Paradigmen der Programmierung (Teil 1)

Funktionale Programmierung und Logikprogrammierung

Prof. Dr. Erich Ehses

FH Köln
Campus Gummersbach

Wintersemester 2014/2015

Inhaltsverzeichnis

Kapitel 1

Einführung

```
sorted(Xs,Ys):- permutation(Xs,Ys), ordered(Ys).
```
unbekannter Prolog Programmierer

Vielen Informatikstudenten geht es darum, möglichst schnell und gut in die gängigen Methoden der objektorientierten Programmierung einzusteigen. Sie wünschen sich dann weiterführende Vertiefungen in Java.

Es gibt aber in der Informatik und in der Programmierung auch die Notwendigkeit, sich mit grundlegenden Konzepten auseinanderzusetzen, auch wenn diese mitunter aktuell nur eine geringe praktische Bedeutung haben. Diese Konzepte bilden das Grundwissen von Informatikern und liegen oft dem Entwurf von neuen Programmiersprachen zugrunde. Ganz abgesehen davon, muss man in der Informatik immer damit rechnen, dass „vergessene" Ansätze plötzlich aktuell werden. Ganz entscheidend ist, dass auch die moderne Programmiermethodik zunehmend auf deklarativen Techniken aufbaut.

In der Lehrveranstaltung *Paradigmen der Programmierung* sollen alle viele solche Ansätze dargestellt werden, die bei Algorithmen und Programmierung noch nicht angesprochen wurden.

Historisch gesehen, waren Sprachkonzepte oft an einer möglichst optimalen Ausnutzung der Rechnerleistung orientiert. Man kann soweit gehen und sagen, dass sie von der Rechnerarchitektur geformt wurden. Gängige Computer entsprechen auch heute noch weitgehend der *von-Neumann-Architektur*, benannt nach dem Mathematiker, Physiker und Computer-Pionier John von Neumann.[1] Programmiersprachen, die an der Ausnutzung der Rechnerarchitektur orientiert sind, sind nicht zufällig untereinander sehr ähnlich – sie heißen von-Neumann-Sprachen.

Es soll am Rande angemerkt werden, dass die von Neumann-Architektur anfangs keineswegs die günstigste Ausnutzung der Hardware ermöglichte. Noch Anfang der 50er war *Pilot Model ACE*, entwickelt von Alan Turing, der weltschnellste Rechner. Dies erreichte er durch ein hohes Maß von Parallelverarbeitung. Allerdings hatte dies seinen Preis: Die extreme Hardwareorientierung der Rechnerarchitektur machte die Programmierung extrem schwierig. Demgegenüber bot das abstrakte Konzept des von Neumann-Rechners sowohl für Hardware-Entwickler als auch für Programmierer einen einfachen und verständlichen Rahmen, der auch bei der extremen Weiterentwicklung der Hardware weitestgehend gültig blieb.

[1] JVN hat unter anderem das Konzept des speicherprogrammierbaren Computers entwickelt.

Der Prototyp der von Neumann-Sprachen, findet sich in den Sprachen und Konzepten der *prozeduralen Programmierung* wieder. Klassische Beispiele sind die Programmiersprachen C und Pascal.

Der prozedurale Charakter kommt auch in der Entwurfsmethode der *schrittweisen Verfeinerung* zum Ausdruck. Die Entwicklung eines Programms orientiert sich an der Formulierung von Vorgängen und Abläufen.

Die Objektorientierung weicht bereits davon ab. Bei ihr wird die Struktur eines Programms durch seine Schnittstellen und Klassen bestimmt, also durch die Art der Daten und ihre Operationen. Prozedurale Programmierung findet sich aber immer da wieder, wo Abläufe dargestellt werden, nämlich in Methoden und Klassenfunktionen. Insbesondere die Klassenfunktionen stellen in Java ein Relikt des prozeduralen Paradigmas dar.

Die Erfahrung mit Objektorientierung hat zu der Erkenntnis geführt, dass die Effizienz einer Programmiersprache nicht das allein ausschlaggebende Kriterium sein darf.

Eine andere historische Erfahrung besagt auch, dass die beste Hardwareausnutzung nicht dadurch zu erreichen ist, dass der Mensch die Computerabläufe in jedem Detail bestimmt. Bei den zunehmend komplexer werdenden Architekturen sind ihm automatisierte Mechanismen überlegen. Diese Optimierung ist aber oft nur möglich, wenn Programme hinreichend abstrakt formuliert werden.

1.1 Unterschiedliche Paradigmen

Zunächst einmal müssen wir uns klar machen, dass es verschiedene *Paradigmen der Programmierung* gibt.

Definition:

> *Unter einem **Paradigma** verstehen wir ein in sich geschlossenes System von Methoden und Grundauffassungen. Ein Paradigma stellt eine bestimmte Sicht auf die Welt dar. Auch dann, wenn es für einen Bereich mehrere gleich gute Paradigmata gibt, so sieht es innerhalb des Denksystems eines einzigen Paradigmas doch oft so aus, als gäbe es kein anderes das gleich gut wäre.*

Der Begriff *Paradigma* hat viele Kennzeichen eines schlecht definierten Modebegriffs. Hier sollen etwas konkreter unterschiedliche Programmierparadigmen dargestellt werden.

Prozedurale Programmierung Ein Programm ist eine Folge von Befehlen, die auf einem passiven Speicher operieren. Programmentwurf ist gleich Algorithmenentwurf. Prozedurale Programmierung ist die gängige Methode bei der Implementierung algorithmischer Verfahren. Ein typisches Kennzeichen der prozeduralen Programmierung ist die Menge der (globalen) Variablen, die den aktuellen Zustand des Programms darstellt.,

Objektorientierte Programmierung Ein Programm ist eine Menge von interagierenden Objekten. Der Programmentwurf ist ein Entwurf von Klassen und Schnittstellen. Der Programmablauf ist nicht mehr gut zu erkennen. Objekte kapseln die Variablen. Die Variableninhalte der Objekte definieren den Zustand des Programms. In einem Programm werden die an sich

passiven Methoden der Objekte in dem sequentiellen Programmablauf ausgeführt.

Aspektorientierte Programmierung ist eine Erweiterung der objektorientierten Programmierung um die modulare Formulierung von Aspekten die quer zur Klassenhierarchie liegen (cross cutting concerns). Aspektorientierte Programmierung ist nicht wirklich ein umfassendes Paradigma. Sie versteht sich als Ergänzung der als unzureichend verstandenen Objektorientierung.

Nebenläufige Programmierung Ein Programm enthält mehrere gleichzeitige Abläufe. Die Reihenfolge der Befehlsausführung ist nicht vollständig definiert.

Funktionale Programmierung Ein Programm ist eine Funktion, die die Eingabe auf die Ausgabe abbildet. Die Programmierung besteht in der Beschreibung des funktionalen Zusammenhangs. Die funktionale Programmierung besitzt eine formale Fundierung in dem λ-Kalkül. Ein Ablauf wird nicht vorgegeben. Funktionale Programmierung kennt keinen Zustand. Funktionale Programme können problemlos nebenläufig abgearbeitet werden.

Logikprogrammierung Ein Programm ist eine Menge von Fakten und Regeln. Die Ausführung eines Programms besteht in der Beantwortung einer Frage. Logikprogrammierung basiert auf dem Kalkül der Prädikatenlogik. Sie begünstigt deklarative Programmierstile. Sie bietet hohe Flexibilität bei *intelligenter* Lösungssuche.

Die dargestellten Programmierstile lassen sich grob in zwei Bereiche einteilen:

- In der **imperativen Programmierung** werden Befehle zur Steuerung der Berechnung formuliert. Die prozedurale Programmierung ist imperativ.
- In der **deklarativen Programmierung** werden Aussagen über die Programmobjekte formuliert. Der Ablauf steht im Hintergrund. Funktionale und Logikprogrammierung sind deklarativ.

Man kann argumentieren, dass der imperativ-prozedurale Stil, auf einem vergleichsweise niedrigen Abstraktionsniveau angesiedelt ist. Vergleichen Sie die beiden folgenden Programmabschnitte, die die Summe der Quadrate der ungeraden Zahlen eines Arrays berechnen.

Prozedural in Java:

```
public static double sumOddSquares(double[] array) {
    int sum = 0.0;
    for (int i = 0; i < array.length; i++) {
        if (array[i] % 2 == 1)
            sum += array[i] * array[i]
    }
    return sum;
}
```

und funktional in Scala:

```
def sumOddSquares(array: Array[Double]): Double =
    array.filter(x => x % 2 == 1).map(x => x * x).sum
```

Vielleicht werden Sie sagen, dass der Vergleich der *Länge* der beiden Formulierungen unfair ist. Durch ein höheres Maß an Modularisierung erreicht man immer kürzere Programme. Das ist aber hier genau der Punkt. Die funktionale Programmierung unterstützt die Modularisierung viel besser, als das prozedural möglich ist.

Der Ausdruck des funktionalen Beispiels besteht aus vier Elementen: den Daten (`array`) einer Filteroperation, die die ungeraden Zahlen heraussucht, der Quadrierung der Zahlen und schließlich der Summenbildung. Das Programm bleibt durchgängig auf diesem hohen Abstraktionsniveau. Die Grundbestandteile der Berechnung sind deutlich erkennbar. Die knappe Formulierung ist möglich, weil man in Scala Funktionen durch Literale definieren kann, und sie dann an andere Funktionen weiter geben kann.

Dagegen können wird die prozedurale Form nicht einfach in Funktionsbausteine zerlegen. Um sie zu verstehen, müssen wir den Ablauf auf niedrigster Ebene – Element für Element – nachvollziehen. Während wir in Java eine Folge von *Anweisungen* haben, haben wir in Scala einen einzigen *Ausdruck*.

Programmierung von Nebenläufigkeit erfordert ebenfalls ein Abgehen von dem rein imperativen Denken, da sich nebenläufige Aktionen nicht exakt vorher bestimmen lassen. Objektorientierung ist ein Konzept, das imperative und deklarative Gesichtspunkte vereint. Grundsätzlich kann man sagen, dass sich komplexe Programme besser deklarativ verstehen lassen. Das imperative Denken ist am besten für die Steuerung von Abläufen geeignet.

1.2 Typsystem

Daneben gibt es noch ein weiteres Unterscheidungsmerkmal für Programmiersprachen, das mehr mit der Formulierung als mit der Ausführung eines Programms zu tun hat.

- Bei **dynamisch getypten** Programmiersprachen findet die Typprüfung ausschließlich zur Laufzeit statt. Variable, Funktionsparameter und Ergebnisse sind in diesem Konzept nicht mit einer Typangabe versehen. Dies ermöglicht ein hohes Maß an Polymorphie.

- Bei **statisch getypten** Sprachen findet die Typprüfung ganz (z.B. Pascal) oder teilweise (z.B. Java) durch den Compiler statt. Dies ermöglicht frühzeitige Fehlermeldungen und effiziente Codegenerierung. Auf der anderen Seite werden geringere Fexibilität und komplexere Typregeln in Kauf genommen.

- Systeme mit **Typinferenz** (automatische Herleitung des Datentyps) sind im Kern statisch getypt. Sie erleichtern aber die Programmierung, indem der Compiler wenn möglich den Typ einer Variablen selbst bestimmt. Wie die Beispiele aus dem Bereich der funktionalen Programmierung zeigen, kann dies die Lesbarkeit eines Programms erheblich verbessern.

- Bei **schwach getypten** Sprachen, wie C, findet keine vollständige Typprüfung statt. Man erreicht hohe Flexibilität und Effizienz für den Preis der Unsicherheit.

Grundsätzlich lässt sich diese Unterscheidung mit allen Programmierparadigmen verbinden. Es ist aber so, dass Systeme für höhere Programmierkonzepte (Objektorientierung, Logikprogrammierung und funktionale Programmierung) von Anfang an die Typsicherheit garantierten. Dabei stand zunächst in allen Bereichen die dynamische Typprüfung im Vordergrund.

Anmerkung:

Java ist in mancher Hinsicht ein Zwitter. Dies gilt auch für die Typprüfung. Die Sprachentwickler favorisieren die statische Prüfung wie das auch in dem Konzept der generischen Typen zum Ausdruck kommt.

Alle *höheren* Programmiersprachen verfügen über Konzepte der automatischen und sicheren Speicherverwaltung. Das war von Anfang an so. Zeigerarithmetik ist nicht bekannt.

1.3 Funktionsbindung

In Algorithmen und Programmierung 2 habe ich betont, dass die späte Bindung von Methoden den entscheidenden Unterschied zwischen objektorientierter Programmierung und prozeduraler Programmierung mit früher Bindung von Funktionen ausmacht.

Bei funktionaler Programmierung und Logikprogrammierung, lernen Sie mit dem „Pattern-Matching“ einen weiteren Mechanismus kennen. Vergleichen wir die Formulierung für die Berechnung des Wertes eines Abstrakten Syntaxbaums:

```
double eval(Node* t) {
    switch (t->tag) {
    case PLUS:
        return eval(t->left) + eval(t->right);
    case MINUS:
        return eval(t->left) + eval(t->right);
        ...
    }
}
```

Mit dem Aufruf von `eval` ist klar, wo das Programm „hin springt“. Die Auswahl zwischen den verschiedenen Operationen muss durch einen programmierten Vergleich von Kennungen getroffen werden.

Dagegen die objektorientierte Fassung:

```
class PlusNode {
    public double eval() {
        return left.eval() + right.eval();
    }
    ..
}
```

Jede Operation ist durch eine eigene Klasse beschrieben. Eine explizite Fallunterscheidung ist her nicht nötig. Die Fallunterscheidung erfolgt dadurch, dass die späte Bindung, anhand des angesprochenen Objekts (`left`, `right`) die passende Methode auswählt (*method dispatching*).

Schließlich das Pattern-Matching in Prolog und Scala:

```
eval(X + Y, Z):-
    eval(X, X1),
    eval(Y, Y1),
    Z is X1 + Y1.

eval(X - Y, Z):-
    eval(X, X1),
    eval(Y, Y1),
    Z is X1 - Y1.
...
```

Die funktionale Formulierung in Scala:

```
def eval(t: Tree): Double = match {
    case Plus(left, right) => eval(left) + eval(right)
    case Minus(left, right) => eval(left) - eval(right)
    ..
}
```

Die Prolog- und Scala-Variante kann auch durch das Konzept des „algebraischen Datentyp“'s beschrieben werden. Algebraische Typen erlauben eine durch eine Typkonstrukte die sehr reguläre Konstruktion und gleichzeitig auch Dekonstruktion der Daten. Die algebraische Sichtweise ist auch bei der Programmierung hilfreich, da sie hilft, alle Spezialfälle zu betrachten.

1.4 Überblick

In dem ersten Teil der Vorlesung werden die funktionale Programmierung und die Logikprogrammierung vorgestellt. Ich werde versuchen, Ihnen die Grundideen dieser Paradigmen zu vermitteln und auch versuchen, Brücken zu Java zu schlagen.

Die Logikprogrammierung basiert auf dem Kalkül der Prädikatenlogik. Deren Konzepte sollen aber nur ganz kurz angesprochen werden, soweit es für das Grundverständnis nötig ist. Die Logikprogrammierung werde ich anhand der Programmiersprache *Prolog* vorstellen

Bei der Diskussion der funktionalen Programmierung werde ich die Anwendbarkeit funktionaler Techniken betonen. Sie lassen sich oft auch in andere Programmiersprachen übertragen. Selbst die über die Grundtechniken hinausgehenden Muster der Programmierung mit Funktionen höherer Ordnung, dringen allmählich in die Java-Welt ein (Java 8). Für die funktionale Programmierung verwende ich die Programmiersprache *Scala*. Scala ist zwar keine rein-funktionale Sprache (wie Haskell), bietet aber dafür den Vorteil, dass das Erlernen wegen der Nähe zu Java etwas ähnlich sein dürfte.

Damit ist der erste Teil der Vorlesung beschrieben. Der zweite Teil wird von Prof. Dr. Kohls gestaltet.

Kapitel 2

Die Programmiersprache Prolog

Das Kunstwort Prolog steht für **Pro**grammierung in **Log**ik. Damit ist gesagt, dass Prolog eine Brücke zwischen den Konzepten von Programmiersprachen und Logik schlägt. Prolog gehört damit in den allgemeineren Kontext der *Logikprogrammierung*.

Das Grundprinzip der Logikprogrammierung besteht darin, dass vorrangig deklarativ die nötigen Zusammenhänge beschrieben werden. Die Steuerung des Ablaufs tritt in den Hintergrund. Die Ausführungsreihenfolge beeinflusst eventuell erheblich die Effizienz eines Programms; sie sollte aber keinen Einfluss auf die Bedeutung und die Korrektheit des Programms haben.

Prolog, als wichtigster Vertreter der Logikprogrammierung, ist in den 70er Jahren in Frankreich entstanden. Nachdem es in den 80ern einen richtigen Boom erlebte, ist es seither wieder in den Hintergrund getreten. Nach wie vor ist Prolog aber *das* Paradebeispiel für eine Programmiersprache, die auf der Idee der Logikprogrammierung aufbaut. Es ist Gegenstand und Hilfsmittel für Forschung im Bereich der Künstlichen Intelligenz und gehört allgemein zum Standardumfang der Informatikausbildung. Die hinter Prolog stehenden Ideen der logikbasierten Programmierung haben praktische Anwendung in Form von Wissensbasierten Systemen und von Expertensystemen gefunden.

