min, Max)`:[2] Der Wertebereich aller Variablen in der Variablenliste `Vs` muss zwischen `Min` und `Max` (einschließlich) liegen.

- 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(Opts, Vs)
```

  belegt nacheinander die Variablen in der Variablenliste `Vs` mit ihren Werten. Das erste Argument von `labeling` erlaubt dabei, zusätzliche Steuerungsoptionen als Liste anzugeben, womit man die Reihenfolge der Aufzählung beeinflussen kann (vgl. Prolog-Manual). SWI-Prolog bietet zusätzlich noch das Prädikat `label(Vs)`, das dem Aufruf `labeling([],Vs)` entspricht.

```
?- 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. Zum Beispiel liefert

```
?- 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

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

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?

Wir 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], D) :-
    no_attack(Q,Q1,D),
    D1 #= D + 1,
    safe(Q,Qs,D1).

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

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 prinzipiell $8^8 = 16.777.216$ potentielle Platzierungen 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.

4.7.4 Weitere FD-Constraints

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

`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 und `Durations` eine Variablenliste mit Zeitdauern ist, so dass die entsprechenden „Aufträge“ sich nicht überlappen. Damit kann man z.B. *Schedulingprobleme* elegant ausdrücken.

4.7.5 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

4.8 Meta-Programmierung

Unter Meta-Programmierung versteht man, dass man bei der Programmierung die Ebenen von Daten und Programmen ändert oder vermischt. Z.B. kann man Datenterme als Anfragen oder Regeln interpretieren oder man kann Ergebnisse über die Struktur von Berechnungen als Daten zusammenfassen. Hierzu bietet Prolog zahlreiche Möglichkeiten, von denen wir in diesem Kapitel einige diskutieren.

4.8.1 Prädikate höherer Ordnung

Wir haben in der funktionalen Programmierung das Konzept von Funktionen höherer Ordnung kennengelernt, d.h. Funktionen, die andere Funktionen als Argumente verarbeiten oder als Ergebnisse liefern. Hierdurch konnten Programme kompakter und wiederverwendbarer gestaltet werden. Obwohl Prolog auf der Prädikatenlogik 1. Stufe beruht, bietet Prolog ähnliche Möglichkeiten wie die funktionale Programmierung.

Um ein Prädikat oder Prädikataufruf als Argument zu anderen Prädikaten zu übergeben und dann als Anfrage zu interpretieren, bietet Prolog eine Familie von `call`-Prädikaten

an. Im einfachsten Fall kann man das Prädikat `call` mit einem Argument aufrufen, wodurch der Argumentterm als Anfrage interpretiert und abgearbeitet wird:

```
?- call(append([1,2],[3,4],X)).
X = [1, 2, 3, 4]
?- call(is(X,3+4)).
X = 7
```

Man kann aber auch weitere Argumente zu `call` hinzufügen, was dann den Effekt hat, dass diese Argumente zum ersten Argument hinzugefügt werden, bevor dieses aufgerufen wird:

```
?- call(is(X),3+4).
X = 7
?- call(is,X,3+4).
X = 7
```

Hierdurch haben wir die Möglichkeit, wie in der funktionalen Programmierung universelle Prädikate zu definieren, mit denen man seinen Code kompakter gestalten kann. Z.B. können wir die Funktion `map` aus der funktionalen Programmierung wie folgt als Prädikat definieren:

```
map_list(_,[],[]).
map_list(P,[X|Xs],[Y|Ys]) :- call(P,X,Y), map_list(P,Xs,Ys).
```

Ein Beispiel zur Benutzung ist:

```
?- map_list(is, L, [3+4,5+6,10+5]).
L = [7, 11, 15]
```

Da dieses Prädikat recht nützlich ist, wie wir aus der funktionalen Programmierung wissen, ist dieses in vielen Prolog-Systemen vordefiniert als Familie von Prädikaten `maplist`, die ein Prädikat auf eine oder mehrere Listen anwenden.[3]

```
?- maplist(is,[X,Y],[3+4,7+X]).
X = 7
Y = 14
?- maplist(<(0),[0,1,2,3]).
no
?- maplist(<(0),[1,2,3]).
yes
```

[3] In SWI-Prolog stehen diese direkt zur Verfügung, während man in SICStus-Prolog die Bibliothek `lists` mittels "`:- use_module(library(lists)).`" importieren muss.

4.8.2 Kapselung des Nichtdeterminismus

In vielen Situationen möchte man nicht nur *eine* nicht-deterministische Lösung erhalten, sondern *alle* Lösungen ermitteln und mit diesen weiter rechnen. Als Beispiel betrachten wir noch einmal unsere Modellierung der Familienbeziehungen. Hier können wir z.B. alle Mütter bzw. Großväter bestimmen wollen und diese dann im Weiteren verwenden. Hierzu können wir das Metaprädikat `findall` wie folgt verwenden:

```
muetter(Ms) :- findall(M,mutter(_,M),Ms).

grossvaeter(Gs) :- findall(G,grossvater(_,G),Gs).
```

`findall` wird wie folgt verwendet: Das zweite Argument ist ein Literal (oder auch eine Anfrage), für welches alle Lösungen berechnet werden sollen. Im ersten Beispiel wird ein Term, in welchem `mutter` als Funktor eigentlich zunächst nichts mit dem Prädikat `mutter` zu tun hat, beim Bestimmen aller Lösungen als Prädikat ausgewertet. Deshalb spricht man hier auch von Meta-Programmierung, weil Terme nun als Prädikate interpretiert werden. Die Liste der berechneten Lösungen wird mit dem dritten Argument unifiziert. Dabei wird die Form der einzelnen Listenelemente durch das erste Argument von `findall` spezifiziert, in unseren Bespiel einfach die möglichen Belegungen der Variablen `M` bzw. `G`. Auf unsere Anfrage erhalten wir

```
?- muetter(L).
L = [christine, christine, maria, monika, monika, angelika].

?- grossvaeter(L).
L = [heinz, heinz, fritz, heinz].
```

Es werden also alle möglichen Lösungen aufgezählt und für jede Lösung die Belegung der Variablen (`M` bzw. `G`) in die Ergebnisliste eingetragen. Dies wird noch deutlicher, wenn wir im Programm auch die Kinder bzw. Enkel mit in unsere Liste eintragen:

```
muetter(Ms) :- findall((K,M),mutter(K,M),Ms).

grossvaeter(Gs) :- findall((E,G),grossvater(E,G),Gs).