Es gibt eine Vielzahl von kommerziell und frei erhältlichen Prolog-Systemen. Auch wenn diese sich in einzelnen Details unterscheiden, so basieren doch fast alle auf einem gemeinsamen Kern (Edinburgh-Prolog). Der Vorlesung liegt das frei erhältliche SWI-Prolog zugrunde. Es zeichnet sich durch Vollständigkeit und insbesondere durch leichte Bedienbarkeit aus. Es ist sowohl für Windows- als auch für Unix-Systeme verfügbar.

Definition:

*Ein Prolog-Programm besteht aus **Fakten** und **Regeln**. Die interaktive Anwendung eines Prolog-Programms besteht in der Beantwortung von **Anfragen**.*

Im Unterschied zu der üblichen Ausführung durch ein ausführbares Programm, werden wir Prolog-Programme durch die Formulierung von Anfragen innerhalb der interaktiven Umgebung testen.

2.1 Ein Überblick über die Verwendung von Prolog

2.1.1 Eine Prolog-Programmdatei

Die Datei familie.pl enthält Fakten und Regeln über Verwandschaftsbeziehungen. Ihr Inhalt sieht so aus:

```
 1 /* familie.pl
 2    Erstes Beispiel zur Struktur eines Prolog-Programms
 3 */
 4
 5 /**
 6  * Fakten
 7  * ======
 8  */
 9 % elternteil_von_kind(Elternteil, Kind)
10 % 'Elternteil' ist ein Elternteil von 'Kind'
11
12   elternteil_von_kind('Hans', 'Karin').
13   elternteil_von_kind('Carmen', 'Karin').
14   elternteil_von_kind('Carmen', 'Bert').
15   elternteil_von_kind('Lisa', 'Carmen').
16
17 % geschlecht(Person, Geschlecht)
18 % Das Geschlecht von 'Person' ist 'Geschlecht' = m/f.
19
20   geschlecht('Hans', m).
21   geschlecht('Karin', f).
22   geschlecht('Carmen', f).
23   geschlecht('Bert', m).
24   geschlecht('Lisa', f).
25
26 /**
27  * Regeln
28  * ======
29  */
30 % mutter(Mutter, Kind)
31 % 'Mutter' ist die Mutter von 'Kind'
32
33   mutter(Mutter, Kind):-
34     elternteil_von_kind(Mutter, Kind),
35     geschlecht(Mutter, f).
36
37 % vorfahre(Vorfahre, Nachkomme):-
38 % 'Vorfahre' ist Vorfahre von 'Nachkomme'
39
40   vorfahre(Vorfahre, Nachkomme):-
41     elternteil_von_kind(Vorfahre, Nachkomme).
42
43   vorfahre(Vorfahre, Nachkomme):-
44     elternteil_von_kind(Elternteil, Nachkomme),
45     vorfahre(Vorfahre, Elternteil).
```

Einiges an dieser Datei ist fast selbsterklärend. So die Kommentare `/* ... */` und die Zeilenkommentare `%` (entspricht den Kommentaren `//`). Auch die Fakten sind leicht zu verstehen. eine Hilfestellung bieten dabei die zugehörigen Kommentare. Natürlich gehören die Zeilennummern nicht zu dem Programm! Die Prolog-Anweisung:

```
13   elternteil_von_kind('Carmen', 'Karin').
```

heißt nichts weiter als: „Carmen ist ein Elternteil von dem Kind Karin“. Mit etwas großzügiger Auslegung der Grammatik kann man den Satz auch schreiben als „Carmen elternteil_von_kind Karin“. In diesem Satz spielt „Carmen“ die Rolle des *Subjekts*, „Karin“ die Rolle des *Objekts* und „elternteil_von_kind“ die Rolle des *Prädikats*.

Man kann Prädikatsnamen natürlich immer beliebig ausführlich wählen. Vielleicht wäre `ist_ein_Elternteil_von_einem_Kind` sprachlich ja am genauesten. Da solche Ausdrücke aber sehr lang werden können, verwende ich in der Folge eher kurze Namen, die den Zusammenhang durch einen kurzen Begriff ausdrücken.

Prolog-Aussagen konzentrieren sich also auf die *Prädikate* der natürlichen Sprache. Entsprechend heißt die zugrunde liegende Logik auch *Prädikatenlogik*.

Definition:

*In der Prolog-Sprechweise ist eine einzelne Aussage (Fakt, Regel oder Anfrage) eine **Klausel**. Die Menge gleichnamiger Klauseln heißt **Prädikat** (manchmal auch **Prozedur**).*

Der besseren Lesbarkeit halber, sollten die Klauseln eines Prädkats stets zusammenhängend beschrieben sein. Es ist aber nicht nötig, Fakten und Regeln zu trennen.

Wie im Alltagsgebrauch dienen auch in Prolog Regeln dazu, dass man sich nicht so viele Details merken bzw. aufschreiben muss. Innerhalb einer Datenbasis zu Familien kann man die Mutter-Kind-Beziehung genauso als ein Fakt auffassen und beschreiben wie die Eltern-Kind-Beziehung. Es gibt keinen logisch zwingenden Grund hier einen Unterschied zu machen. Nur, wenn die eine Art von Beziehung bereits bekannt ist (einschließlich der nötigen Information zum Geschlecht der beteiligten Personen), dann kann man sich die Arbeit sparen, alle Mütter nochmals aufzuzählen, indem man einfach eine Regel angibt, was der Begriff Mutter bedeutet:

> *Wenn M Elternteil von K ist, und das Geschlecht von M gleich f ist, dann ist M Mutter von K.*

Diesen Satz habe ich bewusst etwas formal in die Form einer Implikation gekleidet. In der formalen Sprache der Prädikatenlogik sieht das dann so aus:

$$\forall_{M,K}(\text{elternteil\_von\_kind}(M,K) \wedge \text{geschlecht}(M,f) \Rightarrow \text{mutter}(M,K))$$

Natürlich hat diese formale Betrachtung von Regeln etwas mit den logischen Grundlagen von Prolog zu tun. Doch davon später. Umgekehrt kann man eine Regel aber auch immer als eine Definition lesen:

> *M ist (mindestens) dann Mutter des Kindes K, wenn M Elternteil von K ist und das Geschlecht von M gleich f ist.*

An den Beispielen erkennen Sie auch, dass Prolog nur ganz wenige syntaktische Formen kennt. Sie sehen, dass (Zeilen 33–35) die wenn-dann-Beziehung durch

das Zeichen `:-` und die und-Beziehung durch ein Komma ausgedrückt werden. Prolog Fakten und Regeln werden immer durch einen Punkt abgeschlossen. Ein Letztes: *Zwischen Prädikatsnamen und öffnender Klammer darf kein Leerzeichen stehen!*

In den Zeilen 37–45 sehen Sie in `vorfahre` eine etwas kompliziertere Regel. Es ist nämlich nicht ganz einfach zu erklären, wer ein Vorfahre bzw. ein Nachkomme ist:

> *Ein Elternteil ist Vorfahre seiner Kinder. Die Vorfahren eines Elternteils einer Person sind ebenfalls deren Vorfahren.*

Genauso wie diese Erklärung aus zwei Sätzen besteht, benötigen wir in Prolog zwei Regeln. Die erste Regel erklärt den einfachen Sachverhalt der Eltern-Kind-Beziehung, die zweite Regel den allgemeinen Fall der (rekursiven) Erklärung von Vorfahren.

2.1.2 Eine interaktive Prolog-Sitzung

Nun zur Anwendung dieses Programms. Prolog stellt eine interaktive Umgebung zur Verfügung, in der wir Programme laden und ausführen können. Die Details hängen von dem verwendeten Prolog System und auch von dem Betriebssystem ab. Unter SWI-Prolog und Unix könnte unser Dialog wie folgt aussehen:

```
/home/erich> swipl
...
For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- protocol(p1).
true
2 ?- consult(familie).
familie compiled, 0.00 sec, 2,124 bytes.
true
```

Wie Sie sehen, wird Prolog unter Unix einfach als `swipl` aufgerufen.[1] Anschließend meldet es sich mit einer kurzen Versionsmeldung und dann mit dem interaktiven Prompt:

```
    1 ?-
```

Die 1 ist einfach eine fortlaufende Nummer. Das `?-` drückt aus, das das System jetzt bereit ist, eine Anfrage entgegen zu nehmen.

Gleich die erste Anfrage stellt eine Ausnahme dar. Hier handelt es sich nicht um eine Frage im üblichen Sinne, sondern um den Aufruf einer eingebauten Systemfunktion. Durch `protocol(p1)` wird erreicht, dass der gesamte interaktive Dialog in der Datei p1 gespeichert wird. Den Bezug zur Logik erkennen Sie hier nur daran, dass das Prolog-System schließlich mit „true“ antwortet, was als Antwort auf eine Frage aufgefasst werden kann.

Genauso wird auch in der zweiten Anfrage durch `consult(familie)` ein Systemprädikat aufgerufen. `consult` hat zur Wirkung, dass die Datei familie.pl

[1] Die genaue Form des Aufrufs hängt vom Unix-System ab. Je nach Gusto kann das Kommando auch anders lauten.

(`pl` ist die Standardendung für Prolog-Programme) eingelesen und in eine interne Form übersetzt wird. Ab jetzt können wir Fragen zu den diversen Verwandschaftsbeziehungen stellen.

```
3 ?- geschlecht('Carmen', f).
true
4 ?- geschlecht('Carmen', m).
false
5 ?- geschlecht('Carmen', weiblich).
false
```

Zunächst wollen wir wissen, ob Carmen männlich oder weiblich ist. Wie Sie sehen, gibt Prolog meist die richtige Antwort. Aber obwohl Carmen sicher weiblich ist, antwortet das System mit `false`. Die Begründung dafür ist ganz einfach: „Alles was Prolog nicht in seiner Datenbasis findet, wird als falsch angesehen".

Weiter sehen Sie, dass die Anfragen (fast) genauso aussehen, wie die entsprechenden Fakten unseres Programms. Allerdings werden Sie in der interaktiven Umgebung anders interpretiert. Eben nicht als Feststellungen sondern als Fragen. Innerhalb einer rein textuellen Darstellung bringt man diese Unterscheidung durch das vorangestellte `?-` zum Ausdruck:

```
geschlecht('Carmen', m).      % Fakt
?- geschlecht('Carmen', m).   % Frage
```

Ja-Nein-Fragen sind letztlich etwas langweilig. Um interessantere Fragen stellen zu können, brauchen wir Platzhalter für die Antwort, nämlich Variable. Der Dialog könnte so fortgesetzt werden:

```
6 ?- geschlecht('Carmen', X).
X = f
true
7 ?- geschlecht(X, m).
X = 'Hans' ;
X = 'Bert' ;
false
```

Eine solche Frage lautet in Umgangssprache „Welches Geschlecht hat Carmen?" oder „Wer ist männlich?". Bei der zweiten Frage ist das Besondere, dass sie mehrere richtige Antworten hat. SWI-Prolog liefert zunächst nur die erstbeste Antwort und überlässt dem Benutzer die Entscheidung, wie es weitergeht. Tippt dieser ein Return ein, so ist die Frage abgeschlossen und Prolog meldet sich mit `true` und mit einem neuen Eingabe-Prompt. Tippt der Benutzer jedoch ein Semikolon, so wird die nächste Antwort ausgegeben. Dies wird solange fortgesetzt, bis der Benutzer genug hat (Return) oder bis es keine weitere Antwort mehr gibt.

Der Aufruf von Regeln unterscheidet sich überhaupt nicht von dem Aufruf von Fakten:

```
24 8 ?- vorfahre_von_nachkomme(V, N).
25 V = 'Hans'
26 N = 'Karin' ;
27
28 V = 'Carmen'
```

```
29 N = 'Karin' ;
30
31 V = 'Carmen'
32 N = 'Bert'
33 true
```

Hier haben wir jetzt einige Vorfahren/Nachkommen-Paare kennengelernt. Der interne Ablauf der Anwendung einer Regel ist schon etwas komplexer als die einfache Beantwortung einer Frage. Das soll uns hier aber nicht kümmern.

Wie jeder weiß, der einmal eine Internet-Suchmaschine benutzt hat, können Anfragen, die viele mögliche Lösungen haben, zu einer praktisch unüberschaubaren Fülle von Antworten führen. In einem solchen Fall ist es nötig, die Frage genauer zu formulieren und so die Menge der möglichen Antworten einzuschränken. Anstatt wie oben nach allen möglichen Vorfahren und Nachkommen zu fragen, könnten wir die Frage auf die männlichen Vorfahren von weiblichen Nachkommen einschränken. Halbverbal können wir das so ausdrücken:

> *Ich suche ein V und ein N, so dass V Vorfahre von N ist, V männlich ist und N weiblich ist.*

In der Besprechung der Programm-Datei haben Sie gesehen, dass in Prolog *und* durch ein Komma ausgedrückt wird. Dies gilt nicht nur in Regeln sondern auch in Fragen:

```
34 9 ?- vorfahre_von_nachkomme(V, N), geschlecht(V,m),
        geschlecht(N,f).
35 V = 'Hans'
36 N = 'Karin' ;
37 false
38 13 ?- halt.
39 /home/erich>
```

Nachdem wir wissen, dass Hans der einzige männliche Vorfahr einer weiblichen Nachkomme ist, können wir unsere Prolog-Sitzung durch Aufruf des Systemprädikats `halt` beenden.

2.2 Die Prolog-Syntax

An dem gerade besprochenen Beispiel haben Sie bereits fast alle syntaktischen Elemente von Prolog kennengelernt. Diese Syntax-Regeln sollen hier noch einmal etwas vollständiger zusammengefasst werden. Zunächst sind dies die *lexikalischen Regeln*, die festlegen, welche Arten von Grundelemente (Zahlen, Namen etc.) es in Prolog gibt. Danach werden die eigentlichen *Syntax-Regeln* kurz beschrieben.

2.2.1 Lexikalische Grundelemente

Prolog kennt einige wichtige lexikalische Grundbegriffe, die durch ihre Schreibweise eindeutig gekennzeichnet sind

Atom

Jede Zeichenfolge, die mit einem Kleinbuchstaben beginnt, der von einer Folge von Buchstaben, Ziffern oder Unterstrich gefolgt ist bezeichnet ein Atom. Ebenso ist jede Zeichenfolge, die in einfache Hochkommata eingeschlossen ist, ein Atom.

Beispiele für Atome:

```
'Carmen'
carmen
mutter_von_kind
x12_und_y16
'Ein ganz beliebiger Text'
```

Atome dienen in Prolog in erster Linie als symbolische Bezeichner. Sie benennen Prädikate oder dienen als Konstanten, wie `m` für männlich oder `'Carmen'` für den Namen Carmen. Da in Prolog Zeichenketten (Strings) eine nur untergeordnete Rolle spielen, werden manchmal auch bloße Ausgabetexte durch Atome kodiert.

Zahlen

Prolog kennt die gleichen Zahlenkonventionen wie andere Programmiersprachen. SWI-Prolog unterstützt sowohl ganze wie auch Gleitkommazahlen. Beispiele:

```
17
1.45
1.2e-5
```

Variable

Variable werden in Prolog durch Wörter (Folge von Buchstaben, Ziffern, Unterstrich), die mit einem Großbuchstaben oder mit einem Unterstrich beginnen, ausgedrückt. Beispiele:

```
X
X1
Vorfahre
Das_wuesste_ich_gerne
_unbekannte
_
```

Vorsicht! Das was Sie bisher über Variable wussten, gilt hier nicht mehr! In Prolog haben Variable eine ganz andere Bedeutung als in anderen Programmiersprachen.

Eine Prolog Variable ist ein Platzhalter und ein Name für eine Unbekannte. In Anfragen steht die Variable für die gesuchte Größe:

```
% fuer welches M gilt mutter(M, 'Carmen'):
?- mutter(M, 'Carmen').
```

Etwas anders ist die Lesart bei Regeln:

```
% fuer alle M gilt: aus mensch(M) folgt sterblich(M)
% (alle Menschen sind sterblich)
sterblich(M):- mensch(M).
```

Es ist in Prolog ein Fehler, einer Variable nacheinander verschiedene Werte zuzuweisen:

```
?- X=1, X=5.
false

?- X = 5, X is X + 1.   % is "rechnet"
false
```

Sie können diese Antworten verstehen, wenn Sie schrittweise vorgehen:

```
?- X=1, X=5.
   %% X=1
   %% 1=5    diese Gleichung ist nie erfuellt!
false

?- X = 5, X is X + 1.
   %% X=5
   %% 5 is 5 + 1
   %% 5=6    auch Unsinn!
false
```

Das Problem liegt nicht bei Prolog sondern darin, dass prozedurale Sprache den Begriff „Variable“ nicht im üblichen Sinn verwenden.