?- muetter(L).
L = [ (herbert, christine), (angelika, christine), (hubert, maria),
(susanne, monika), (norbert, monika), (andreas, angelika)].

?- grossvaeter(L).
L = [(susanne, heinz), (norbert, heinz), (andreas, fritz), (andreas, heinz)].
```

Das Prädikat `findall` können wir also nur anwenden, wenn der SLD-Suchbaum für das Prädikat im zweiten Argument endlich ist und alle Lösungen durch Prolog effektiv bestimmt werden können. Ähnlich wie bei der Negation würde `findall` sonst nicht terminieren.

Man bezeichnet `findall` oft auch als Kapsel, in welcher die Suche bzw. der Nichtdeterminismus eingekapselt wird und eben alle Lösungen berechnet werden. Hierbei erhält man wie oben oft auch bestimmte Lösungen mehrfach, da alle Lösungen in eine Liste geschrieben werden. Im Beispiel ist `christine` eben zweifache Mutter und wird deshalb auch zwei Mal in der Liste aufgeführt.

Manchmal ist es gewünscht, dass Lösungen nur bezüglich bestimmter Variablen gekapselt und den Nichtdeterminismus über den anderen Variablen „außen“ beibehalten wird. Hierzu kann man das Meta-Prädikat `bagof` verwenden, welches die verschiedenen Lösungen nur für die Variablen kapselt, die innerhalb von `bagof` eingeführt werden. Nehmen wir an, wir möchten ein Prädikat definieren, mit dem man alle Kinder einer Mutter in einer Liste aufsammelt. Mittels `findall` könnten wir dies wie folgt machen:

```
kinder(M,Ks) :- findall(K, mutter(K,M), Ks).
```

Wenn wir die Kinder einer gegebenen Person wissen wollen, verhält es sich wie erwartet:

```
?- kinder(monika,Ks).
Ks = [susanne, norbert].
```

Wenn wir allerdings für alle Personen die Kinder wissen wollen, ist das Ergebnis zunächst unerwartet:

```
?- kinder(M,Ks).
Ks = [herbert,angelika,hubert,susanne,norbert,andreas]
```

Dies liegt daran, dass beim Aufruf von `findall` die Variable `M` ungebunden ist und somit dieser Aufruf sich wie die folgende Anfrage verhält:

```
?- mutter(K,_).
K = herbert ;
K = angelika ;
K = hubert ;
K = susanne ;
K = norbert ;
K = andreas.
```

Bei `findall` werden also alle ungebundenen Variablen im zweiten Argument gleich behandelt und die Lösungen aufgesammelt. Woran wir hier eigentlich interessiert sind, sind Bindungen für `M` und für jede Bindung die Liste aller Lösungen für `K`. Genau dies macht `bagof`, welches nur die Bindungen für Variablen, die nicht außerhalb des Aufrufs von `bagof` vorkommen, kapselt:

```
kinder(M,Ks) :- bagof(K, mutter(K,M), Ks).
```

Nun verhält sich das Prädikat wie erwartet:

```
?- kinder(monika,Ks).
Ks = [susanne, norbert].
```

```
?- kinder(M,Ks).
M = angelika,
Ks = [andreas] ;
M = christine,
Ks = [herbert, angelika] ;
M = maria,
Ks = [hubert] ;
M = monika,
Ks = [susanne, norbert].
```

Weil der Nichtdeterminismus anderer Variablen durch `bagof` nicht gekapselt wird, erhält man für jede Lösung dieser anderen Variablen eine Lösungsliste:

```
vater(Vs) :- bagof(V,vater(_,V),Vs).

?- vater(Vs).
Vs = [hubert] ;
Vs = [heinz] ;
Vs = [heinz] ;
Vs = [fritz] ;
Vs = [herbert] ;
Vs = [herbert].
```

Hier ist die andere Variablen anonym, sodass deren Bindung nicht ausgegeben wird. Wenn man dagegen die anderen Variablen in die Kapsel aufnehmen will, ohne ihre Werte im Ergebnis zu sehen, so kann man diese Variablen innerhalb einer separaten Regel einführen:

```
istVater(V) :- vater(_,V).

vater(Vs) :- bagof(V, istVater(V), Vs).

?- vaeter(Vs).
Vs = [heinz, heinz, fritz, herbert, herbert, hubert].
```

Eine elegantere Möglichkeit ist, diese Variablen direkt innerhalb von `bagof` einzuführen. Hierzu kann man diese mit dem Operator `^` in der gekapselten Anfrage einführen (dies entspricht einem Existenzquantor in der Prädikatenlogik):

```
vaeter(Vs) :- bagof(V, K^vater(K,V), Vs).

?- vaeter(Vs).
Vs = [heinz, heinz, fritz, herbert, herbert, hubert].
```

Ähnlich wie `bagof` funktioniert auch `setof`. Der einzige, aber in der Praxis oft wichtige Unterschied, ist, dass die Lösungsliste keine doppelten Elemente enthält und zusätzlich noch sortiert ist:

```
 ?- setof(V, K^vater(K,V), Vs).
Vs = [fritz, heinz, herbert, hubert].
```

Man sollte aber generell darauf achten, dass man mit dem Sammeln von Lösungen vorsichtig umgeht. Hierdurch geht oft der deklarative Charakter eines Prolog-Programms verloren und man programmiert eher (nichtdeterministisch) prozedural.

4.8.3 Veränderung der Wissensbasis

Prolog wurde insbesondere auch dafür entworfen, dass man das Wissen des Systems dynamisch verändern kann. Dies ist zum einen durch Hinzuladen weiterer Module möglich. Es gibt aber auch einige Meta-Prädikate, welche es erlauben, während der Programmausführung angegebene Terme als Fakten und Regeln zur Wissensbasis hinzuzufügen bzw. zu entfernen:

- `assert/1` fügt den übergebenen Term als Faktum oder Regel zur Wissensbasis hinzu.
- `retract/1` entfernt das Faktum oder die Regel, die zu dem übergebenen Term passt, aus der Wissensbasis. Das Prädikat `retract` ist nur erfolgreich, wenn das übergebene Faktum, dass auch ein Muster sein kein, entfernt werden konnte. Mittels Backtracking kann man auch mehrere Fakten oder Regeln entfernen.

Generell gilt, dass Prädikate, die mittels `assert` oder `retract` verändert werden, als dynamisch (`dynamic`) deklariert werden müssen. Z.B. muss in unserem Verwandtschaftsbeispiel die folgende Definition hinzugefügt werden, damit die Prädikate `mutter` und `ehemann` veränderbar werden:

```
:- dynamic ehemann/2, mutter/2.
```

Werden neue Prädikate direkt mittels `assert` definiert, dann sind diese automatisch dynamisch.

Da Prolog die SLD-Resolution bekanntlich über eine Tiefensuche implementiert, kann es durchaus eine Rolle spielen, in welcher Reihenfolge die Fakten zum Code hinzugefügt werden. Mehr Flexibilität bieten hier die Varianten `asserta` und `assertz`, welche die Fakten vorne bzw. hinten hinzufügen.

Enthält das einzufügende Prädikat eine freie Variable, so wird auch in dem eingefügten Faktum eine Variable verwendet. Hierbei handelt es sich dann aber um eine frische Variable, die Bindung zur verwendeten Variablen geht verloren. In der Regel sollte man aber auch besser keine Variablen in den eingefügten Fakten verwenden, sondern darauf achten, dass alle Variablen im Argument bei der Ausführung von `assert` gebunden sind.

Wie schon erwähnt wurde, können neben Fakten auch Regeln eingefügt bzw. entfernt werden. Sie werden wie gewöhnlich mit dem Toplevel-Funktor "`:-`" notiert. Als Beispiel fügen wir zu unserer Familie noch eine Regel für das Prädikat `istMutter/1` hinzu (die doppelten Klammern sind notwendig, damit die Klausel mit dem Funktion "`:-`" als ein Argument zusammengefasst wird):

```
?- assert((istMutter(M) :- mutter(_,M))).
true.

?- istMutter(M).
M = christine ;
M = christine ;
M = maria ;
M = monika ;
M = monika ;
M = angelika.
```

4.8.4 Meta-Interpretierer

Klauseln können auch als Terme interpretiert werden, d.h. **Programme** können auch **als Daten** aufgefasst und manipuliert werden. Dies wurde schon in den oben vorgestellten Prädikaten ausgenutzt. Eine weitere interessante Anwendung ist die Programmierung eines Interpreters für Prolog in Prolog, was auch als *Meta-Interpretierer* bezeichnet wird.

In Meta-Interpretierer hat den Vorteil, dass dieser leicht änderbar und erweiterbar ist. Damit kann man z.B.

- die Beweisstrategie ändern,
- den Sprachumfang erweitern, oder auch
- neue Programmierumgebungen implementieren.

Als Beispiel betrachten wir zunächst einen einfachen Meta-Interpretierer für Prolog. Dieser kann wie folgt implementiert werden:

```
prove(true) :- !.                    % Rumpf von Fakten
prove( (A,B) ) :- !,
    prove(A),                        % Beweise 1. Teilziel
    prove(B).                        % Beweise 2. Teilziel
prove(A) :-
    clause(A,G),                     % Suche passende Klausel
    prove(G).                        % Beweise Klauselrumpf
```

Hierbei können wir mit dem Prädikat `clause(H,B)` auf die vorhandenen Klauseln zugreifen. Dieses Prädikat ist beweisbar, falls die Klausel "`H :- B.`" existiert, wobei `H` keine Variable sein darf. Mittels Backtracking kann man damit alle Klauseln eines Prädikates aufzählen. Bei Fakten wird der Rumpf `B` als `true` interpretiert.[4]

Eine Beispielinterpretation des Programms

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

läuft dann wie folgt ab:

[4] Wie auch bei der Änderung der Datenbank müssen in SCIStus-Prolog alle Prädikate, auf die mit `clause` zugegriffen werden soll, als dynmisch deklariert werden!

```
?- prove(append([a,f],[f,e],L)).
   ⊢
?- clause(append([a,f],[f,e],L),G), prove(G).
   ⊢
?- prove(append([f],[f,e],L1)).
   ⊢
?- clause(append([f],[f,e],L1),G1), prove(G1).
   ⊢
?- prove(append([],[f,e],L2)).
   ⊢
?- clause(append([],[f,e],L2),G2), prove(G2).
   ⊢
?- prove(true).
   ⊢
?- .

L = [a,f,f,e]
```

Erweiterung des einfachen Meta-Interpretierers:

Um die Flexibilität von Meta-Interpretierern zu demonstrieren, wollen wir diesen so erweitern, dass er die *Länge eines Beweises*, d.h. die Anzahl der Resolutionsschritte, berechnet. Dazu fügen wir ein zweites Argument (die Beweislänge) hinzu:

```
lprove(true,0) :- !.                  % Fakten: Beweislaenge = 0
lprove((A,B),L) :- !,
     lprove(A,LA),
     lprove(B,LB),
     L is LA+LB.                      % Laenge = Summe der Teilbeweise
lprove(A,L) :-
     clause(A,G),
     lprove(G,LG),
     L is LG+1.                       % 1 Schritt fuer die Klausel

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

?- lprove(append([a,b,c],[d],[a,b,c,d]),Len).
Len = 4
```

Die Implementierung von Meta-Interpretierern bietet eine Reihe weiterer Möglichkeiten:

- Ausgabe des Beweisverlaufes
- Interpretierer mit Tiefenbeschränkung
- *Vollständige Breitensuche* durch wiederholte tiefenbeschränkte Suche
- Interpretierer mit *Lemmagenerierung*

4.9 Weitere Aspekte von Prolog

Prolog ist nicht nur eine Sprache zum Lösen von Rätseln, sondern eine vollwertige Programmiersprache, in der auch komplette Anwendungen implementiert werden. Daher skizzieren wir nachfolgend noch einige weitere Aspekte von Prolog, die z.B. für die Anwendungsprogrammierung relevant sind.

4.9.1 Ein- und Ausgabe von Daten

In den bisherigen Beispielen wurden alle Eingaben im Prolog-System gemacht (Fakten/Regeln, Anfragen) und die Ausgaben erfolgten als Antworten zu Anfragen, wie z.B.