Definition:

*Prozedurale Programmiersprachen bezeichnen mit **Variable** den Namen einer Speicherzelle. Variablennamen stehen entweder für den Inhalt der Speicherzelle (*R-Value*) oder für die Speicheradresse, deren Inhalt geändert werden soll (*L-Value*).*

Eine Prolog-Variable ist der Platzhalter für einen Wert. Es gibt sie in zwei Rollen: Eine Variable kann noch *ungebunden* sein, dann hat sie noch keinen Wert, oder aber sie ist *gebunden*, dann hat sie einen festen unveränderlichen Wert.

Je nachdem wo eine Variable innerhalb eines Programms steht, hat sie entweder die Bedeutung:

- *X* gilt für *alle* möglichen Werte:

  ```
  % alle Menschen sind sterblich
  sterblich(X):- mensch(X).
  ```

- *X* steht für *einen* möglichen Wert:

  ```
  % gibt es einen Menschen -- und wer ist das?
  ?- mensch(X).
  ```

- manche (insbesondere vordefinierte) Prädikate verlangen, dass eine Variable X für einen konkreten Werte steht:

```
% wie lautet die Summe der (bekannten) Zahlen X und Y?
summe(X, Y, Z):- Z is X - Y.
```

Die Tatsache, dass eine Variable an einen Wert gebunden ist, ergibt sich oft schon direkt aus dem Ablauf. Sie lässt sich in Prolog aber auch durch Metaprädikate (wie `var`, `nonvar`, usw.) überprüfen. Das folgende Beispiel berechnet die unbekannte Zahl innerhalb einer Summenbeziehung:

```
summe(X, Y, Z):-
  var(Z),!,
  Z is X + Y.
summe(X, Y,  Z):-
  var(X),!,
  X is Z - Y.
summe(X, Y, Z):-
  var(Y),!,
  Y is Z - X.
summe(X, Y, Z):-
  X + Y =:= Z.
```

Das Programm überlässt dem Prolog-System die Fehlermeldung, wenn nicht genügend Werte bekannt sind, oder falsche Datentypen auftreten. Das Ausrufezeichen (ausgesprochen *cut*) stellt eine Steuerung des Ablaufs dar. Es drückt aus, dass die weiteren Alternativregeln nicht anzuwenden sind. Das Zeichen `=:=` ist ein eingebautes Prädikat, das die numerische Gleichheit zweier Ausdrücke bestimmt (dabei dürfen keine Variablen auftreten).

Der Unterstrich _ steht für eine anonyme Variable. Jedes Vorkommen des Unterstrichs steht für einen anderen möglichen Wert. Ansonsten stehen Namen, die mit einem Unterstrich beginnen, immer für eine Variable.

Sonderzeichen

Alles, was nicht Zahl, Variable, String oder Atom ist, ist in Prolog ein Sonderzeichen. Sie kennen aus der Diskussion der Beispiele bereits:

Klammern zum Darstellen von Prädikaten. Klammern werden aber auch in arithmetischen und in formalen Ausdrücken benutzt.

Punkt zum Abschluss einer Prolog-Klausel.

`:-` zur Trennung von Kopf und Körper einer Regel.

`?-` zur Kennzeichnung einer Anfrage.

Komma zur Formulierung von Und-Verknüpfungen.

Damit kennen Sie auch schon alle unbedingt nötigen Operatoren. Daneben gibt es (natürlich) noch die üblichen mathematischen Verknüpfungen und verschiedene Vergleichsoperationen (und ein paar wenige zusätzliche Prolog-Sonderzeichen).

String

Strings sind Zeichenketten, die in doppelte Hochkommata eingeschlossen sind (wie in Java). Sie werden zur Darstellung von veränderlichen Texten verwendet. Da wir uns hier nicht mit Textverarbeitung befassen und da wir mit Atomen einen ausreichenden Ersatz haben, werden wir Strings nicht verwenden.

2.2.2 Die syntaktischen Strukturen von Prolog

Ein Prolog-Programm ist eine Menge von Fakten und Regeln. Die Anwendung eines Prolog-Programms besteht aus dem Aufruf einer Anfrage.

Die Syntax von Prolog legt fest, wie Fakten, Regeln und Anfragen dargestellt werden.

Klausel ::= *Fakt*
| *Regel*
| *Anfrage*

Fakt ::= *Literal* `.`
Regel ::= *Kopfliteral* `:-` *Literalliste* `.`
Anfrage::= `?-` *Literalliste* `.`

Das grundlegende Element von Prolog sind Aussagen, die durch *Literale*[2] ausgedrückt werden. Sie entsprechen den atomaren Aussagen der Logik. Syntaktisch gesehen sind es Atome (Namen von Aussagen) oder Atome, die mit einer Liste von Argumenten versehen sind.

Literal ::= *Atom*
| *Atom* `(` *Termliste* `)`

Term ::= *Atom*
| *Atom* `(` *Termliste* `)`
| *Zahl*
| *Variable*
| *String*
| *Ausdruck*
| *Liste*

Die Anzahl der Parameter eines Literals heißt *Stelligkeit*. In Prolog ist die Stelligkeit genauso wie der Name charakteristisch für das Literal. Oft wird auch die Stelligkeit zusammen mit dem Namen angegeben. `mutter(M, K)` hat zwei Parameter. Dies wird in Prolog dann durch die Schreibweise `mutter/2` ausgedrückt.

[2]Das ist etwas vereinfacht. In der Logik schließt man die eventuelle Negation in den Begriff Literal ein und unterscheidet positive (nicht negierte) und negative (negierte) Literale.

Zunächst einfach ein paar Beispiele, die die Syntax erläutern. Einzelheiten werden später besprochen:

```
elternteil_von_kind('Hans', 'Karin')  % elternteil_von_kind/2
                                      % Parameter sind Atome

elternteil_von_kind(X, 'Karin')  % X ist eine Variable
alter('Hans', 22)                % 22 ist Zahl
aequivalent(X+Y, Y+X)            % X+Y ist ein Ausdruck
member(X, [1,2,3])               % [1,2,3] ist eine Liste
wert(sin(30), 0.5)               % sin(30) ist ein komplexer
                                 %   Term (Struktur)
```

Bei der Logik geht es nicht primär um die Berechnung von Formeln oder das Hervorrufen äußerer Effekte wie Ausgabe oder Bildschirmaufbau. Prolog unterstützt aber die übliche Syntax arithmetischer Ausdrücke. Zusammen mit dem unten zu besprechenden eingebauten `is`-Prädikat lassen sich auch Zahlenwerte berechnen. Zunächst und in erster Linie sind logische Formeln aber einfach nur Formeln.

Diese wird durch das folgende Sitzungsprotokoll verdeutlicht:

```
1 ?- 3 + 5 = 5 + 3.
false.

3 ?- 3 + 5 = X + Y.
X = 3,
Y = 5.

4 ?- 3 + 4 * 5 = X + Y.
X = 3,
Y = 4 * 5.

5 ?- 3 + 4 * 5 = X * Y.
false.
```

Das Gleichheitszeichen = bewirkt eine Unifikation (siehe nächster Abschnitt) der Formeln der rechten und der linken Seite. Diese Art der Mustererkennung lässt sich zur eleganten Umwandlung symbolischer Ausdrücke verwenden. Die beiden letzten Beispiele zeigen, dass Prolog die üblichen Vorrangregeln beachtet. `3 + 4 * 5` lässt sich als Summe schreiben aber nicht als Produkt.

2.3 Unifikation und Resolution

Nachdem Sie die Syntax kennengelernt haben, sollen Sie kurz mit der *operationalen Semantik* von Prolog, das heißt mit der Art, wie Prolog-Programme ausgeführt werden, vertraut gemacht werden. Die beiden wichtigen Grundbegriffe sind die *Unifikation* und die *Resolution*. Sie realisieren zusammen ganz grob die Mechanismen des Funktionsaufrufs und der Funktionsausführung in prozeduralen Sprachen.

Die Unifikation ermittelt eine Variablenersetzung, die die Instanzen zweier Terme identisch macht.

Die Resolution ist eine logische Schlussfolgerung. Die Resolution und die Suchstrategie des Prolog-Interpreters bestimmen die logische Grundlage des Prolog-Systems.

2.3.1 Die Unifikation

Zunächst sollten wir uns mit der Unifikation in Prolog vertraut machen. Sie ist die einzige Form, in der eine Variablenbindung vorgenommen werden kann und übernimmt damit die Aufgabe von Zuweisung und Parameterübergabe der prozeduralen Programmierung.

Zunächst soll der Begriff der *Substitution* definiert werden.

Definition:

*Eine **Substition** ordnet einer Variablen einen Term zu. Variablen können in allen Termen, in denen Sie auftreten durch den zugeordneten Term ersetzt werden. Die Konkretisierung eines Terms durch (teilweises) Ersetzen von Variablen nennt man auch **Instanziierung**. Eine Substituion $X \leftarrow Y$ einer Variablen durch eine andere Variable stellt eine bloße Umbennung dar. In Prolog nennt man dies auch* sharing von Variablen.

Beispielsweise wird der Term

```
f(g(X,Y), Y)
```

durch die Substitution $X \leftarrow g(Z), Y \leftarrow 3$ zu

```
f(g(g(Z),3), 3).
```

Die Unifikation versucht für zwei Terme eine gemeinsame Substitution zu finden, die beide Terme gleich macht.

Definition:

*Die **Unifikation** ist eine Instanziierung zweier Terme, die diese beiden Terme identisch macht. Der **Unifikator** ist die Liste der dazu nötigen Substitutionen. Prolog bestimmt bei der Unifikation den **allgemeinsten Unifikator** (*mgu = most general unificator*), der nicht mehr als die unbedingt nötigen Variablenbindungen vornimmt.*

Als Beispiel können wir daran denken, dass wir die Funktionswerte mathematischer Funktionen gespeichert haben. Als nächstes wollen wir einen bestimmten Funktionswert auffinden.

```
% Datenbasis
...
wert(sin(30), 0.5).

% Anfrage
?- wert(sin(30), X).
```

Damit Prolog auf die Anfrage antworten kann, muss es mehrere Schritte durchführen:

1. Prolog prüft ob das Anfrageliteral und das Fakt gleichen Namen und gleiche Stelligkeit (Parameterzahl) haben. Dies ist hier der Fall. In Prolog-Schreibweise gilt beide male `wert/2`.
2. Prolog überprüft Parameter für Parameter, ob diese übereinstimmen oder durch Einsetzen für noch offene Variable übereinstimmend gemacht werden können.

Die Durchführung der Unifikation am Sinus-Beispiel besteht aus folgenden Schritten:

1. Beide Literale heißen `wert` und haben die Stelligkeit 2.
2. Als nächstes gilt es, die Terme `sin(30)` und `sin(30)` aus Anfrage und Fakt zu unifizieren. Dass dies geht, ist sofort klar. Prolog geht natürlich dabei so vor, dass es zunächst wieder Name und Stelligkeit von `sin` prüft und anschließend die Parameter untersucht.
3. Jetzt nehmen wir uns das jeweilige 2. Argument vor. Also einmal `X` und zum andern `0.5`. Da `X` für eine Variable steht, ist diese Unifikation möglich, indem `X` an den Wert `0.5` gebunden wird.

Wenn man die Unifikation in Prolog genau beschreiben will, muss man angegeben, wie sie für die verschiedenen Arten von Parametern zu verstehen ist:

1. Zwei Konstante sind genau dann unifizierbar, wenn sie gleich sind.
2. Eine ungebundene Variable ist mit einem beliebigen Ausdruck unifizierbar. Die Variable erhält dann diesen Ausdruck als Wert und gilt als *gebunden*.
3. Zwei verschiedene ungebundene Variable sind stets unifizierbar. Nach der Unifikation werden die beiden Variablen als verschiedene Namen der gleichen Größe aufgefasst, so wie aus der mathematischen Gleichung $x = y$ folgt, dass ich überall x für y einsetzen kann und umgekehrt.
4. Eine gebundene Variable steht für den an sie gebundenen Wert. Dieser muss dann mit dem Gegenstück unifizierbar sein.
5. Zwei (komplexe) Terme sind nur dann unifizierbar, wenn sie nach Name und nach Stelligkeit übereinstimmen. Zusätzlich muss jedes Argument des ersten Terms mit dem entsprechenden Argument des anderen Terms unifizierbar sein.

Betrachten wir ein weiteres Beispiel (= erfordert immer die Unifikation von linker und rechter Seite):

```
abc(X,Y,3) = abc(Z, A, Z).

allgemeinster Unifikator: Z <- 3, A <- Y, X <-3
speziellerer Unifikator: zusaetzlich A <- 1, Y <-1
```

Anmerkung:

In Prolog hat nur der allgemeinste Unifikator Bedeutung. Die Betonung auf allgemeinst *drückt nur aus, dass es keinen Sinn macht, unnötige und willkürliche Variablenersetzungen vorzunehmen.*

Grundsätzlich ist es nicht notwendig, den genauen Hergang der Unifikation zu kennen. Das Ergebnis ist unabhängig von der Reihenfolge der Aktionen. In dem Beispiel können wir unmittelbar sehen, dass `Z` gleich 3 sein muss. Daraus ergibt sich, dass auch `X` gleich 3 ist. Für `A` und `Y` können beliege Werte stehen. Die einzige Voraussetzung ist, dass stets `A` gleich `Y` ist.

Die Unifikation übernimmt in Prolog die Funktion der Parameterübergabe von funktionalen Programmiersprachen. Sie ist aber erheblich mächtiger. Nehmen wir einmal ein einfaches Beispiel:

```
% Fakt:
equal(X, X).

% interaktiver Dialog:
?- equal(hans, Y).
Y = hans
?- equal(Y, hans).
Y = hans
?- equal(hans, heinrich).
false
```

Alle klar? Die Lösung besteht darin, dass bei der ersten Anfrage zunächst `X` an `hans` gebunden und dann Y ebenfalls an `hans` gebunden wird, da `X` ja inzwischen für `hans` steht. Die zweite Anfrage führt zunächst zur Unifikation von `X` und `Y`. Da die beiden Variablen jetzt für dasselbe Objekt stehen, führt die Bindung von `X` an `hans` schließlich dazu, dass auch `Y` an `hans` gebunden ist. Im dritten Fall scheitert schließlich die Unifikation, da ja die Variable `X` zunächst an `hans` gebunden wird und da anschließend festgestellt wird, dass die beiden konstanten Namen `hans` (für `X`) und `heinrich` verschieden sind.

Betrachten Sie nun noch diese Unifikationsaufgabe:

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

Die richtige Antwort lautet `false`. Aus Performancegründen kommen aber Prolog-Implementierungen damit meist nicht klar. Sie liefern falsche Antworten oder verheddern sich in endloser Rekursion. Dieses Problem ist bekannt und kann vom Programmierer leicht vermieden werden. Aus Performancegründen ist der Prolog-Unifikationsalgorithmus aber meist so implementiert, dass er dieses Problem nicht erkennt.

2.3.2 Die Resolution

Die gerade besprochene Unifikation stellt im Vergleich zu prozeduralen Sprachen – bei allen Unterschieden! – das Gegenstück zur Variablenbindung beim Funktions*aufruf* dar. Was uns jetzt noch fehlt, ist das Gegenstück zur Übergabe Funktions*ausführung* an den Funktionskörper. So wie also der Funktionskopf durch den

Funktionskörper ersetzt wird. Prolog verwendet hier die aus der Prädikatenlogik übernommene Deduktionsregel der *Resolution*.

Definition:

*Die **binäre Resolution** ist eine Schlussregel der Prädikatenlogik. Allgemein wird dabei ein positives Literal einer Klausel mit einem negativen Literal unifiziert und anschließend wird aus den verbleibenden Literalen beider Klauseln nach Instanziierung durch den Unifikator eine neue Klausel gebildet. In Prolog wird sie (in eingeschränkter Form) so durchgeführt, dass das linkeste Literal Anfrageklausel mit dem Kopf einer Regel unifiziert wird. Der Körper der Regel wird sodann der Anfrage vorangestellt.*

Um den Vorgang zu illustrieren, führe ich ein kleines Beispiel ein:

```
% Fakten und Regeln
g(hans, m).
e(hans, karin).
v(X, Y):- e(X, Y), g(X, m).

% Anfrage
?- v(Z, karin). % Wie heißt der Vater von Karin?
```

Dieses Programm entspricht dem einleitenden Prolog-Beispiel. Nur habe ich hier etwas kürzere Namen verwendet.

Als erstes müssen wir eine Unifikation durchführen, nämlich zwischen den beiden Literalen `v(Z, karin)` und `v(X, Y)`. Dies führt zu den Bindungen $Z \leftarrow X$ und $Y \leftarrow$ karin. Aber wieso gerade diese beiden Literale?

Die Antwort wird durch das Resolutionsverfahren gegeben. Zunächst ist das (linkeste) Literal der Anfrage ein Partner der Unifikation. Der zweite Partner ist entweder ein Fakt oder der Kopf einer Regel. Der Kopf einer Regel ist das links von `:-` stehende Literal.