```
?- last([a,f,f,e)], L).
L = e
```

Um interaktive Programme zu schreiben, bietet Prolog viele Möglichkeiten zur Eingabe und Ausgabe von Daten (siehe auch Handbücher der Prolog-Systeme). Hierzu werden, wie in imperativen Sprachen, Prädikate verwendet, die Seiteneffekte haben. Da die Berechnungsstrategie von Prolog genau festgelegt ist, weiß man auch präzise, wann diese Seiteneffekte passieren. Einige wichtige Prädikate sind:

1. `write(X)`
   - `X` ist ein beliebiger Term
   - Das Prädikat ist immer beweisbar.
   - Nebeneffekt beim Beweis: Ausgabe des Terms `X` (mit Berücksichtigung der Operatoren, Listen in eckigen Klammern, Atome ohne Apostroph)
   - Es gibt nur einen Beweis.
   - Die Ausgabe wird beim Backtracking nicht rückgängig gemacht.

   Beispiel

   ```
   ?- write(+(2,3)).
   2+3
   yes
   ?- write(affe).
   affe
   yes
   ?- write('Prolog ist toll').
   Prolog ist toll
   yes
   ```

2. `nl`
   Diese Prädikat ist immer beweisbar und gibt als Nebeneffekt einen Zeilenvorschub aus.

   ```
   ?- write(a), nl, write(b), nl, write(c).
   a
   ```

```
b
c
yes
```

3. `read(X)`
   - Nebeneffekt beim Beweis: Der nächste Term wird von der Eingabe gelesen. Hinter dem Term muss ein Punkt und ein Zeilenvorschub/Leerzeichen folgen.
   - Das Prädikat ist beweisbar, wenn der gelesene Term mit `X` unifizierbar ist.
   - Es gibt nur einen Beweis.
   - Die Eingabe wird beim Backtracking nicht rückgängig gemacht.

Beispiel: Wir wollen für das Verwandtschaftsbeispiel folgendes interaktives Programm realisieren (die Benutzereingaben sind hierbei unterstrichen):

```
?- vater.
Von welcher Person moechten sie den Vater wissen?
susanne.
Der Vater von susanne ist herbert.
```

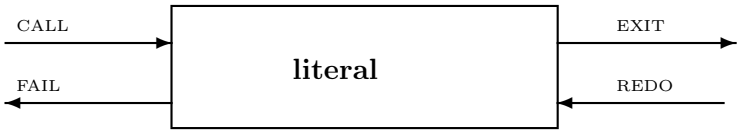
Die folgende Definition implementiert diesen einfachen Dialog:

```
vater:- write('Von welcher Person moechten Sie den Vater wissen?'), nl,
        read(P),
        vater(P,V),
        write('Der Vater von '), write(P), write(' ist '),
        write(V), write('.'), nl.
```

Wir können natürlich auch einzelne Zeichen einlesen und ausgeben (z.B. mit den Prädikaten `get_char`, `get_code`, `get_byte`, `put_char`, `put_code`, `put_byte`), Dateien lesen und schreiben (z.B. mit den Prädikaten `open`, `close`) und vieles mehr.

4.9.2 Debugging von Prolog-Programmen

Zur **Visualisierung des Beweisverlaufes** bei Prolog-Programmen bieten Prolog-Systeme *Tracer* an, die auf folgendem **Box-Modell** [4] beruhen:



- Jedes Literal, das im Beweisverlauf auftritt, wird durch eine **Box** dargestellt.
- Der Prolog-Tracer zeigt jedes Betreten und Verlassen einer Box (durch die oben dargestellten **Ports**) an. Hierbei bedeuten die einzelnen Ports:

  `call` erstmaliger Beweisversuch des Literals

  `exit` erfolgreicher Beweis des Literals

`redo` nächster Beweisversuch des Literals (Backtracking)

`fail` Fehlschlag für dieses Literal

- Die Boxen haben im allgemeinen eine hierarchische Struktur (jede Box enthält die angewendeten Regeln)

Beispiel: Gegeben sei das folgende Programm:

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

Der Debugger/Tracer wird mit `trace`/`notrace` ein- bzw. ausgeschaltet:

```
?- trace, p(X), q(X).
Call: p(_15)
Exit: p(a)
Call: q(a)
Fail: q(a)
Redo: p(a)
Exit: p(b)
Call: q(b)
Exit: q(b)
X = b
```

Tracer sind zwar in Prolog-Systemen fest eingebaut, aber wir können einen eigenen Tracer auch durch einen Meta-Interpretierer einfach implementieren:

```
tprove(true) :- !.
tprove( (A,B) ) :- !,
   tprove(A),
   tprove(B).
tprove(A) :-
   callfail(A),            % Ausgabe call/fail-Port
   clause(A,G),
   tprove(G),
   exitredo(A).            % Ausgabe exit-redo-Port

callfail(L) :- write('Call: '), write(L), nl.
callfail(L) :- write('Fail: '), write(L), nl, fail.

exitredo(L) :- write('Exit: '), write(L), nl.
exitredo(L) :- write('Redo: '), write(L), nl, fail.

?- tprove( (p(X),q(X)) ).
Call: p(X)
Exit: p(a)
Call: q(a)
Fail: q(a)
```

```
Redo: p(a)
Exit: p(b)
Call: q(b)
Exit: q(b)

X = b
```

4.10 Differenzlisten

Differenzlisten sind kein Konstrukt von Prolog, aber sie wurden im Rahmen von Prolog entwickelt und werden z.B. eingesetzt, um natürlichsprachliche Sätze effizient zu verarbeiten.

Die Motivation zur Entwicklung von Differenzlisten stammt aus dem Wunsch, Listen schneller zu konkatenieren. Betrachten wir dazu unser bekanntes Prädikaten `append`:

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

Da dieses über den Aufbau des ersten Argumentes definiert ist, ist die Laufzeit linear zur Länge der ersten Liste. Wir könnten die Laufzeit verbessern, wenn wir einen direkten Zugriff auf das Ende der ersten Liste hätten. Zu diesem Zweck repräsentieren wir eine Liste als *Differenzliste*, d.h. als „Differenz“ zweier Listen dar.

Beispiel: Die Terme

```
[a,b,c,d] - [d]
[a,b,c] - []
[a,b,c,d,e,f] - [d,e,f]
[a,b,c|L] - L
```

repräsentieren alle die Liste `[a,b,c]`.

Allgemein stellen wir bei Verwendung von Differenzlisten eine Liste `[`e_1`,...,`e_n`]` dar durch das Paar `[`e_1`,...,`e_n`|L]-L`. Die leere Liste entspricht dann dem Paar `L-L`

Der Vorteil ist offensichtlich: Ist `M-L` eine Differenzliste, dann ist `L` das „Ende“ der Liste, d.h. wir haben Zugriff auf das Ende der Liste in konstanter Zeit!

Damit können wir die Konkatenation von Differenzlisten durch ein Faktum definieren, das in konstanter Zeit ausgeführt wird:

```
append_dl(L-M, M-N, L-N).
```

Beispiel: Die Konkatenation von `[1,2]` und `[3,4]` wird durch einen einzigen Resolutionsschritt erledigt:

```
?- append_dl([1,2|L1]-L1, [3,4|L2]-L2, L3).
⇝ L3 = [1,2,3,4|L2]-L2
```

Laufzeitverbesserung mit Differenzlisten:
Betrachten als Beispiel das Umkehren der Reihenfolge aller Elemente einer Liste. Die direkte und einfache Lösung ist:

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

Die Laufzeit ist hierbei quadratisch in Abhängigkeit von der Länge der 1. Liste, weil für jedes Listenelement `E` eine Konkatenation von `UR` mit `[E]` erforderlich ist.
Da hierbei die Ergebnisliste von `rev` konkateniert wird, nehmen wir für das 2. Argument von `rev` Differenzlisten:

```
rev_dl([],L-L).
rev_dl([E|R],L-M) :- rev_dl(R,UR-T), append_dl(UR-T,[E|N]-N,L-M).
```

Da `append_dl` durch ein einfaches Faktum definiert ist, rechnen wir dieses direkt aus und repräsentieren die Differenzliste durch 2 Argumente:

```
rev_dl([],L,L).
rev_dl([E|R],L,M) :- rev_dl(R,L,[E|M]).

rev(L,M) :- rev_dl(L,M,[]).
```

Damit erhalten wir ein lineare Laufzeit für diese verbesserte Definition!

Differenzlisten sind auch für viele andere Probleme einsetzbar. Allerdings muss man beachten, dass Differenzlisten kein universeller Ersatz für normale Listen sind, denn sie können nur einmal konkateniert werden!
Beispiel:

```
?- DL = [a|L]-L, append_dl(DL,[b|M]-M,X), append_dl(DL,[c|N]-N,Y).
⇝ no!
```

Diese Eigenschaft muss man als Programmierer sicherstellen.

4.11 Parsing und Grammatiken in Prolog

Differenzlisten werden verwendet, um Sätze zu analysieren, die durch Grammatiken beschrieben werden. Hierzu bietet Prolog eine syntaktische Erweiterung an, mit der man einfach Grammatiken aufschreiben kann. Tatsächlich ist die Analyse natürlichsprachlicher Sätze einer der ursprünglichen Anwendungen von Prolog.
Allgemein legt die *Syntax* einer Sprache fest, welche Sätze formal zulässig sind, d.h. welche zu der Sprache gehören. Zum Beispiel ist „Prolog ist eine Programmiersprache.“ ein korrekter deutscher Satz, wohingegen „Prolog Programmiersprache ist.“ kein syntaktisch korrekter Satz ist. Ebenso beschreibt die Syntax einer Programmiersprache, welche Programme formal zulässig sind.

Um die zulässigen syntaktischen Strukturen präzise zu beschreiben, verwendet man üblicherweise *Grammatiken*. Obwohl es für Grammatiken unterschiedliche Formen gibt, verwendet man zur Beschreibung häufig kontextfreie Grammatiken, die aus Regeln der Form

$$A \rightarrow B_1 \dots B_n$$

bestehen. Hierbei ist die linke Seite A ein *Nichtterminalsymbol* und die rechte Seite eine Folge von Symbolen, wobei jedes Element B_i entweder ein Nichtterminalsymbol oder ein *Terminalsymbol* ist. Ein Terminalsymbol kann nicht weiter abgeleitet werden, während Nichtterminalsymbole eine Menge ableitbarer Sätze beschreiben. Ohne den Ableitungsprozess genauer zu definieren, kann man diese Regeln ähnlich zu Termersetzungsregeln interpretieren (wobei keine Terme, sondern nur Strings ersetzt werden): ein Nichtterminalsymbol in einer Symbolfolge kann durch Anwendung einer Grammatikregel durch die Symbole auf der rechten Seite ersetzt werden. Zudem gibt es noch ein ausgezeichnetes Startsymbol, sodass die Sprache, die durch eine Grammatik beschrieben ist, als die Menge aller aus dem Startsymbol ableitbaren Sätze definiert ist.

In Prolog gibt es eine feste Notation für Grammatiken, die auch als *Definite Clause Grammars* oder *DCG*s bekannt ist. Hierbei wird der Grammatikpfeil "$\rightarrow$" durch "`-->`" notiert, die Symbole auf der rechten Seite durch Komma getrennt und einen Punkt abgeschlossen und Terminalsymbole in eckigen Klammern eingefasst.

Beispiel: Die folgende Grammatik beschreibt eine Sprache mit einigen einfachen deutschen Sätzen:

```
satz       --> subjekt, verb, objekt.

subjekt    --> artikel, substantiv.

objekt     --> artikel, substantiv.

artikel    --> [die].
artikel    --> [eine].

substantiv --> [maus].
substantiv --> [katze].

verb       --> [jagt].
verb       --> [beisst].
```

Dies ist ein zulässiges Prolog-Programm und implementiert gleichzeitig einen *Parser*, d.h. ein Programm, das feststellt, ob eine Symbolfolge ein Satz der Sprache ist. Hierzu werden beim Einlesen der Grammatik die Nichtterminalsymbole als Prädikate mit zwei Argumenten angereichert, wobei das erste Argument den zu verarbeitenden Satz darstellt und das zweite Argument die unverarbeiteten Satzsymbole sind (im Prinzip entspricht dies der Darstellung des Eingabesatzes als Differenzliste). Um also festzustellen, ob ein Satz

L aus dem Startsymbol `satz` ableitbar ist, können wir prüfen, ob das Literal `satz(`L`,[])` beweisbar ist:

```
?- satz([die,katze,jagt,die,maus],[]).
yes
?- satz([die,katze,beisst,eine,katze],[]).
yes
```

Wir können an Stelle des Satzes auch eine Variable einsetzen, um alle ableitbaren Sätze zu generieren:

```
?- satz(L,[]).
L = [die, maus, jagt, die, maus]
L = [die, maus, jagt, die, katze]
L = [die, maus, jagt, eine, maus]
L = [die, maus, jagt, eine, katze]
L = [die, maus, beisst, die, maus]
L = [die, maus, beisst, die, katze]
L = [die, maus, beisst, eine, maus]
L = [die, maus, beisst, eine, katze]
L = [die, katze, jagt, die, maus]
L = [die, katze, jagt, die, katze]
L = [die, katze, jagt, eine, maus]
L = [die, katze, jagt, eine, katze]
L = [die, katze, beisst, die, maus]
L = [die, katze, beisst, die, katze]
L = [die, katze, beisst, eine, maus]
L = [die, katze, beisst, eine, katze]
L = [eine, maus, jagt, die, maus]
L = [eine, maus, jagt, die, katze]
L = [eine, maus, jagt, eine, maus]
L = [eine, maus, jagt, eine, katze]
L = [eine, maus, beisst, die, maus]
L = [eine, maus, beisst, die, katze]
L = [eine, maus, beisst, eine, maus]
L = [eine, maus, beisst, eine, katze]
L = [eine, katze, jagt, die, maus]
L = [eine, katze, jagt, die, katze]
L = [eine, katze, jagt, eine, maus]
L = [eine, katze, jagt, eine, katze]
L = [eine, katze, beisst, die, maus]
L = [eine, katze, beisst, die, katze]
L = [eine, katze, beisst, eine, maus]
L = [eine, katze, beisst, eine, katze]
```

Da die Nichtterminalsymbole in normale Prädikate mit zwei zusätzlichen Argumenten übersetzt werden, kann man auch Argumente zu Nichtterminalsymbolen hinzufügen. Außerdem darf man auf der rechten Seite auch Bedingungen schreiben, die zur Unterscheidung anderer Symbole in geschweifte Klammern gesetzt werden müssen. Hierdurch kann

man auch Kontextabhängigkeiten prüfen, was gerade zur Verarbeitung natürlichsprachlicher Sätze relevant ist. Mittels DCGs kann man also mehr als nur kontextfreie Sprachen analysieren und es ist auch erlaubt, dass auf der linken Seite von Grammatikregeln noch Kontextbedingungen in Form von Terminalsymbolen stehen.

Betrachten wir als weiteres Beispiel die folgende Grammatik:

```
abc --> as, bs, cs.

as --> [a].
as --> [a], as.

bs --> [b].
bs --> [b], bs.

cs --> [c].
cs --> [c], cs.
```

Diese Grammatik beschreibt alle Folgen von as, bs und cs:

```
?- abc([a,a,a,b,b,b,b,c,c,c],[]).
yes
```

Nun wollen wir alle Folgen mit jeweils gleich vielen as, bs und cs beschreiben. Dies ist bekanntermaßen keine kontextfreie Sprache, aber wir können dies trotzdem mit DCGs beschreiben, indem wir Argumente zum Zählen und Bedingungen verwenden:

```
abc_equal --> as(A), bs(B), cs(C), { A=:=B, B=:=C }.

as(1)   --> [a].
as(N+1) --> [a], as(N).

bs(1)   --> [b].
bs(N+1) --> [b], bs(N).

cs(1)   --> [c].
cs(N+1) --> [c], cs(N).
```

Damit sind nun nur noch Folgen mit gleich vielen Vorkommen ableitbar:

```
?- abc_equal([a,a,b,b,c,c],[]).
yes
?- abc_equal([a,a,b,c,c],[]).
no
```

Mittels Argumenten für Nichtterminalsymbole kann man nicht nur Kontextbedingungen abprüfen, sondern man kann auch Ableitungsbäume berechnen und damit Übersetzer implementieren. Als Übungsaufgabe kann man z.B. eine Grammatik für einen Parser definieren, der Binärzahlen einliest und den dazugehörigen Dezimalwert ausgibt, wie

z.B.:

```
?- binaer(Wert,[1,0,1],[]).
Wert = 5
yes
?- binaer(Wert,[2,1,0],[]).
no
```

Dies soll nur einen kleinen Einblick in die Mächtigkeit von Define Clause Grammars geben und zeigen, dass Prolog auch für viele andere Anwendungen verwendet werden kann. Als weiteres Beispiel für diese Grammatiken zeigen wir die Definition der DCG-Syntax selbst in Form einer DCG, wobei das Startsymbol `grammatikregel` ist:

```
grammatikregel      --> linke_seite, ['-->'], rechte_seite.

linke_seite         --> nichtterminalsymbol.
linke_seite         --> nichtterminalsymbol, [','], terminalsymbole.

rechte_seite        --> alternativen.

alternativen        --> regel_rumpf.
alternativen        --> regel_rumpf, [';'], alternativen.

regel_rumpf         --> regel_element.
regel_rumpf         --> regel_element, [','], regel_rumpf.

regel_element       --> nichtterminalsymbol.
regel_element       --> terminalsymbole.
regel_element       --> bedingung.
regel_element       --> ['!'].
regel_element       --> ['('], alternativen, [')'].

nichtterminalsymbol --> atom.
nichtterminalsymbol --> atom, ['('], argumente, [')'].

argumente           --> term.
argumente           --> term, [','], argumente.

terminalsymbole     --> ['[]'].
terminalsymbole     --> ['['], terme, [']'].

bedingung           --> ['{'], literale, ['}'].

terme               --> term.
terme               --> term, [','], terme.