Die Unifikation führt zu der angegebenen Variablenbindung. Wir sind aber noch nicht fertig! Wir müssen jetzt den Körper der Regel beachten. Dazu ersetzen wir zunächst die Variablen durch die an sie gebundenen Werte und betrachten dann den Körper selbst als neue Anfrage (genauer: der Körper ersetzt, das gerade verwendete Literal in der Anfrage).

In etwas abgekürzter Schreibweise, kann man diesen Ablauf so darstellen:

```
% Programm:
/* 1 */ g(hans, m).
/* 2 */ e(hans, karin).
/* 3 */ v(X, Y):- e(X, Y), g(X, m).

% Ablauf:
/* R */: ?- v(Z, karin)
/* 3 */:    v(X, Y) :- e(X, Y), g(X, m)  // X <- Z, Y <-
   karin
         ------------------------------
/* R */  :- e(Z, karin), g(Z, m)
/* 2 */     e(hans, karin)               // Z <- hans
         ------------------------------
/* R */  :- g(hans, m)
```

```
/* 1 */     g(hans, m)
          ------------------------------
              []
```

Es ist zu beachten, dass die Reihenfolge der Auswahl der Teilziele einer Anfrage von der Logik her irrelevant ist. In dem Beispiel ist die Vorgehensweise von Prolog, nämlich von links nach rechts, angewendet.

Da bei jeder Unifikation und bei jeder Resolution zwei Ausdrücke vorkommen, habe ich hier eine Art Rechenschema gewählt. In diesem Schema können Sie (zunächst als Merkregel, die eigentliche Begründung kommt später) das Minuszeichen in `?-` und in `:-` als eine Form der Negation ansehen. Die Resolution ist in dieser Denkweise nichts anderes als die Unifikation der beiden führenden Literale, gefolgt von deren Streichen (da sie ja bis auf das *Vorzeichen* identisch sind) und die Kombination aller übrigbleibenden Literale als eine *negative* Unteranfrage.

2.3.3 Die Prolog Backtracking-Strategie

In dem gerade dargestellten Beispiel war nicht klar, ob wir zunächst die Frage nach `e(Z, karin)` oder zunächst die nach `g(Z. m)` klären. Das Ergebnis ist jedenfalls immer gleich. Es kann aber komplizierter werden, wenn wir nicht wissen, welche von unterschiedlichen Regeln wir zunächst anwenden sollen (man spricht hier einmal von *Zielreihenfolge* (goal order), bei der Reihenfolge der Anfrageliterale, und von *Regelreihenfolge* (rule order), bei der Reihenfolge der Regelauswahl.

Die logische Korrektheit eines Programms sollte nicht von der Reihenfolge der Ableitungsschritte abhängen. Dies ist auch damit gemeint, wenn man von einem deklarativen Programmierstil spricht.

Definition:

*Eine **Deklaration** stellt eine Aussage dar. In einem **deklarativen Programm** ist die Bedeutung des Programms nicht an einen bestimmten Ablauf gebunden.*

Der Unabhängigkeit von der Reihenfolge der Verarbeitungsschritte kann am besten durch einen Suchbaum dargestellt werden. In diesem sind alle notwendigen Schritte enthalten ohne dabei eine bestimmte Reihenfolge festzulegen.

Die Abb. 2.1) stellt die Ableitungsschritte für ein einfaches Prolog-Programm dar. Verzweigungen entsprechen den jeweiligen Alternativen. Die Kanten sind mit den verwendeten Regeln und den dabei vorzunehmenden Unifikationen (Variablenersetzungen) markiert. Die Knoten stellen die jeweiligen Resolventen dar.

Die Lösungssuche besteht in der Suche nach einem erfolgreichen Endpunkt des Baums. Grundsätzlich kann man einen Baum unterschiedlich durchlaufen. Prolog geht nach der *Tiefensuche* vor.

Durch den genau festgelegten Ablauf erhalten Prolog-Programme auch eine prozedurale Komponente. Es ist möglich festzulegen in welcher Reihenfolgen Anweisungen mit Seiteneffekt *ausgeführt* werden sollen.

Zusammengefasst ergibt sich das folgende Schema für den Prolog-Ablauf.

- Bei der Beantwortung einer Anfrage, die ja im allgemeinen Fall eine Liste

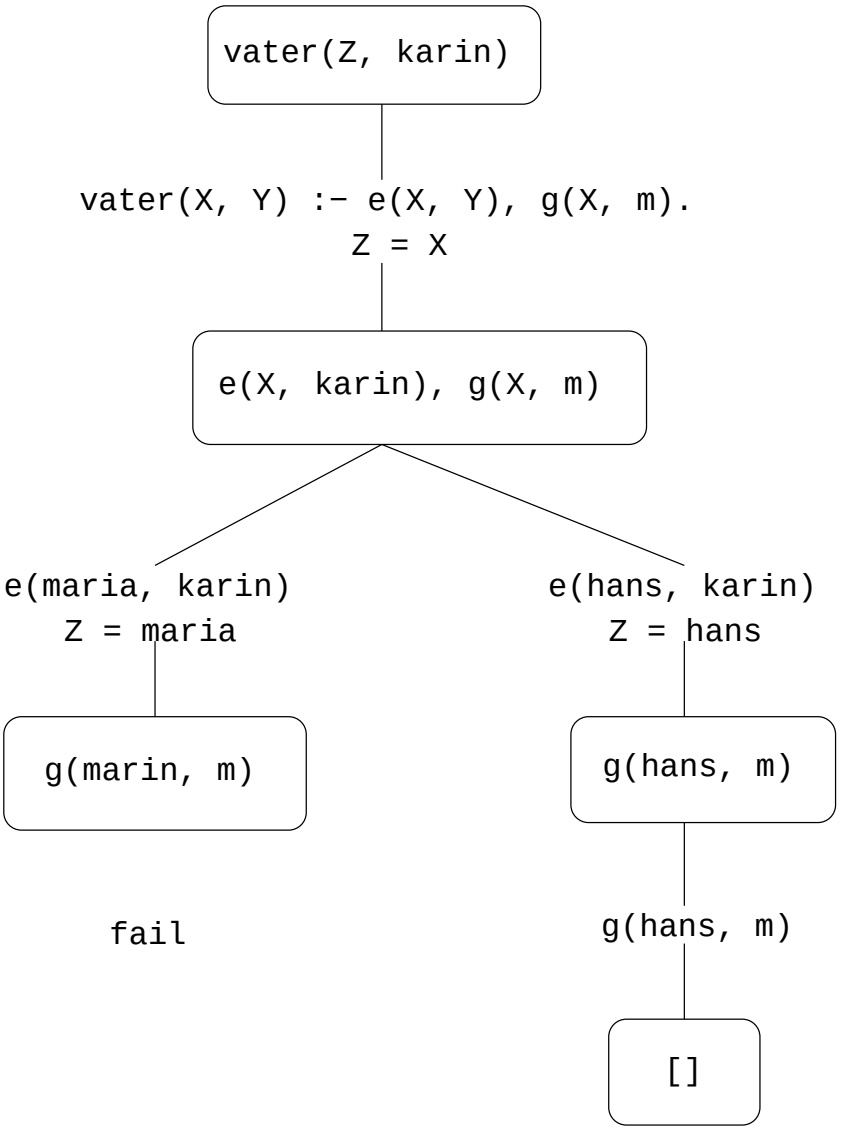

Abbildung 2.1: Prolog-Suchbaum. Die Ovale stellen die jeweiligen Resolventen dar.

von (negativen) Literalen darstellt, wählt Prolog zunächst das linkeste dieser Literale aus (*Zielliteral*).

- Als nächstes sucht Prolog ein nach Namen und Stelligkeit zum Zielliteral passendes Fakt oder einen passenden Regelkopf aus. Besteht ein Prädikat aus mehreren Fakt- oder Regelklauseln wird zunächst stets die erste gewählt. Für den Fall, dass diese Auswahl später revidiert werden muss, wird dieser *Auswahlpunkt* gespeichert.
- Prolog versucht Zielliteral und Programmliteral (Fakt/Regelkopf) zu unifizieren. Nötigenfalls muss durch Variablenumbenennung erreicht werden, dass vor der Unifikation in Anfrage und Programm keine gleichnamigen Variablen vorkommen.
- Wenn die Unifikation nicht gelingt, wählt Prolog die nächste passende Fakt-/Regelklausel.
- Wenn die Unifikation nicht gelingt, und wenn es keine weiteren passenden Klauseln gibt, wird zum nächsten möglichen *Auswahlpunkt* zurückgegangen und dort eine neue Wahl getroffen und so weiter (*Backtracking*).
- Wenn keinerlei Möglichkeit übrigbleibt, ist die ursprüngliche Anfrage gescheitert und Prolog antwortet mit `false`.
- Wenn die Unifikation gelingt, wird zunächst das Zielliteral aus der Anfrage entfernt.
- Als nächstes werden eventuelle Literale des Regelkörpes der Anfrage vorangestellt.
- Die bei der Unifikation gefundene Variablenbindung wird in der neuen Anfrage angewendet.
- Wenn die Anfrage leer ist, d.h. keine Literale mehr enthält, wird sie mit Erfolg beendet. Andernfalls wird der gesamte Ablauf für die neu entstandene Anfrage durchgeführt.

Die bei der Resolution aus Anfrage und Regelkörper entstehende Klauselliste heißt *Resolvente*. Der Prolog-Ablauf kann daher auch so erklärt werden, dass zuerst die Anfrage zur Resolventen erklärt wird und dann wiederholt zwischen der Resolventen und einer passend ausgesuchten Programmklausel eine Resolution durchgeführt wird, die beim Erreichen der leeren Resolventen mit Erfolg abgeschlossen ist. Wenn an irgendeinem Punkt keine Möglichkeit zur Resolution besteht, wird zum letzten Auswahlpunkt zurückgegangen und dort wird dann die Resolution mit einer anderen Programmklausel versucht. Dieses Verfahren heißt *Backtracking*. Beim Backtracking, d.h. beim Zurücknehmen von Resolutionen, werden die bei der Resolution/Unifikation vorgenommenen Variablenbindungen wieder aufgehoben.

Als Beispiel folgt nochmals das letzte Programm. Dieses mal ist es so erweitert, dass auch die Mutter von Karin gespeichert ist. Dies führt dazu. dass die Suche nach dem Vater erst in die Irre geht und durch Backtracking gelöst werden muss.

```
% Programm:
/* 1 */ g(hans, m).
```

```
/* 2 */ g(maria, f).
/* 3 */ e(maria, karin).
/* 4 */ e(hans, karin).
/* 5 */ v(X, Y):- e(X, Y), g(X, m).

% Ablauf:
/* R */: ?- v(Z, karin)
/* 5 */:    v(X, Y) :- e(X, Y), g(X, m)  // X <- Z, Y <-
   karin
         ------------------------------
/* R */  :- e(Z, karin), g(Z, m)
/* 3 */     e(maria, karin)               // Z <- maria (1)
         ------------------------------
/* R */  :- g(maria, m)
         fail => Backtracking
         ------------------------------
/* R */  :- e(Z, karin), g(Z, m)
/* 4 */     e(hans, karin)               // Z <- hans (2)
         ------------------------------
/* R */  :- g(hans, m)
/* 1 */     g(hans, m)
         ------------------------------
         []
```

Dies ist eine *logische* Darstellung der Suche. Der Prolog-Interpreter selbst verfügt über einen automatischen Trace-Mechanismus, der eine ähnliche Ausgabe erzeugt. Diese ist insofern etwas prozeduraler gestaltet, dass auch die Rückkehr von erfolgreicher (*exit*) und erfolgloser Suche (*fail*) angezeigt ist. Sie sieht für das Beispiel so aus:

```
[debug] 11 ?- v(Z, karin).
 T Call: (6) v(_G26538, karin)
 T Call: (7) e(_G26538, karin)
 T Exit: (7) e(maria, karin)
 T Call: (7) g(maria, m)
 T Fail: (7) g(maria, m)
 T Redo: (7) e(_G26538, karin)
 T Exit: (7) e(hans, karin)
 T Call: (7) g(hans, m)
 T Exit: (7) g(hans, m)
 T Exit: (6) v(hans, karin)
Z = hans.
```

`_G26538` steht für eine Variable (hier `X`). Intern werden nämlich bei jeder Anwendung einer Regel neue Namen vergeben. `Redo` bezeichnet die Wiederaufnahme der Suche nach Backtracking.

Dies ist ganz grob der normale Ablauf des Prolog-Interpreters. Eine Besonderheit bilden eingebaute Systemprädikate. Am einfachsten merken Sie sich, dass bei der Behandlung eines internen Prädikats einfach ein vordefinierter Ablauf den normalen Resolutionsmechanismus ersetzt. Viele Systemprädikate haben einen Seiteneffekt. Dabei wird die Reihenfolge der Ausführung beachtet.

Ein typisches Beispiel für das Programmierung mit Seiteneffekten ist die Ein- und Ausgabe. In dem folgenden Beispiel ist sicher die Reihenfolge der Ausführung entscheidend.

```
regel:- write('hello').
```

```
regel:- write(' ').
regel:- write('world'), nl.

?- regel, fail.
```

Die Anfrage mittels Backtracking „führt“ nacheinander die drei Regeln aus. Die verwendeten Systenmprädikate sind: `write` zur einfachen Ausgabe, `nl` für eine neue Zeile und `fail`, ein Prädikat, das nie erfüllt ist.

Das Beispiel war künstlich so geschrieben, dass mittels Backtracking die Ausführung der drei Regeln erzwungen wird. Sicher würde man das Ganze normalerweise wie folgt schreiben.

```
write_hello:- write('hello').
write_space:- write(' ').
write_world:- write('world'),

?- write_hello, write_space, write_world, nl.
```

Während die erste Fassung auf der Regelreihenfolge und dem Backtracking beruhte, kommt hier die Zielreihenfolge zum Tragen.

Nur ein ganz kleiner Teil der vordefinierten Prädikate ist wirklich als Systemprädikat eingebaut. Die allermeisten vordefinierten Prädikate sind bereits in Prolog selbst definiert.

Im nächsten Abschnitt will ich die prozedurale Sichtweise auf Prolog-Programme etwas vertiefen.

2.3.4 Die prozedurale Interpretation von Prolog

Durch die Festlegung der Auswahl von Zielliteral und Regelklausel erhalten Prolog-Programme einen genau nachvollziehbaren sequentiellen Ablauf. Damit lässt sich eine weitgehende Parallelität zwischen Prolog und prozeduralen Sprachen herstellen. Dies sollt hier an der Fibonacci-Funktion gezeigt werden.

Zunächst die Java-Fassung:

```
 1 int fibo(int n) {
 2     if (n == 0)
 3         return 0;
 4     if (n == 1)
 5         return 1;
 6     if (n > 1) {
 7         int n1 = n - 1;
 8         int n2 = n - 2;
 9         int f1 = fibo(n1);
10         int f2 = fibo(n2);
11         int f = f1 + f2;
12         return f;
13     }
14 }
```

Ihnen fällt vielleicht auf, dass diese Form etwas umständlich aussieht. Durch die etwas pedantische Form wird aber die Ähnlichkeit zu Prolog betont. Der logische

Background von Prolog erlaubt nämlich keine kurze funktionale Formulierung. Jeder Wert muss für sich berechnet werden. Das äquivalente Prolog-Programm sieht jetzt so aus:

```
 1 % fibo(N, F)
 2 %  F = fibo(N)
 3 fibo(0, 0).
 4 fibo(1, 1).
 5 fibo(N, F):-
 6     N > 1,
 7     N1 is N - 1,
 8     N2 is N - 2,
 9     fibo(N1, F1),
10     fibo(N2, F2),
11     F is F1 + F2.
```

Sie sehen, dass die eine Java-Funktion in eine Folge von drei Prolog-Klauseln zerfällt. Jede Klausel für sich entspricht einem If-Zweig. Die logische Bedingung kann in Prolog durch Konstanten im Kopf der Klausel ausgedrückt werden, wie in den Zeilen 3 und 4 (der Vergleich wird durch Resolution/Unifikation vorgenommen) oder aber auch als eigene Bedingung formuliert werden, wie in der Zeile 6. Die Konjunktion von Literalen in den Zeile 7–11 entspricht genau der sequentiellen Formulierung von Java.

Zusammengefasst ergeben sich also die folgenden „Vergleichsregeln“:

- Ein Prolog Prädikat entspricht einer Prozedur, der „Aufruf“ eines Prädikats entspricht einem Prozeduraufruf. Die Argumente eines Klauselkopfes entsprechen den Übergabeparametern. *Prolog kennt keine funktionalen Rückgabewerte.*
- Die Liste von Literalen im Klauselkörper entspricht einer Sequenz von Anweisungen. Durch die festgelegte Zielauswahl werden sie in der üblichen Reihenfolge von oben nach unten[3] verarbeitet.
- Eine Alternative wird in Prolog durch Angabe mehrerer Klauseln zu einem Prädikat ausgedrückt.
- Die Wiederholung wird in Prolog durch rekursive Prädikatsaufrufe realisiert.[4]

Der wichtigste Unterschied zwischen einer prozeduralen Sprache und Prolog besteht in dem grundverschiedenen Variablenbegriff.

In prozeduralen Sprachen bezeichnet eine Variable einen Speicherbereich, der im Laufe des Programmablaufs unterschiedliche Werte annehmen kann.

Eine Variable in Prolog steht für beliebige Werte. Eine Konkretisierung dieses Wertes kann nur durch die mit einer Unifikation verbundenen Substitution eintreten. Eine Prolog Variable kann nur einmal festgelegt und dann nicht verändert werden. Wenn allerdings durch Backtracking ein Teil der Lösungssuche rückgängig gemacht wird, werden auch die in der „Sackgasse“ vorgenommenen Bindungen wieder gelöscht.

[3] oder von links nach rechts

[4] Eine weitere Variante drückt Wiederholung durch Backtracking aus.

2.4 Abweichungen von der Logik

2.4.1 Eingebaute Prädikate

Arithmetik spielt innerhalb von Prolog und innerhalb von Logikprogrammierung eine Sonderrolle. Das äußert sich zunächst darin, dass arithmetische Berechnungen nur dann ausgeführt werden, wenn Sie innerhalb von bestimmten vordefinierten Prädikaten auftauchen. In jedem anderen Fall werden sie als bloße Formeln betrachtet:

```
?- X = 3 * 4.
X = 3 * 4
?- 3 * 4 = 3 * 4.
true
?- 3 * 4 = 12.
false
?- X is 3 * 4.
X = 12.
?- 3 * 4 =:= 12.
true
```

Variablen müssen zum Zeitpunkt der Ausführung gebunden sein, d.h. ihr Wert muss dann feststehen.

Das is-Prädikat

Die Syntax des is-Prädikats lautet:

is-Prädikat ::= *Variable* `is` *arithmetischer-Ausdruck*
| *Zahl* `is` *arithmetischer-Ausdruck*

In einem Ausdruck können Zahlen und die üblichen Rechenoperationen und mathematischen Funktionen stehen. In dem Ausdruck vorkommende Variablen müssen an einen numerischen Wert gebunden sein. Da Prolog auf Logik aufbaut, müsste Arithmetik eigentlich auch logisch begründet werden. Ehe Prolog numerische Berechnungen auf logischer Grundlage durchführen kann, müssen aber zunächst alle benötigten mathematischen Definitionen und Regeln in das Prolog-System integriert werden. Ich weiß nicht, ob das schon einmal jemand versucht hat. In der eingeschränkten logischen Ausdrucksfähigkeit von Prolog ist das auch nicht einfach. Jedenfalls würde die formale Arithmetik zwar sehr mächtig aber gleichzeitig auch extrem ineffizient sein. Nicht umsonst hat man in Prozessorbausteinen die mathematischen Grundoperationen in die Hardware integriert. Prolog macht also auch nichts anderes als andere Programmiersprachen, wenn es für die Arithmetik auf Systemprädikate zurückgreift.

Nachdem der Wert des arithmetischen Ausdrucks berechnet ist, wird er mit dem links von `is` stehenden Term unifiziert. Steht dort eine Zahl oder eine bereits gebundene Variable, so entspricht die Unifikation einem numerischen Vergleich. Steht links von `is` eine ungebundene Variable, so erhält die Variable den Wert des Ausdrucks „zugewiesen“.

Arithmetische Vergleichsoperationen

Für Größenvergleiche gilt genauso wie für Berechnungen, dass sie am effizientesten durch die Computerhardware auszuführen sind. Also sind in Prolog auch hierfür Systemprädikate vordefiniert. Weitgehend entspricht ihre Darstellung den Ihnen bekannten Formen:

```
A =:= B        % arithmetische Gleichheit
A =\= B        % arithmetische Ungleichheit
A < B          % kleiner
A =< B         % kleiner oder gleich
A > B          % groesser
A >= B         % groesser oder gleich
```

Die Vergleichsoperationen werden so ausgeführt, dass zunächst beide Seiten arithmetisch ausgewertet werden und anschließend ein Vergleich der Ergebnisse stattfindet.

2.4.2 Weitere eingebaute Prädikate

Prolog verfügt über eine Vielzahl weiterer vordefinierter Prädikate.

- Die gerade besprochenen *arithmetischen Prädikate* sind nötig, da nur so eine effiziente Arithmetik implementiert werden kann.
- Einige Prädikate, wie *Vergleichsoperationen*), greifen auf die interne Darstellung von Datenelementen und Termen zu.
- Die *Ein- / Ausgabefunktionen* sind genauso vordefiniert wie die wichtigsten Operationen des Betriebssystems.
- Es gibt *Metaprädikate*, die es erlauben, die Prolog-Datenbasis gezielt zu interpretieren.
- *Prädikate höherer Ordnung* erweitern die Ablaufsteuerung von Prolog über die grundlegende Logik hinaus, indem sie selbst Prolog-Prädikate als Argumente erhalten.
- Viele Prädikate, wie z.B. *Listenprädikate*, sind wegen ihrer häufigen Verwendung ebenfalls vordefiniert.

Hier sollen nur einige der wichtigeren Prädikate angesprochen werden. Einige weitere werden später besprochen, wenn sie benötigt werden. Wenn Sie an Vollständigkeit interessiert sind, muss ich Sie auf die Literatur, z.B. auf das Online-Handbuch von SWI-Prolog verweisen.

2.4.3 Negation und Cut

In Prolog kann Verneinung nicht ausgedrückt werden. Dies hat ein paar Konsequenzen.

Definition:

*In Prolog gilt die Annahme einer abgeschlossenen Welt (*closed world assumption*. Diese besagt, dass alles was nicht explizit im Programm erwähnt ist nicht gilt. In Konsequenz antwortet Prolog auf jede nicht definitiv zu bejahende Frage mit* false*.*

Mit dieser Annahme kann man leben, da es oft ja auch durchaus der normalen Erwartung entspricht. Wenn etwas nicht gefunden wird, ist es nicht vorhanden.

Anders ist es mit der positivenVerneinung, die ausdrücken will, dass ein Sachverhalt gilt, wenn ein anderer nicht gilt. Als Beispiel soll die Formulierung von gleich und ungleich dienen.

```
gleich(X, X).

ungleich(X, Y) /* ??? */
```

Das Gleichheitsprädikat ist vollkommen korrekt. Natürlich gibt es schon das vordefinierte elegante =. Die Aussage ist aber exakt identisch: „Alles ist sich selbst gleich“.

Dagegen ist die Definition der Ungleichheit falsch. Hier sind wir zwingend auf vordefinierte Möglichkeiten angewiesen. Betrachten Sie nämlich einmal die folgenden Regeln:

```
ungleich(X, X):- false.
ungleich(X, Y):- true.
```

Zunächst sagt die 1. Regel, dass zwei gleiche Objekte nicht ungleich sind. Die 2. Regel will aussagen, dass alle Objekte, für die die erste Regel nicht zutrifft, ungleich sind. Dass ist zwar gut gemeint. Es ist aber erstens nicht logisch, da es eine Aussage über den Ablauf der Lösungssuche macht. Und es funktioniert auch nicht, da Prolog auch dann, wenn zunächst die 1. Regel greift, im Rahmen des Backtrackings dann doch die 2. Regel verwendet.

Wenn wir bei dieser Denkweise bleiben, bleibt uns nichts anderes übrig, als mit dem Systemprädikat ! (gesprochen „cut“) in die Lösungssuche einzugreifen und das Backtracking zu verhindern.

```
ungleich(X, X):-
    !,              % kein Backtrackting fuer diese Regel
    fail.
ungleich(X, Y).     % da kommt man nur hin, wenn die 1. Regel
                    % nicht gilt.
```

Auch wenn in diesem Beispiel diese Cut-Fail-Kombination „funktioniert“ so ist doch das vordefinierte Prädikate wie \= vorzuziehen. Diese Schreibweise ist erheblich lesbarer und verständlicher. Man sollte sich aber im Klaren sein, dass auch \= prozedural definiert ist.

Das Beispiel mit dem Cut (!) ermöglicht sehr vielseitige Optimierungen von Prolog. Ganz allgemein sagt der Cut aus, dass andere Alternativen zu der gerade angewendeten Regel nicht mehr infrage kommen.

Eine sinnvolle Verwendung ist die Optimierung von Prolog-Programmen. Es mag sein, dass ich in meinen Beispielen ab und an einen solchen Cut verwende. Dieser *grüne* Cut hat keine logische Bedeutung.

Eine andere Verwendung besteht darin zu verhindern, dass Prolog unlogische Regeln anwendet. Die genaue Betrachtung der bei `ungleich` verwendeten Cut-Fail Kombination macht dies deutlich. Die erste Regel sagt aus, dass etwas niemals sich selbst ungleich sein kann (`fail` ist einfach unerfüllbar). Die zweite Klausel sagt aus,. dass alles sich selbst ungleich ist. Das ist natürlich falsch. Der Prolog-Ablauf verwendet aber nun zunächst die erste Regel. Wenn die zu vergleichenden Objekte verschieden sind, scheitert die Unifikation. Die zweite Regel wird dann angewendet und ist erfolgreich. In dem Fall, dass die zu vergleichenden Objekte gleich sind darf die zweite Regel niemals angewendet werden. Genau dies verhindert der Cut. Die erste Regel ist erfolgreich, sie sagt mittels Cut, dass man keine andere Regel versuchen darf und dann sagt sie dass die Ungleichheit nicht gilt. Mit Recht empfinden Sie dies als sehr kompliziert.

Diesen zuletzt besprochenen, *roten* Cut werde ich in meinen Beispielen nicht verwenden. Das (abschreckende) Beispiel kann aber vielleicht dazu herhalten, deutlich zu machen, dass in der Informatik selten das Ideal der heilen Welt gilt. Immer wieder gibt es „Workarounds“, die oft zwar nötig aber auch doch sehr hässlich sind.

2.5 Die logische Grundlage von Prolog

Auch wenn ich mich etwas scheue, zu tief in die formale Logik einzusteigen. Wenn wir Prolog als *Logikprogrammiersprache* verstehen wollen, kommen wir um eine logische Rechtfertigung der Mechanismen nicht herum.

2.5.1 Anforderungen

Als vollständige Verkörperung von der Logik sollte Prolog mehrere Mindestanforderungen erfüllen:

- Die Herleitungsmechanismen müssen *korrekt* sein. Es darf nicht möglich sein, dass falsche Ergebnisse gefolgert werden.
- Die Ausdrucksmöglichkeiten der Sprache sollten die gesamte Logik abdecken.
- Alle wahren Sätze sollten ein endlich vielen Schritten ableitbar sein. (*Vollständigkeit*).

Zusätzlich gibt es einige Forderungen und Wünsche, die sich aus der Eigenschaft einer Programmiersprache ergeben:

- Die Ablauf eines Programms sollte deterministisch sein.
- Programme sollte effizient ablaufen.
- Die Programmierumgebung sollte prozedurale Erweiterungen zulassen.

Man kann leicht einsehen, dass die Forderung der logischen *Korrektheit* sich gut mit der Korrektheit der Mechanismen einer Programmiersprache verträgt. Aber die Forderungen nach *vollständiger* Umsetzung der Logik und *effizienter* Ausführung eines Programms stehen in einem unlösbaren Widerspruch. Prolog löst diesen Konflikt indem es einige Kompromisse eingeht.

2.5.2 Logik und Prolog-Syntax

Prolog ist von vornherein auf die Prädikatenlogik erster Stufe beschränkt. Dies bedeutet, dass logische Formeln mit den bekannten logischen Operatoren der Aussagenlogik aufgebaut sein können. Darüber hinaus kennt die Prädikatenlogik Variablen und funktionale Ausdrücke. Variablen können quantisiert sein(für alle x, es gibt ein x). In Prädikatienlogik erster Stufe können Prädikatsnamen nicht durch Variablennamen ersetzt werden.

Die Prädikatenlogik erster Stufe hat als Basis für die Programmierung den Vorteil, dass es formale Systeme gibt, die es erlauben, in endlich vielen Schritten jeden wahren Satz formal zu beweisen (Vollständigkeit). Das ist für höhere Systeme der Logik nicht mehr der Fall.[5]

Prolog nimmt nun, wie gesagt ein paar Vereinfachungen vor. Zunächst vermeidet es die Quantisierungsoperatoren. Dies gelingt relativ leicht. Für Existenzquantoren wird einfach verlangt, dass man sie durch ein *Skolemisierung* genanntes Verfahren durch Konstanten und Funktionen ersetzt. Alle übrigen Variablen gelten als allquantisiert. Man benötigt dafür keine besondere Schreibweise.

Die nächste Vereinfachung betrifft die logischen Formeln, die in Prolog formulierbar sind. Sie kennen aus der Aussagenlogik die Normalisierung von logischen Ausdrücken. Diese ermöglicht die Umwandlung von Formeln in eine äquivalente standardisierte Form. Auch Prolog geht so vor.

Systeme der Logikprogrammierung basieren auf der *Klauselform*. Die Klauselform ist nichts anderes als eine vereinfacht geschriebene konjunktive Normalform.

Nehmen wir ein Beispiel für eine aussagenlogische Formel:

$$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b) \wedge (a)$$

Die Klauselschreibweise nutzt die Regelmäßigkeit der Normalform aus. Wir können es uns schenken, die Operatoren $\vee$ und $\wedge$ auszuschreiben. Wir wissen, dass in der innersten Klammerebene immer nur Oder-Verknüpfungen stehen und außerhalb nur Und-Verknüpfungen. Die Reihenfolge der Operanden innerhalb einer Verknüpfung spielt keine Rolle. Dies führt dazu, dass wir die Formel als eine Menge von Mengen von Literalen schreiben können. Die Literale sind positive oder negative atomare Ausdrücke. In der Klauselschreibweise schreibt man dann die Negation meist durch ein Minuszeichen. Die obige Formel sieht in Klauselform so aus:

$$\{\{a, b\}, \{a, -b\}, \{-a, -b\}, \{a\}\}$$

[5] Man braucht aber bereits für die formale Definition der Eigenschaften der natürlichen Zahlen die Prädikatenlogik zweiter Stufe.

Die inneren Mengen der Klauselform heißen *Klauseln*. Die gesamte Aussage heißt auch Klauselmenge. Man kann sich gut vorstellen, dass die Klauseldarstellung sich gut für die computerinterne Verabeitung von Logik eignet, da sie gut durch Listenobjekte darstellbar ist.

In der Klauselform gibt es ein paar Namensgebungen. Zunächst unterscheidet man *positive Literale* und *negative Literale*. Dann spricht man von *Einheitsklausen*, wenn die Klausel nur ein einziges Literal enthält. letzte Aufgrund ihrer besonderen Eigenschaften hat man für eine Teilmenge von logischen Aussagen einen eigenen Namen erfunden. Man bezeichnet Klauseln als Hornklausel, wenn sie maximal ein positives Literal enthalten.

Definition:

*Eine **Hornklausel** ist eine Klausel mit höchstens einem einzigen positiven Literal. Wenn vorhanden, heißt das positive Literal **Kopf der Klausel** und die Menge der negativen Literale heißt **Körper der Klausel**.*

Schauen wir uns die oben angegebene Klauselmenge an:

1. $\{a, b\}$ enthält zwei positive Literale und ist *keine Hornklausel*.
2. $\{a\}$ enthält ein positives Literals. Es ist eine positives Einheitsklausel und auch eine *Hornklausel*
3. $\{-a, -b\}$ enthält nur negative Literale (negative Klausel) und ist eine *Hornklausel*
4. $\{a, -b\}$ ist eine *Hornklausel*

Da in Prolog nur Hornklauseln zulässt, sind Aussagen der ersten Form nicht möglich. Einige Aussagen lassen sich daher in Prolog nicht gut formulieren. Wir können leicht ausdrücken, dass Hans Vater von Karin *und* von Fritz ist. Wir können aber nicht als Formel hinschreiben, dass Hans Vater von Karin *oder* von Fritz ist.

Die andern drei Formen haben in Prolog jeweils eine besondere Bedeutung. Die positiven Einheitsklausen stellen in Prolog *Fakten* dar. Die Klausel $\{a\}$ lautet in Prolog `a.`

Die negativen Klausel entsprechen den interaktiven *Anfragen* von Prolog. Die Klausel $\{-a, -b\}$ lautet `?- a, b.` Beachten Sie, dass am Anfang der Anfrage ein Minuszeichen steht. Es bezieht sich sinngemäß auf alle Literale der Anfrage.

Die Hornklausel $\{a, -b\}$ ist eine Prolog-Regel `a :- b.` Auch hier steht das Minuszeichen vor einer Menge von negativen Literalen.

Die Bedeutung der Einschränkung auf Hornklauseln liegt darin, dass jede Klausel ein ausgezeichnetes positives Literal enthält. In Prolog ist dieses positive Literal der *Kopf* einer Regel im Unterschied zum negierten Körper der Regel. Diese Unterscheidung ermöglicht einen effizienten Interpreterablauf und ermöglicht auch eine leichte Interpretation der Formeln.

Nehmen wir die Klausel $\{a, -b, -c\}$. Diese Klausel lässt sich umformen in $(b \wedge c) \Rightarrow a$. Die Umformung zeigt, dass man Hornklauseln lesen kann als Regel: „Aus b und c folgt a“ oder als Definition „a gilt, wenn b und c“ gilt.[6]

[6] Bedeutet das Komma in Prolog nun „und“ ($\wedge$) oder „oder“ ($\vee$)? Das kommt auf die Lesart an.