literale            --> literal.
literale            --> literal, [','], literale.
```

5 Multiparadigmen-Sprachen

Wir haben in dieser Vorlesung verschiedene Programmiersprachen und Programmierparadigmen kennengelernt, die für unterschiedliche Aufgaben eingesetzt werden können. Bei großen Softwaresystemen wird selten nur eine Sprache verwendet, sondern unterschiedliche Teile werden in der jeweils am besten geeigneten Sprache realisiert.

Da die Verwendung unterschiedlicher Sprachen auch einen gewissen Aufwand erfordert, stellt sich die Frage, ob man nicht auch verschiedene Programmierparadigmen in einer Sprache realisieren kann. In diesem Fall spricht man dann von *Multiparadigmen-Sprachen*. Dies ist tatsächlich möglich und wir wollen einige Ansätze nachfolgend skizieren.

5.1 Imperative Ansätze

Bisher haben wir schon einige Ansätze zur Integration unterschiedlicher Programmierparadigmen kennengelernt. In Kapitel 2 haben wir gesehen, wie man nebenläufige und imperative Programmierung kombinieren kann. Nebenläufige Programmierung kann man auch mit funktionaler oder logischer Programmierung kombinieren, was wir hier aber nicht weiter betrachten.

In Kapitel 3.9 haben wir gesehen, wie man imperative Aspekte in funktionalen Sprachen integrieren kann. Für eine konzeptuell saubere Integration ist es notwendig, den sequenziellen Charakter imperativer Programmierung mit flexiblen Auswertungsstrategien funktionaler Programmierung zu kombinieren. Hierzu wird in Haskell das Typsystem ausgenutzt, wodurch diese Aspekte klar getrennt werden können.

In Kapitel 3.5.7 haben wir Ansätze skizziert, Funktionen höherer Ordnung aus funktionalen Sprachen auch in imperativen Sprachen nutzbar zu machen, wobei wir insbesondere einige Fallstricke diskutiert haben. Die Programmiersprache Scala[1] ist ein erfolgreicher Ansatz, objektorientierte und funktionale Ansätze zu kombinieren. Auch wenn hier gute Konzepte zur Integration realisiert sind (z.B. wird deutlich zwischen veränderbaren und nicht veränderbaren Variablen unterschieden), hat man keine so deutliche Trennung imperative und funktionaler Aspekte wie in funktionalen Sprachen (z.B. können Funktionen auch Seiteneffekte beinhalten.

[1] `https://www.scala-lang.org/`

5.2 Curry

Funktionale und logische Programmiersprachen basieren beide auf mathematischen Kalkülen. Daher sind diese gut für die Integration in einer Sprache geeignet [2]. Aus diesem Grund wurden unterschiedliche Ansätze entwickelt, diese Programmierparadigmen in einer Sprache zu integrieren. Im folgenden wollen wir die Programmiersprache Curry[2] betrachten, dessen Syntax sehr ähnlich zu Haskell ist, diese aber um logische Aspekte erweitert.

Als einführendes Beispiel betrachten wir die Modellierung von Verwandtschaftsbeziehungen aus Kapitel 4.1 in Haskell:

```
data Person = Christine | Heinz | Maria | Fritz | Monika
            | Herbert | Angelika | Hubert
            | Susanne | Norbert | Andreas
 deriving (Eq,Show)

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

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

vater :: Person  → Person
vater k = ehemann (mutter k)

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

Wir hatten dort festgestellt, dass wir damit einige einfache Berechnungen durchführen können, allerdings für komplexere Fragen wie

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

Variablen in Ausdrücken benötigen, die aber in Haskell nicht zulässig sind. Curry erweitert Haskell um logische Aspekte wie Nichtdeterminismus, Lösungssuche und logische

[2] http://www.curry-language.org/

Variablen. Um also diese Fragen zu beantworten, müssen wir das Programm als Curry-Programm markieren (d.h. einfach in eine Datei mit der Endung "`.curry`" schreiben), dieses in ein Curry-System laden und dann Anfragen mit logischen Variablen aufschreiben, wobei letztere (im Gegensatz zu Prolog) explizit als `free` deklariert werden müssen. Z.B. können wie die Kinder von Herbert durch Auswertung des folgenden Ausdrucks berechnen:

```
> vater k == Herbert where k free
```

Als Antwort erhalten wir, ähnlich zu Prolog, Variablenbindungen und, ähnlich zu Haskell, den ausgerechneten Wert des Ausdrucks:

```
{k=Herbert} False
{k=Angelika} False
{k=Hubert} False
{k=Susanne} True
{k=Norbert} True
{k=Andreas} False
```

Wie wir sehen, erhalten wir auch die Lösungen für die Werte `False`, die uns aber eigentlich nicht interessieren. Dies können wir durch die Anwendung der Funktion `solve`, die nur `True`-Werte berechnet, auf diesen Ausdruck erreichen:

```
> solve (vater k == Herbert)  where k free
{k=Susanne} True
{k=Norbert} True
```

Tatsächlich ist `solve` als partielle Funktion wie folgt definiert:

```
solve True = True
```

Somit ist klar, dass bei Verwendung von `solve` nur positive Resultate geliefert werden, an denen man typischerweise auch nur interessiert ist.

Wir können in Curry natürlich auch die anderen in Kapitel 4.1 genannten Anfragen stellen und uns Lösungen berechnen lassen:

```
Welche Großväter hat Andreas?
> solve (grossvater Andreas g)  where g free
{g=Heinz} True
{g=Fritz} True

Welche Enkel hat Heinz?
> solve (grossvater e Heinz)  where e free
{e=Susanne} True
{e=Norbert} True
{e=Andreas} True
```

In Curry sind somit die Eigenschaften von Haskell und Prolog in einer Sprache vereinigt.

Ein Vorteil im Vergleich zu Prolog ist die Möglichkeit, Funktionen nicht als Prädikate zu codieren sondern direkt zu definieren und in geschachtelten Ausdrücken zu verwenden. Z.B. können wir die Fakultätsfunktion wie in Haskell definieren:

```
-- Computes the factorial of a non-negative number.
fac :: Int → Int
fac n = if n == 0 then 1 else n * fac (n - 1)
```

Und natürlich sieht auch unsere bekannte Listenkonkatenation wie in Haskell aus:

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

Im Gegensatz zu Haskell können wir diese allerdings wie in Prolog auch in umgekehrter Richtung durch Verwendung freier Variablen anwenden:

```
> solve (prog ++ ".curry" == "Fac.curry")  where prog free
{prog="Fac"} True
> solve (xs ++ ys == [1,2,3])  where xs,ys free
{xs=[], ys=[1,2,3]} True
{xs=[1], ys=[2,3]} True
{xs=[1,2], ys=[3]} True
{xs=[1,2,3], ys=[]} True
```

Und wir können, wie in Prolog, die Listenkonkatenation in einer Bedingung verwenden, um ganz einfach das letzte Element einer Liste zu berechnen:

```
-- Compute the last element of a list:
last xs | xs == _ ++ [x]
        = x                    where x free
```

Wie man hier sieht, drückt die Bedingung aus, dass das Argument eine Liste sein muss, die man aus dem Ausdruck "`_ ++ [x]`" (durch geeignete Belegung der Variablen `_` und `x`) ableiten kann. In Curry ist es erlaubt, solche Bedingungen durch *funktionale Muster* [1] auszudrücken, womit man eine kompakte und klare Definition von `last` erhält:

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

Mittels funktionaler Muster können wir nicht nur einfach auf das Element am Ende der Liste zugreifen, sondern auch auf beliebige Elemente in der Liste. Z.B. können wir eine Permutation einer Liste berechnen, indem wir ein beliebiges Element an den Anfang einer permutierten Restliste einfügen:

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

Damit können wir Permutationen einer Liste berechnen, welche (wie bei Prolog gezeigt) für verschiedene Zwecke nützlich sind:

```
> perm [1,2,3]
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

Funktionale Muster sind eine sehr elegante Möglichkeit, Pattern Matching beliebig tief in Datenstrukturen durchzuführen, ohne diesen Prozess von Hand zu programmieren. Z.B. kann man damit sehr einfach in XML-Dokumenten suchen (dies ist in [9] im Detail beschrieben). Diese Programmiertechnik wird verwendet, um in die Moduldatenbank des Instituts[3] (die vollständig in Curry implementiert ist) Veranstaltungsdaten automatisch einzupflegen, die aus dem XML-Format des UnivIS der CAU Kiel[4] extrahiert werden.

Wir haben gesehen, dass man jedes funktionale Programm in ein logisches Programm übersetzen kann, in dem man Funktionen durch Prädikate mit einem weiteren Ergebnisargument übersetzt. Daher kann man sich fragen, was es überhaupt für Vorteile hat, funktionale Sprachen um logische Anteile zu erweitern und nicht direkt nur logische Sprachen zu verwenden. Neben der natürlicheren Syntax funktionaler Sprachen (viele Programmieraufgaben sind eigentlich Funktionen) bietet diese Integration auch einen Effizienzvorteil, wenn man Resolution mit Lazy Evaluation kombiniert.

Betrachten wir hierzu einmal die Darstellung natürlicher Zahlen als Peano-Zahlen und die Addition auf Peano-Zahlen in Prolog (vgl. Kapitel 4.3.4):

```
add(o   , Y, Y   ) .
add(s(X), Y, s(Z)) :- add(X,Y,Z).
```

Da Curry wie Haskell eine streng getype Sprache ist, müssen wir in Curry einen Datentyp für Peano-Zahlen einführen und können darauf dann die Addition definieren:

```
data Nat = O | S Nat
 deriving (Eq,Show)

add :: Nat  → Nat  → Nat
add O     n = n
add (S m) n = S (add m n)
```

Nun wollen wir Lösungen für die Gleichung

$$x + y + z = 0$$

berechnen. In der Prolog-Version stellen wir die Anfrage:

```
?- add(X,Y,R), add(R,Z,o).
X = o, Y = o, R = o, Z = o
```

[3] https://mdb.ps.informatik.uni-kiel.de/
[4] http://univis.uni-kiel.de/

Wir erhalten damit zwar eine Lösung, aber wenn wir nach weiteren Lösungen suchen, terminiert das Prolog-System nicht: es probiert immer größere Zahlen für `X` aus, die zu größeren Werten für `R` führen, aber nie zu einem weiteren Ergebnis. Der Suchraum oder SLD-Baum für diese Anfrage ist also unendlich.

Wenn wir dasselbe in Curry ausprobieren, ist der Suchraum dagegen endlich:

```
> solve (add (add x y) z == 0)  where x,y,z free
{x=0, y=0, z=0} True
>
```

Dies liegt an der bedarfsgesteuerten (lazy) Berechnung: nachdem die erste Lösung `x=0` gefunden wurde, werden für `x` weitere Möglichkeiten gesucht, also zunächste `x=S x1`. Da dies aber (unabhängig von `x1`) zu einem Fehlschlag führt, ist es nicht notwendig, weitere Alternativen für den inneren Aufruf `(add x y)` auszuprobieren. Somit ist nach diesen zwei Versuchen alles ausprobiert worden.

Dieses Beispiel zeigt, dass die Kombination von logischen Techniken (Resolution) mit funktionalen Techniken (Lazy Evaluation) deutliche Vorteile bietet, weil hierdurch eine *bedarfsgesteuerte Suche* realisiert wird, welche man manuell (z.B. in imperativen Sprachen) nur mit größerem Aufwand implementieren kann.

Logisch-funktionale Sprachen wie Curry vereinigen einerseits die guten Eigenschaften der funktionalen Programmierung (Funktionen höherer Ordnung, bedarfsgesteuerte Auswertung, strenges polymorphes Typsystem) und der logischen Programmierung (freie Variablen, nichtdeterministische Suche, Unifikation und Constraints). Darüberhinaus bieten sie einen besseren Auswertungsmechanismus als logische Sprachen (bedarfsgesteuerte Suche) und eine höhere Ausdrucksmächtigkeit als funktionale Sprachen (z.B. freie Variablen, funktionale Muster).

Die Entwicklung logisch-funktionaler Programmiersprachen ist ein aktueller Forschungsgegenstand [2]. Die Entwicklung von Curry ist eine Initiative internationaler Forscher in diesem Bereich. Insbesondere ist dies ein Forschungsschwerpunkt der Arbeitsgruppe „Programmiersprachen und Übersetzerkonstruktion“ im Institut für Informatik der CAU Kiel. In dieser Arbeitsgruppe werden verschiedene Implementierungen von Curry entwickelt, wobei eine Implementierung (PAKCS) als Paket in neueren Debian- und Ubuntu-Versionen integriert ist. In zahlreichen Abschlussarbeiten wurde an Implementierungen, Anwendungen und Werkzeugen für Curry gearbeitet. Wer Interesse hat, hieran mitzuarbeiten, kann gerne einmal in der Arbeitsgruppe vorbeischauen!

Literaturverzeichnis

[1] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[2] S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[3] T. Ball and B. Zorn. Teach foundational language principles. *Communications of the ACM*, 58(5):30–31, 2015.

[4] L. Byrd. Understanding the control flow of Prolog programs. In *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.

[7] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[8] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *Proc. IFIP '83*, pages 909–914. North-Holland, 1983.

[9] M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011.

[10] 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.

[11] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[12] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[13] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[14] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[15] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[16] N. Savage. Using functions for easier programming. *Communications of the ACM*, 61(5):29–30, 2018.

[17] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[18] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer LNCS 925, 1995.

Abbildungsverzeichnis

Index