2.5.3 Das negative Resolutionskalkül

Nachdem wir die Menge der möglichen Formeln eingeschränkt haben, können wir die Frage nach der logischen Grundlage von Prolog beantworten: „Prolog basiert auf einem negativen Resolutionskalkül“.

Negatives Resolutionskalkül bedeutet, dass Prolog versucht, einen *Widerspruchsbeweis* mittels Resolution zu führen. Wir haben ja zunächst ein *Programm* als eine Menge von Fakten und Regeln. Das Programm stellt das positive Wissen dar. Aus Sicht der Logik sollte das Programm *erfüllbar* sein. Zu dem Programm kommt dann die Anfrage. In Prolog wird eine Anfrage als eine verneinte Aussage aufgefasst. Ist nun die Anfrage selbst aus dem (selbst widerspruchsfreien) Programm herleitbar, dann steht die Verneinung der Anfrage bestimmt im Widerspruch zum Programm.

Das ist nun genau die Idee hinter Prolog. Prolog versucht nicht, die Antwort aus dem Programm herzuleiten (dabei entstünde ja die Frage, wie man da vorgehen soll). Stattdessen versucht Prolog zu beweisen, dass die Behauptung, dass Frage nicht erfüllbar ist, zu einem logischen Widerspruch führt. Wenn ein Widerspruch vorliegt, ist die Frage mit „ja“ zu beantworten (dabei wird die Einschränkung vorgenommen, dass dies für bestimmte Werte der Variablen gilt).

Nehmen wir an, wir haben folgendes Programm und die folgende Frage:

```
weiblich(karin).

?- weiblich(karin).
```

Nach der logischen Auffassung haben wir hier einen offensichtlichen Widerspruch. Das Programm sagt: „Karin ist weiblich“. Die negierte Frage sagt: „Karin ist nicht weiblich“. Die Resolution erkennt diesen Widerspruch. Unifizierbare positive und negative Ausdrücke heben sich auf. Übrig bleibt die leere Klausel. Die leere Klausel ist in der Logik nichts anderes als ein Widerspruch.

Das Vorkommen von Variablen, ändert nichts Grundsätzliches. Prolog versucht in jedem Fall einen Widerspruch zu finden. Die bei der Resolution versuchte Unifikation setzt für Variable solche Werte ein, die zum Widerspruch führen:

```
weiblich(karin).

?- weiblich(X).

  -weiblich(X)         // negative Frage
   weiblich(karin)     // X = karin
   --------
   []
```

Die (formale) Antwort auf die Frage `?- weiblich(X)` lautet: „Wenn wir annehmen, dass `X` für `karin` steht, dann ergibt die negierte Frage einen Widerspruch, also ist $weiblich(karin)$ aus dem Programm ableitbar.

Das Beantworten von Faktenfragen ist einfach. Man kann die Vorgehensweise

In der Klauselform (Normalform) ist es eindeutig ein $\vee$. In der Schreibweise als Regel (Implikation) ist es als $\wedge$ zu lesen.

aber auch leicht auf Regeln ausdehnen. Dies soll hier an einem Beispiel verdeutlich werden:

```
a.
b :- a.

?- b.

  -b       // negierte Frage
   b:- a   // Regel a => b
   ------
  -a       // wenn a gilt, ergibt sich ein Widerspruch
   a       // Fakt.
   ------
   []      // wir haben den Widerspruch
```

Es bleibt eine weitere Anmerkung zu Prolog. Um die angedeutete Vorgehensweise rechtfertigen zu können, verlangt Prolog, dass das Programm keine negativen Klauseln enthält. Negative Klauseln sind ja die Anfrage. Das Programm enthält in jeder Klausel genau ein positives Literal.

Diese Einschränkung ermöglicht erst die systematische Suche nach dem nächsten Resolutionspartner (Regel mit „passendem“ Kopf). Gleichzeitig garantiert sie, dass das Programm von sich aus noch keinen (formalen) Widerspruch enthalten kann. Diese Voraussetzung erlaubt die Folgerung, dass gefundenen Widersprüche zur positiven Beantwortung der Frage führen.

Das Verbot negativer Programmklauseln ist eine ganz wesentliche Einschränkung der Ausdrucksfähigkeit von Prolog. Es lassen sich in Prolog keine negativen Aussagen formulieren. Man behilft sich in der Prolog-Welt mit der Annahme, dass alles das, was nicht positiv belegt ist, verneint werden muss. Diese Annahme nennt man „Annahme einer geschlossenen Welt“ (closed world assumption). Wenn im Programm nicht festgestellt ist, dass Karin ein Kind hat, dann folgert Prolog, dass Karin definitiv kein Kind hat. Andersherum, „wenn sie ein Kind hätte, dann müssten wir das wissen“.

Merksatz:

*Prolog geht davon aus (**closed world assumption**), dass alles relevantes Wissen im Programm enthalten ist. Es nimmt an, dass eine Frage, die im Programm nicht bejahrt werden kann, definitiv zu verneinen ist.*

In der Logik ist nicht zwingend vorgeschrieben, in welcher Reihenfolge man verschiedene Regeln zu einer Herleitung heranzieht. Wenn man eine Ableitung gefunden hat, ist es schließlich egal, wie man darauf gekommen ist.

Prolog verwendet, wie bereits erwähnt, die Tiefensuche als *Strategie*. Dies hat den Vorteil großer Speichereffizienz und einfacher prozeduraler Interpretation. Tiefensuche ist aber (im Vergleich zur Breitensuche) grundsätzlich unvollständig! Für Prolog heißt das, dass es manche Fragen nicht beantworten kann, selbst wenn dies grundsätzlich formal möglich ist. Das sind die Kosten der Effizienz. Prolog-Programmierer müssen diesem Nachteil durch entsprechende Programmierung Rechnung tragen.

Kapitel 3

Logikprogrammierung in Prolog

3.1 Grundregeln

Logikprogrammierung geht von der Vision aus, dass sich alles Wissen durch logische Formeln beschreiben lässt.

Bei der Darstellung von Fakten- und Regelwissen erscheint dies unmittelbar einleuchtend. An einigen Beispielen wurde dies bereits im letzten Kapitel erläutert. Eine weitere Stärke ist die Fähigkeit, elegant mit symbolischen Ausdrücken umzugehen. Beide Eigenschaften sind wichtig für die Anwendung von Prolog in der Künstlichen Intelligenz.

Prolog hat zum Ziel, Programme möglichst deklarativ (ohne einen konkreten Ablauf zu beschreiben) auszudrücken.

Dies ist aus mehreren Gründen nicht vollständig möglich:

- Ein-/Ausgabeoperationen stehen mit der äußeren Umgebung in Beziehung. Sie habe notwendigerweise eine bestimmte Reihenfolge.
- Eine effiziente Arithmetik lässt sich (bei der gegebenen Rechnerhardware) nur imperativ realisieren.
- Prolog unterstützt die Formulierung von Interpretern und von Metaprogrammierung. So deklarativ das aussehen mag, in der Regel setzt die Funktionsweise einen bestimmten Ablauf voraus.
- Optimierung setzt die Kenntnis des Ablaufs voraus.
- Die Frage, ob eine Variable gebunden oder frei ist, macht nur im Kontext eines konkreten Ablaufs Sinn.
- Der Cut (!) beeinflusst direkt den Ablauf.

Auch in Prolog hat man bei der effizienten Programmierung von Algorithmen einen Ablauf im Kopf. Ein gutes Logikrogramm sollte aber unabhängig vom konkreten Ablauf logisch korrekt sein. Wie auch in der funktionalen Programmierung hat das Vorteile bei dem Verständnis, für die Fehlerfreiheit und gegebenenfalls auch bei der Optimierung.

- Die Variablen der Logikprogrammierung sind entweder ungebunden (frei) oder gebunden. Eine einmal gebundene Variable kann ihren Wert nicht ändern.
- Die Bedeutung eines Logikprogramms ist unabhängig von der Reihenfolge der „Ausführung“. Die prozedurale Reihenfolge spielt aber für die effiziente Ausführung (und für Ein-/Ausgabe) eine Rolle.
- Das Ergebnis einer Operation auf Datenstrukturen ist eine neue Datenstruktur.
- Logikprogramme kennen keine statischen Typen. Typbeziehungen können in Logik ausgedrückt werden.
- In Logikprogrammen spielen Listen eine zentrale Rolle als Datenstruktur.
- Datenstrukturen können für symbolische Ausdrücke stehen. Die Unifikation stellt dabei eine Art von Mustererkennung dar.

Prolog als möglichst effiziente Realisierung der Logikprogrammierung hat eine ganze Reihe von Einschränkungen gegenüber der formalen Logik. Diese werden in der Regel durch eingebaute Mechanismen oder durch besondere Vorgehensweise umgangen.

3.2 Das Ablaufmodell von Prolog

Zusammenfassend soll der Ablauf nochmals kurz dargestellt werden.

1. Der Ablauf beginnt mit der Anfrage, einer Liste von Literalen.
2. Die Anfrageliste wird von links nach rechts abgearbeitet.
3. Die „Ausführung“ eines Literals besteht darin, dass das linkeste Literal mit dem Kopf einer Regel unifiziert wird. Dabei wird die im Programm an weitesten oben stehende unifizierbare Regel ausgewählt Die dabei vorgenommen Variablenbindungen sind bis auf ein eventuell späteres Backtracking unveränderlich. Fuer die weitere Ausführung merkt sich Prolog, wenn nötig diesen Auswahlpunkt und fährt mit der Berechnung fort, nachdem es zu nächst den Körper der verwendeten Regel der Anfrageliste voranstellt. Dieser Schritt heißt *Resolution*
4. Wenn die Anfrageliste leer ist, war die Anfrage erfolgreich. Der Prologinterpreter gibt die Variablen der ursprünglichen (top level) Anfrage aus.
5. Wenn sich im 2. Punkt keine geeignete Regel findet, wird zum letzten Auswahlpunkt zurückgesprungen. Dabei werden alle seither vorgenommen Variablenbedingungen rückgängig gemacht. Und der Punkt 2 wird mit der nächsten möglichen Regel fortgesetzt. Diesen Vorgang nennt man *Backtracking*.
6. Wenn es grundsätzlich keine Möglichkeit der erfolgreichen Fortsetzung mit Backtracking gibt, ist das Programm gescheitert.
7. Ein Cut löscht alle Auswahlpunkte seit und einschließlich der Festlegung der Regel in der der Cut steht.

8. Seiteneffekte, wie Ein-/Ausgabe, werden durch Backtracking nicht rückgängig gemacht.

Die Prolog Vorgehensweise von links nach rechts und von oben nach unten entspricht den üblichen Ablaufregeln einer Programmiersprache. Wie in einer prozeduralen Sprache wird ein Laufzeitstack aufgebaut, der es erlaubt nach erfolgreicher Rückkehr von einer Methode/Regel, im ursprünglichen Kontext fortzufahren. Im Unterschied zu imperativen Sprachen gibt es beim Backtracking in Prolog auch bei Misserfolg eine Rückkehr zu einem vorherigen Auswahlpunkt. Dies erfordert, dass in Prolog alle Auswahlpunkte auf einem Stack gespeichert sind. Die Optimierung eines Prolog-Programms hat deshalb auch damit zu tun, die Menge der Auswahlpunkte zu reduzieren und den Stack klein zu halten.

3.3 Endrekursion

Der folgende Abschnitt geht auf die als Endrekursion bezeichnete Optimierung rekursiver Abläufe ein. Optimierung und bestimmte Abläufe, also auch die Auffassung wie Rekursion ausgeführt wird sind aber kein Bestandteil der Logik. Sie sind vielmehr ein Problem der Umsetzung logischer Probleme in einen imperativen Ablauf. Diese Umsetzung wird durch den Compiler vorgenommen. Um ein effizient ausführbares Programm zu haben, muss der Programmierer jedoch einige Aspekte dieser Umsetzung kennen und beachten. Dies soll im Folgenden besprochen werden.

Iteration mittels While-Schleife ist eine typisch imperative Technik. Sie ist daher auch direkt in den Ablauf der grundlegenden Ausführungsumgebung umsetzbar.

Die Logikprogrammierung (ebenso wie die funktionale Programmierung) kennt als grundlegendes Wiederholungskonstrukt die Rekursion. Rekursive Zusammenhänge lassen sich nämlich direkt formulieren und verstehen, ohne an einen Ablauf zu denken. Die effiziente Umsetzung der Rekursion ist dafür komplizierter als die der Iteration.

Schauen wir uns mal wieder das klassische Beispiel der Fakultätsfunktion an:

```
% fak(N, N_Fak)
fak(0, 1).
fak(N, N_Fak):-
    N > 0,
    N1 is N - 1,
    fak(N1, N1_Fak),
    N_Fak is N * N1_Fak.
```

Funktionale Lösungen sind für solche Aufgaben leichter zu lesen. Aber immerhin, erkennt man in der letzten Zeile die Grundregel der Fakultätsfunktion $n! = n(n-1)!$.

Eine Besonderheit der Rekursion besteht darin, dass jeder Aufruf über einen eigenen Satz von lokalen Variablen verfügt. Dies macht die Stärke der Rekursion aus (denken Sie an Baumalgorithmen). Gleichzeitig erfordert die Verwaltung der Variablen in Stackframes, die Bereitstellung von Übergabeparametern. Auch der Sprung in eine Funktionen hinein und nachher wieder zurück benötigt einen

zusätzlichen Aufwand. Daher gilt die Rekursion als weniger effizient als die Iteration. Selbst wenn dieser (meist sehr geringe) Zusatzaufwand vernachlässigt werden kann, bleibt immer noch der zusätzliche Speicheraufwand, der in ungünstigen Fällen sogar zum Programmabruch durch „stack overflow“ führen kann.

Ein Beispiel, das sich mit herkömmlicher Rekursion garantiert nicht direkt ohne Optimierung lösen lässt, ist die für Serveranwendungen typische Endlosschleife:

```
// Endlosschleife als while
void aufgabenErledigen() {
    while (true) {
        Request r = leseAnforderung();
        bearbeite(r);
    }
}

// Endlosschleife mittels Rekursion
void aufgabenErledigen() {
    Request r = leseAnforderung();
    bearbeite(r);
    aufgabenErledigen();
}
```

Stören Sie sich nicht daran, dass die rekursive Form ungewohnt aussieht. Das liegt nur an der imperativen Denkweise. Die rekursive Fassung sagt nur, dass man nach der Erledigung einer Aufgabe weitere Aufgaben erledigen muss. Klingt gut. Der Haken ist nur, dass bei der Endlosschleife ein ins Unendliche wachsender Stack entstehen kann,

In Wirklichkeit ist dieser ”Nachteil”der Rekursion aber hausgemacht! Die Compiler imperativer Sprachen – und damit meine ich auch Java – behandeln Rekursion absolut stiefmütterlich und übersetzen rekursive Programme einfach in oft unnötig ineffizienten Code.

Schauen wir uns nämlich das Server-Beispiel nochmals an. Wer sagt denn, dass der Compiler für die Anweisung `aufgabenErledigen();` wirklich einen rekursiven Funktionsaufruf auf der Ebene der Zielmaschine (virtuelle Maschine, Maschinensprache) erzeugen soll? Das ist überhaupt nicht nötig. Es genügt vielmehr, wenn er hier einen einfachen Sprung zur ersten Anweisung generiert. Wirklich moderne Compiler tun das.

Mit Recht werden Sie einwenden, dass man so nicht jede Rekursion weg bekommt. Die oben angegebene Fakultätsfunktions lässt sich nicht ganz so einfach lösen. Das liegt daran, dass wir nach dem rekursiven Aufruf auf die vorher bestimmten lokalen Variablen (N1, N) zugreifen. Diese *müssen* gespeichert werden. Es gibt für den Compiler keinen einfach erkennbaren besseren Weg als die echte Rekursion.[1]

Aber es gibt halt auch die anderen Fälle, in denen eine einfache Optimierung möglich ist. Diese Fälle sind dadurch gekennzeichnet, dass der rekursive Aufruf die letzte Aktion der Funktion ist. Dann ist es niemals nötig, die lokalen Variablen für später aufzubewahren.

Definition:

[1] Der optimierende GNU-C Compiler erzeugt auch aus der rekursiven Fakultätsfunktion ein optimales iteratives Maschinenprogramm.

> *Eine Funktion / Prädikat / Methode ist **endrekursiv** wenn nach einem rekursiven Aufruf keine weitere Aktion erfolgt. Endrekursive Funktionen können durch den Compiler wie eine Iteration behandelt werden. In Prolog setzt die Optimierung der Endrekursion voraus, dass der Programmablauf deterministisch ist und kein Backtrackung stattfinden kann.*[2]

Die bisherige Diskussion zeigt, dass es wohl einige rekursive Programme gibt, die (zufällig) effizient übersetzt werden können und andere nicht. Das ist nur die halbe Wahrheit. In Wirklichkeit lassen sich sehr viele Programme in eine endrekursive Form umschreiben. Dies sind letztlich genau diejenigen, die man auch leicht iterativ schreiben könnte. Endrekursion und Iteration sind sich letztlich so ähnlich, dass man in der Praxis die endrekursive Formulierung als iterativ bezeichnet.

Wir werden auf die Technik der Umwandlung in die endrekursive Form später bei der funktionalen Programmierung noch eingehen. Hier aber auch schon mal ein Kochrezept:

Merksatz:

> *Wenn man einen rekursiven Ablauf in einen endrekursiven Ablauf umwandeln will, muss man dafür sorgen, dass die gesamte Berechnung jeweils vor dem Aufruf erfolgt. Es wird sozusagen auf dem „Hinweg" gerechnet. Sobald die Abbruchbedingung zutrifft, muss das Endergebnis feststehen. Man erreicht dies dadurch, dass man eine (oder mehrere) Akkumulatorvariablen einführt, in denen das Ergebnis nach und nach aufgebaut wird. Vor dem ersten Aufruf müssen diese Variablen natürlich geeignet initialisiert werden. Dies geschieht in der Regel durch eine eigene Funktion.*

Schauen Sie sich als Beispiel die Fakultätsfunktion an.

```
% fak(N, N_Fak)
% N_Fak = N!
fak(N, N_Fak):-
    N >= 0,
    fak(N, 1, N_Fak).

% fak(X, SoFar, N_Fak)
% N_Fak = X! * SoFar
fak(0, N_Fak, N_Fak).
fak(X, SoFar, N_Fak):-
    X > 0,
    X1 is X - 1,
    SoFar1 is X * SoFar,
    fak(X1, SoFar1, N_Fak).
```

Der Übersetzer von SWI-Prolog löst dieses Problem tatsächlich ohne einen Stack aufzubauen. Ähnlich verfahren auch praktisch alle Programmiersprachen, die die funktionale Programmierung unterstützen.

[2] In der Objektorientierung ist Endrekursion nur gegeben, wenn der Compiler die Rekursion erkennen kann, d.h. wenn man mit früher Bindung auskommt.

3.4 Algebraische Datentypen in Prolog

Neben den elementaren Typen wie Zahlen und Atomen kennt Prolog Strukturen zur Konstruktion zusammengesetzter Typen.

Als dynamisch getypte Sprache kennt Edinburgh-Prolog keine Typbeschreibung[3]. Die Struktur komplexer Typen wird ausschließlich durch die für sie definierten Prädikate deutlich. Es dient demnach der Klarheit eigene Typprädikate zu definieren, die diese Struktur genauer beschreiben.

Definition:

Ein algebraischer Datentyp definiert strukturierte Typen auf der Basis von Grundtypen. Die Typkonstruktion verwendet die Grundoperationen Produkt *und* Summe. *Die Produktoperation kann man durch einen Konstruktor beschreiben, der mehrere Elemente von anderen Datentypen als kartesisches Produkt zusammenfasst. Die Summenoperation fasst unterschiedliche Typen zu der Einheit eines Obertyps zusammen.*

Diese Definition wird im Rahmen der funktionalen Programmierung nochmals aufgegriffen und vertieft verdeutlicht. Hier soll die Verwendung in Prolog gezeigt werden.

Nehmen wir als Beispiel einen Binärbaum zur Darstellung von arithmetischen Ausdrücken. Wir können ihn wie folgt definieren:

```
Expr = Number | +(Expr, Expr) | -(Expr, Expr) | *(Expr, Expr)
     | /(Expr, Expr).
```

Ich habe hier als Namen der Typkonstruktoren die arithmetischen Operatorzeichen genutzt. Wie Sie gleich sehen werden, erhalten wir damit automatisch einen Parser, der arithmetische Ausdrücke in den zugehörigen Baum verwandelt. Diese Tatsache ist aber nicht wesentlich. Wir hätten ebenso gut symbolische Namen, wie `add(Expr,Expr)` verwenden können.

Ein Datentyp der, wie hier `Expr`, auf sich selbst Bezug nimmt, heißt auch *rekursiver Datentyp*.

Die obige Formel (sie liest sich wie eine Syntax) ist kein Prolog. Die Verwendung in Prolog soll hier durch eine formale Typprüfung und durch Evaluierungsregeln beschrieben werden.

```
expr(Number):- number(Number).
expr(+(Left,Right)):- expr(Left), expr(Right).
expr(-(Left,Right)):- expr(Left), expr(Right).
expr(*(Left,Right)):- expr(Left), expr(Right).
expr(/(Left,Right)):- expr(Left), expr(Right).

eval(Number, Number):-
    number(Number).
eval(+(Left, Right), Value):-
    eval(Left, V1),
    eval(Right, V2),
    Value is V1 + V2.
```

[3] Es gibt aber getypte Prolog Systeme, wie z.B. Turbo-Prolog.

```
eval(-(Left, Right), Value):-
    eval(Left, V1),
    eval(Right, V2),
    Value is V1 - V2.
eval(*(Left, Right), Value):-
    eval(Left, V1),
    eval(Right, V2),
    Value is V1 * V2.
eval(/(Left, Right), Value):-
    eval(Left, V1),
    eval(Right, V2),
    Value is V1 / V2.
```

Der entscheidende Punkt, den Sie hier mitnehmen sollen, ist die Beobachtung, dass die Eval-Regeln genauso aufgebaut sind, wie die Typregeln. Der Typ ist sozusagen eine Vorlage für das Programm.

Wie sieht nun die Anwendung aus?

Nun, die ganz reguläre Antwort lautet:

```
?- eval(expr(+(3, expr(*(4,5))), X).
```

Solch, kompliziert aussehende Datenstrukturen werden in der Regel durch andere Programmteile konstruiert. In Prolog ist diese Übersetzung schon in den Interpreter eingebaut. Wir können also einfacher auch schreiben:

```
? eval(3 + 4*5, X).
```

Der Prolog-Parser ist sozusagen das Programm, das die syntaktische Struktur kennt und den Baum aufbaut (das gilt natürlich nur für dem Parser bekannte oder bekannt gemachte Operatoren).

In unserm Fall hätte wir die Regeln selbst auch einfacher schreiben können, wie z.B.:

```
expr(Left + Right):- ..
eval(Left + Right, Value):- ..
```

Auch die im nächsten Abschnitt beschriebenen Listen stellen die Realisierung eines algebraischen Datentyps dar. Wegen der großen Bedeutung von Listen, ist die syntaktische Unterstützung besonders ausgeprägt, so dass die formal algebraische Struktur nicht mehr unmittelbar erkennbar ist.

Ganz abstrakt könnte man sie aber so beschreiben:

```
Liste = nil | cons(Any, Liste)
```

Eine Liste ist entweder leer, oder sie ist ein beliebiges Listenelement, gefolgt von einer Liste.

Der wichtige Punkt, der hier festzuhalten wird und der im folgenden mehrfach vertieft wird ist die folgende Feststellung. Das gilt dann auch für die funktionale Programmierung.

Merksatz:

Programme auf algebraischen Datenstrukturen basieren auf der Fallunterscheidung durch Mustererkennung.

In der Objektorientierung tritt an diese Stelle häufig die Methodenauswahl durch späte Bindung.

3.5 Listenverarbeitung

Die Listenverarbeitung mit Prolog wird hier vor allem auch deshalb besprochen, weil sie deutliche Auswirkungen auf moderne Programmiersprachen hat (Erlang, Scala). Dabei werden gleichzeitig grundsätzliche Vorgehensweise im Umgang mit Datenstrukturen deutlich.

3.5.1 Grundlagen

Prolog hat eine extrem einfache Syntax. Dies macht die Formulierung von manchen Sachverhalten etwas formal und kompliziert. Da in Prolog sehr viele Sachverhalte durch Listen, als der grundlegenden Datenstruktur für eine unbekannte Anzahl von Datenelemente, ausgedrückt werden, wurde für Listen eine besondere Syntax eigenführt. Prolog enthält Listenliterale und Ausdrücke zum Zerlegen und zum Aufbau von Listen. Natürlich enthält Prolog auch eine ganze Reihe von vordefinierten Listenfunktionen.

Im Unterschied zum prozeduralen Umgang mit Zahlen sind die meisten Listenoperationen logische Verknüpfungen. Ein und dasselbe Prädiktat, kann mitunter unterschiedlich angewendet werden.

Listenliterale sind in Prolog in eckige Klammern `[` und `]` eingeschlossen. Die Listenelemente sind durch Komma getrennt. Listen können ihrerseits beliebige Prologelemente, auch ungebundene Variablen, enthalten.

Beispiele für Listenliterale sind:

```
[]                  /* leere Liste */
[1, 2, 3]           /* Liste mit Zahlen */
[a, b, c]           /* Liste mit Symbolen */
[X, 1, a, [1, 2]]   /* gemischte Liste */
```

Prolog enthält eine weitere Schreibweise für Listen, nach der Listen in ein (oder mehrere Anfangselemente und in eine Restliste zerlegt werden können.

Die Liste `[ 1, 2, 3]` lässt sich so auf verschiedene Art und Weise schreiben.

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

Der durchbrochene Strich `|` trennt den Bereich der Aufzählung von der Liste der restlichen Elemente ab. Zusammen mit dem mächtigen Unifikationsmechanismus lassen sich damit alle Listenoperationen nachbilden.

Um die Notation zu verdeutlichen, sollen die Grundoperationen nochmals als Regeln formuliert werden.[4]

```
% head(Liste, ErstesElement)
head([First|_], First).

% tail(Liste, Rest)
tail([_|Rest], Rest]).

% prepend(Element, Liste, NeueListe)
prepend(Element, Liste, [Element | Liste]).
```

Als nächstes Beispiel soll die (allerdings schon vordefinierte) Operation zur Berechnung der Anzahl der Elemente einer Liste dienen.

```
% length(Liste, Anzahl)
% Anzahl ist die Anzahl der Listenelemente
length([], 0).
length([_ | Xs], Anzahl) :-
    length(Xs, N),
    Anzahl is N + 1.
```

Umgangssprachlich ausgedrückt sagt dieses Programm „Die Länge der leeren Liste ist 0. Die Länge einer nichtleeren Liste ist um 1 größer als die Länge der Liste, bei der das erste Element entfernt wurde.

Dies lässt sich auch endrekursiv formulieren:

```
% length(Liste, Anzahl)
% Anzahl ist die Anzahl der Listenelemente
length(Liste, Anzahl):-
    length(Liste, 0, Anzahl).

% length(Liste, N, Anzahl)
% Anzahl ist die Anzahl der Listenelemente + N
length([], Anzahl, Anzahl).
length([_|Rest], AnzahlBisher, Anzahl):-
    AnzahlNeu is AnzahlBisher + 1,
    length(Rest, AnzahlNeu, Anzahl).
```

Die Längenberechnung ist nun aber nicht vollständig logisch, da sie ja mit arithmetischen Berechnungen verknüpft ist. Mit den Möglichkeiten eines Prolog-Systems lassen sich auch diese Einschränkungen umgehen. d.h. das vordefinierte Prädikat `length` kann auch zum Aufbau einer n-elementigen Liste verwendet werden.

Kommen wir zu grundlegenden „logischen“ Prädikaten.

```
% member(Element, Liste}
% Element ist in der Liste enthalten.
%    Das Kopfelement ist in der Liste.
member(Element, [Element| _]).
%    Ein Element der Restliste ist in der Liste.
member(Element, [_ | Rest]):-
    member(Element, Rest).
```

[4] In Prolog macht man das allerdings nicht, da die Grundoperationen schon so einfach sind.

```
% append(Liste1, Liste2, Liste12)
%  in Liste12 folgt Liste2 auf Liste1
%      Haengt man eine Liste Bs an die leere Liste, erhaelt
%      man die Liste Bs.
append([], Bs, Bs).
%      Eine Liste Bs angehaengt an eine
%      nichtleere Liste A hat als Anfangselement das
%      erste Element von A und als Rest die zusammengesetzte
%      Liste aus dem Rest von A und der Liste Bs.
append([A|As], Bs, [A|Cs]):-
    append(As, Bs, Cs).

% select(Element, Liste, RestListe)
%  Element ist Element der Liste und die RestListe ist
%  gleich Liste ohne dieses Element.
select(Element, [Element| Rest], Rest).
select(Element, [Anfang | Rest1]. [Anfang|Rest2]):-
    select(Element, Rest1, Rest2).
```

Die Flexibilität der nichtdeterministischen Auswahl von Lösungen in der Logikprogrammierung führt zu sehr vielseitigen Verwendungsmöglichkeiten für diese Prädikate. Dies wird unten kurz angesprochen. Hier soll aber mal gezeigt werden, wie eine Liste definiert werden kann, die aus einer beliebigen Anordnung der drei Zahlen 1, 2 und 3 besteht (Permutation).[5] Zunächst werden durch `member` eine Liste definiert. die die Zahlen 1 bis 3 in beliebiger Reihenfolge enthält. Anschließend folgt eine alternative Formulierung mit `select`. Schließlich ist die typische allgemeine Definition einer Permutation angegeben.

```
?- Liste=[_,_,_],
   member(1, Liste),
   member(2, Liste),
   member(3, Liste).

?- select(1, Liste, Liste1),
   select(2, Liste1, Liste2),
   select(3, Liste2, []).

% permutation(As, Bs)
%   Die As sind eine Permutation der Bs (und umgekehrt)
permutation([], []).
permutation(As, [A|Bs]):-
    select(A, As, A1s),
    permutation(A1s, Bs).
```

Abschließend wollen wir festhalten, dass der durchbrochene Strich `|` sowohl zum Zerlegen der Liste in Anfangselement und Restliste als auch zur Konstruktion einer Liste aus einem ersten und weiteren Elemente besteht (Cons-Operation genannt). Wir werden diese Doppelfunktion auch bei der funktionalen Programmierung wiederfinden. Auch dort sind dies die grundlegenden Listenoperationen.

[5] `permutation` ist in SWI-Prolog schon vordefiniert.

3.5.2 Prädikate höherer Ordnung

Einleitend hatte ich bemerkt, dass Prolog sich auf die Prädikatenlogik 1. Ordnung beschränkt. Das ist auch korrekt, soweit die *logischen* Grundlagen von Prolog betroffen sind.

Allerdings gibt es in paar eingebaute Prädikate, die ihrerseits Prädikate als Argumente enthalten. Ich will hier nur drei der wichtigeren nennen:

- `bagof(X, Anfrage, Xs)`
  `bagof` ist erfüllt, wenn `Xs` die Liste der Substitutionen für `X` ist, mit denen die `Anfrage` erfüllt ist. `Xs` ist demnach die Menge aller Lösungen.
- `setof(X, Anfrage, Xs)`
  Wie `bagof`, nur dass doppelt vorkommende Lösungen entfernt sind (*set* = *Menge*).
- `call(P, X1, X2, ...)` „Ruft" das Prädikat `P` mit den Argumenten `X1` bis `Xn` auf.

`bagof` und `setof` werden unten bei der Lösungssuche besprochen (siehe Abschnitt 3.6).

Mittels `call` will ich hier ein paar Prädikate formulieren, die den Übergang zur funktionalen Programmierung aufzeigen.

Aus dem Bereich Datenbanken ist das Akronym CRUD = (Create, Read, Update, Delete) bekannt. Es steht für die elementaren Operationen, die man mit Daten durchführen kann. `Elementar` bedeutet dabei aber auch, dass diese Operationen immer nur für einen einzigen Datensatz stehen. Wenn größere Datenmengen bearbeitet werden müssen, so muss dafür gesorgt werden, dass geeignet durch die Datenmenge iteriert wird. Das kennen Sie auch aus der Programmierung in Java. Sie wiederholen die Operationen auf den Datenelementen, durch die Programmierung von Schleifen (For-Anweisung, While-Anweisung) oder durch Iteration.

Wir müssen uns aber klarmachen, dass dies eine elementare Vorgehensweise ist, die nicht dem üblichen Sprachgebrauch entspricht: „addiere alle Zahle der Liste", "wie lauten die ungeraden Zahlen in der Liste?" usw. Der allgemeine Sprachgebrauch meint nämlich, dass wir eine Operation oder eine Frage direkt auf alle Elemente einer Datenstruktur anwenden.

`bagof` und `setof` sind bereits entsprechend vorformulierte Prädikate dieser Art („Wie lauten alle Lösungen?"). Das Prädikat `call` ermöglicht uns weitere solche Prädikate höherer Ordnung zu definieren.

Hier seien drei solcher Prädikate beispielhaft dargestellt:

```
% filter(Xs, P, Ys)
%   Ys ist die Liste aller Elemente X aus Xs
%   fuer die P(X) erfuellt ist.
filter([], _, []).
filter([X|Xs], P, [X|Ys]):-
        call(P, X), !,
        filter(Xs, P, Ys).
filter([_|Xs], P, Ys):-
        filter(Xs, P, Ys).
```

```
% map(Xs, P, Ys)
%   Ys ist die Liste der Y-Werte
%   aus P(X, Y) mit X aus Xs
%   (die Liste von Funktionsresultaten)
map([], _, []).
map([X|Xs], P, [Y|Ys]):-
        call(P, X, Y), !,
        map(Xs, P, Ys).

% reduce(Xs, P, R)
%    R ergibt sich, wenn aufeinanderfolgnde
%    Elemente X,Y aus Xs mit P(X, Y, Z) zusammengefasst
%    werden koennen.
reduce([X|Xs], P, R):-
        reduce(Xs, P, X, R).
reduce([], _, A, A).
reduce([X|Xs], P, A, R):-
        call(P, A, X, A1), !,
        reduce(Xs, P, A1, R).
```

Prolog unterstützt die funktionale Programmierung nicht besonders. Dementsprechend ist die Anwendung dieser Konzepte nicht immer ganz einfach. Insbesondere müssen wir die verwendeten Fragestellungen explizit als Prädikate formulieren:

```
summe(X,Y,Z):- Z is X + Y.
product(X, Y, Z):- Z is X * Y.
quadrat(X, Q):- Q is X * X.
ungerade(X):- 1 is X mod 2.
```

Aber wenn wir das einmal haben, können wir einige Ausdrücke sehr elegant hinschreiben:

```
% Summe der Quadrate aller ungeraden Zahlen von 1 bis 100
?- bagof(X, between(1, 100, X), Xs),
   filter(Xs, ungerade, Us),
   map(Us, quadrat, Qs),
   reduce(Qs, summe, S).

fakultaet(N, F):-
   bagof(X, between(1, N, X), Xs),
   reduce(Xs, product, F).
```

Wir wollen es nicht übertreiben – Prolog ist keine funktionale Sprache. Mir geht es hier nur darum schon einmal aufzuzeigen, dass es mittels Prädikaten (und Funktionen) höherer Ordnung möglich ist, Abstraktionen zu definieren, die man in prozeduralen Sprachen (bisher) so nicht kennt.

3.6 Lösungssuche

Die eigentliche Stärke von Prolog liegt in der symbolischen Verarbeitung und in der Fähigkeit, selbst Lösungen auf eine Frage zu finden.

3.6.1 Tiefensuche in Prolog

Noch einmal: Bei der Beantwortung einer Anfrage trifft Prolog zwei Festlegungen hinsichtlich der Reihenfolge der versuchten Resolutionen:

1. Bei der Auswahl der Teilziele einer Zielanfrage geht Prolog stets von links nach rechts vor (goal order).
2. Bei der Auswahl der Regeln geht Prolog immer von oben nach unten vor (rule order).
3. Nach erfolgreicher Resolution eines Zielliterals mit dem Kopf einer Regel, setzt Prolog den Körper der Regel an den Anfang der Zielanfrage.

In der Kombination führen diese drei Punkte dazu, dass der Suchbaum eines Problems in Tiefensuche durchlaufen wird.

Tiefensuche hat als Suchstrategie große Vorteile. Sie ist laufzeit- und speichereffizient. Sie hat darüber hinaus den Vorteil, gut nachvollziehbar zu sein. Insbesondere stimmt die Prolog-Suchstrategie auch mit dem erwarteten prozeduralen Ablauf überein.

Tiefensuche hat aber auch Nachteile. Sie ist keine *vollständige* Suchstrategie.

Definition:
*Ein Beweisverfahren oder eine Suchstrategie sind **vollständig**, wenn sie jeden endlichen Beweis in endlich vielen Schritten finden.*

Die Tiefensuche findet nicht immer eine vorhandene Lösung. Die Lösung einer Anfrage ist endlich viele Ableitungsschritte von dem Wurzelknoten des Suchbaums entfernt. Die Tiefensuche durchläuft den Suchbaum von links nach rechts. Wenn einer der links vom Lösungsweg liegender Teilbäume unendlich lang ist, wird mittels Tiefensuche die Lösung niemals gefunden. Die Breitensuche ist dagegen ein Beispiel für eine vollständige Suchstrategie.

Als Beleg für die Unvollständigkeit von Prolog nehmen Sie bitte einmal das Beispiel von Seite 52. Dieses Mal habe ich nur die Reihenfolge der Literale vertauscht.

```
?- member(1, Liste),
   member(2, Liste),
   member(3, Liste),
   Liste = [_,_,_].
```

Prolog findet zwar noch die einfache Lösung `[1,2,3]`, bei der Suche nach weiteren Lösungen gerät es jedoch in einen endlosen Ablauf. Versuchen Sie nachzuvollziehen, woran das liegt.

Ein ähnliches Problem taucht auch bei vielen anderen Problemen auf. Das folgende Prädikat definiert die allgemeine Graphsuche:

```
% weg(Start, Ziel)
% Es gibt einen Weg von Start zu Ziel.
% Der Graph darf keine Kreise haben!
```

```
%    Wenn Ziel = Start ist, gib es einen Weg
weg(Ziel, Ziel).
%    Es gibt einen Weg von Knoten zum Ziel,
%    wenn es eine Kante von Knoten zu Nachbar
%    und einen Weg von Nachbar zum Ziel gibt.
weg(Knoten, Ziel):-
    kante(Knoten, Nachbar),
    weg(Nachbar, Ziel).
```

Dies ist eine logische Beschreibung für die Existenz eine Weges in einem Graphen, der durch eine Reihe von `kante`-Aussagen beschrieben ist. Dieses Programm funktioniert in Prolog aber nur, wenn der Graph keine Kreise enthält. Sonst gerät man in eine endlose Rekursion.

Man kann eine solches Suche leicht verbessern, indem man nachhält, welche Knoten schon besucht wurden. Wir sagen nur: Wenn es einen Weg gibt, dann gibt es auch einen solchen, der keine Kreise enthält.

```
weg(Start, Ziel):- weg (Start, [Start], Ziel).

% weg(Start, Besucht, Ziel).
% Es gibt einen Weg von Start zu Ziel unter Vermeidung der
% besuchten Knoten.
weg(Ziel, _, Ziel).
weg(Knoten, Besucht, Ziel):-
    kante(Knoten, Nachbar),
    notmember(Nachbar, Besucht),
    weg(Nachbar, [Nachbar| Besucht], Ziel).

% notmember(X, Liste)
% X ist nicht in Liste enthalten.
notmember(X, Liste) :- \+ member(X, Liste).
```

Schließlich lässt sich das Prädikat so ausbauen, dass es am Schluss auch noch den gefundenen Weg mitteilt. Dies sei Ihnen zur Übung überlassen.

3.6.2 Lösungssuche durch systematisches Ausprobieren

In diesem Beispiel geht es um die systematische Suche nach einer Lösung durch Ausprobieren aller Möglichkeiten. Als Beispspiel soll eine vollständige Zahl gesucht werden, das ist eine Zahl bei der die Summe der Teiler gleich der Zahl selbst ist. Die kleinste solche Zahl ist 6. Gibt es weitere?

```
% vollkommen(Zahl).
% Zahl ist eine vollkommene Zahl.
vollkommen(Zahl):-
    teilerliste(Zahl, Teilerliste),
    sumlist(Teilerliste, Zahl).

% teilerliste(Zahl, Liste)
% Teilerliste enthaelt alle Teiler von Zahl
teilerliste(Zahl, Teilerliste):-
    bagof(X, teiler(X, Zahl), Teilerliste).

% teiler(Teiler, Zahl)
% Teiler ist ein Teiler von Zahl
teiler(Teiler, Zahl):-
```

```
    Limit is Zahl / 2,
    between(1, Limit, Teiler),
    0 is Zahl mod Teiler.

?- between(Zahl), 1, 10000), vollkommen(Zahl).
```

Versuchen Sie wieder selbst dieses Beispiel zu verstehen. Eine Anmerkung zu `bagof`: Dieses vordefinierte Prädikat `bagof` findet die Liste aller Lösungen zu einem logischen Ausdruck. Das erste Argument ist eine freie Variable, die eine Lösung aufnehmen kann. Das dritte Element ist die Liste aller Lösungen. Das mittlere Argument ist der logische Ausdruck, der mit der Variablen erfüllt sein soll.

Sie werden nicht viele vollständige Zahlen finden. Mit ziemlicher Sicherheit wird keine ungerade Zahl dabei sein. Wenn doch, dann haben Sie bestimmt was falsch gemacht! Man kennt nämlich bisher keine ungerade vollkommene Zahl. Allerdings weiß niemand, warum das so ist (oder ob es nicht doch eine ungerade vollkommene Zahl gibt). Dagegen hat bereits im 17. Jhd. Leonhard Euler einen effizienten Algorithmus zum Auffinden *gerader* vollkommener Zahlen entdeckt. Er konnte dabei auf den Vorarbeiten von Generationen von Mathematikern (Euklid, Ibn al-Haythan, Mersenne) aufbauen.[6]

Wenn Sie zu mathematischen Experimenten neigen, können Sie mit ähnlicher Technik weitere Vermutungen überprüfen. Die *Goldbach-Vermutung* behauptet, dass sich jede gerade Zahl größer 2 als Summe zweier Primzahlen schreiben lässt. Wirklich jede?

3.6.3 Kombinatorische Suche

Die Stärke von Prolog liegt im Ausprobieren einer Vielzahl von Möglichkeiten. In der einfachsten Form, verwendet man hierzu einfach die Permutation.

Hier soll am Beispiel des Sortierens die kombinatorische Suche verdeutlicht werden. Zwar sind hier bessere Algorithmen bekannt. Andererseits ist das Sortieren aber überschaubar und auch einfacher verständlich als andere nur durch Suche lösbare Probleme. Es geht hier nur darum, dass die grundsätzlichen Eigenschaften der kombinatorischen Suche deutlich werden.

Zunächst eine ganz primitive Implementierung:

```
% sorted(Unsorted, Sorted)
% Sorted ist Unsorted als sortierte Liste
% (einfach ausprobieren)
sorted(Unsorted, Sorted):-
    permutation(Unsorted, Sorted),
    ordered(Sorted).

% odered(Xs)
% die Zahlen in Xs stehen in aufsteigender Reihenfolge
ordered([]).
ordered([_]).
ordered([X,Y|Xs]):-
    X =< Y,
```

[6] Mit diesem Prolog-Programm finden Sie auch nicht mehr vollkommene Zahlen, also schon Euklid vor 2000 Jahren bekannt waren. Das ist Fortschritt!

```
    ordered([Y|Xs]).
```

Natürlich ist dieser Algorithmus nicht effizient (er ist in $O(e^N)$). Das Beispiel zeigt aber die Fähigkeit von Prolog zunächst Lösungsvorschläge zu erzeugen, sie anschließend zu überprüfen und das solange zu wiederholen, bis eine Lösung gefunden wurde.[7]

Das beschriebene Vorgehen nennt sich auch *generate and test*, zu deutsch also „erzeuge und überprüfe". Es besteht grundsätzlich aus einem Prädikat, das Lösungsvorschläge erzeugt (Generator) und einem Prädikat, das die Zulässigkeit der Lösung überprüft (Test). Im Folgenden wird dargestellt, wie man fast immer einer effizienteren Form des Verfahrens kommen kann.

Anders als beim Sortieren führt in vielen anderen Fällen kein Weg an kombinatorischem Ausprobieren vorbei. Also kommt es darauf an, soweit wie möglich die Effizienz der Suche zu erhöhen. Ein wichtiger Schritt auf diesem Weg besteht darin, Sackgassen möglichst früh zu erkennen und zu vermeiden. Dies erreicht man indem man beim Erzeugen des Lösungsvorschlags bei jedem einzelnen Schritt prüft, ob er zum Ziel führen kann.

In dem Sortierbeispiel wird anstelle des Prädikats `permutation`, dass stets einen kompletten Lösungsvorschlag erzeugt, das feinkörnigere `select` verwendet. Dabei wird stets eine weitere Zahl (willkürlich) für die nächste Position ausgewählt und es wird dann direkt geprüft, ob nach der Auswahl die Liste immer noch sortiert ist.

```
% sorted(Unsorted, Sorted)
% Sorted ist Unsorted als sortierte Liste
sorted(Unsorted, Sorted):-
    sorted(Unsorted, [], Sorted).

% sorted(Unsorted, AlreadySorted, Sorted)
% In Unsorted bedinden sich die noch nicht einsortierten
   Zahlen.
% Already Sortiert enthaelt bereits sortierte Elemente
% Sorted steht fuer die Sortierte Gesamtliste.
sorted([], Sorted, Sorted).
sorted(Unsorted, AlreadySorted, Sorted):-
    select(X, Unsorted, UnsortedRest),
    ordered([X|AlreadySorted]),
    sorted(UnsortedRest, [X|AlreadySorted], Sorted).
```

Versuchen Sie das Beispiel zu verstehen, auch wenn es immer noch kein effizienter Sortieralgorithmus ist. Immerhin ist es eine erhebliche Verbesserung.

Eine konsequente Verbesserung dieses Algorithmus führt direkt zu dem bekannten Algorithmus der Direkten Auswahl (selection sort). Dabei wird direkt die richtige Zahl ausgewählt, so dass alle Irrwege vermieden werden können. Wegen der korrekten Auswahl kann auch der Test auf die richtige Reihenfolge entfallen.

```
% selection sort
sorted([], []).
sorted(UnSorted, [Min|SortedRest]):-
```

[7] Dieser Lösungsweg erinnert an das planlose Herumprobieren von Programmieranfängern. Der Aufwand von ungeplanten Aktionen ist einfach immer riesig.

```
    selectMin(Min, UnSorted, Rest),
    sorted(Rest, SortedRest).

% selectMin(Min, Xs, Ys)
%   Min ist das kleinste Elemente der Xs.
%   Ys sind die restlichen Elemente
selectMin(Min, [Min], []).
selectMin(Min, [X|Xs], [Y|Ys]):-
    selectMin(Min0, Xs, Ys),
    sort2(X, Min0, Min, Y).

% sortiert zwei Argumente.
sort2(X, Y, X, Y):- X =< Y.
sort2(X, Y, Y, X):- X > Y.
```

Natürlich kann man auch diesen Algorithmus endrekursiv schreiben. Besser wäre dann allerdings schon der Quicksort. Das ist aber jetzt nicht das Thema.

Eine letzte Randbemerkung. Am Beispiel des Sortierens wurde gezeigt, dass effiziente Algorithmen immer besser sind als kombinatorisches Ausprobieren. Das ist grundsätzlich immer richtig. Leider ist es so, dass für sehr viele praktischen Probleme keine effizienten Algorithmen bekannt sind. Vermutlich existieren in solchen Fällen wirklich keine effizienten Algorithmen. In solchen Fällen kommt es darauf an, auf andere Verfahren auszuweichen. Manchmal ist man mit suboptimalen Lösungen zufrieden, die sich einfacher finden lassen.[8] Wenn das nicht ausreicht, führt jedoch kein Weg an der kombinatorischen Suche vorbei.

[8] Hierzu zählt auch die Klasse der Evolutionären Algorithmen,

Kapitel 4

Überblick über Scala

Im Kontext von Paradigmen der Programmierung steht Scala als Beispiel für Programmiersprachen, die die funktionale Programmierung unterstützen. Scala realisiert dieses Paradigma zwar nicht in seiner reinsten Form, aber alle wesentlichen Merkmale werden unterstützt.

Scala hat als objektorientierte Sprache viele Gemeinsamkeiten mit Java. Ich hoffe, dass dadurch das Verständnis erleichtert wird. Der Kern von Scala wird hier – soweit es für die Vorlesung nötig ist – beschrieben. Nicht notwendige Dinge werden weggelassen. Die gilt z.B. auch für die Einschränkung der Sichtbarkeit durch `protected` und `private`. In den folgenden Scala-Beispielen ist alles automatisch `public`

Viel stärker als der Java-Compiler unterstützt der Scala-Compiler den Programmierer durch eine automatische Vervollständigung des Programms. Dies gilt z.B. für das Semikolon, das in Scala praktisch immer weggelassen wird, und dies gilt auch in vielen Fällen für Typangaben. Bei allen Vorteilen für die Lesbarkeit von Scala-Programmen mag das manchmal etwas verwirren. Zum Glück betrifft dies jedoch nur die oberflächlichen Aspekte der Syntax und nicht den Kern der Konzepte. Im Zweifelsfall können Sie immer auf die vollständige Schreibweise zurückgreifen.

Dieses Kapitel beschränkt sich bewußt auf die bloße Beschreibung der Besonderheiten von Scala. Sie können auch später darauf zurückkommen.

Wenn Sie sich fragen, wozu die funktionale Programmierung gut ist, sollten Sie dieses Kapitel zunächst überspringen und zunächst im folgenden Kapitel die Gründe für die funktionale Programmierung nachlesen.

4.1 Alles ist ein Objekt

Scala ist eine streng objektorientierte Sprache. Auch Zahlen und boole'sche Werte sind Objekte. Beispiel:

```
val s: String = 17.toString();
```

Die Anweisung definiert eine unveränderliche Stringvariable `s`. Dies hätte auch kürzer geschrieben werden können:

```
val s = 17 toString
```

Es fehlen hier die Typangabe, die automatisch ermittelt wird (Typinferenz[1]), der Punkt, der in Scala genauso wie die Klammern der parameterlosen Methoden fehlen darf und es fehlt das Semikolon.[2]

Die Regel, dass Zahlen auch Objekte sind, beseitigt eine Reihe von Ungereimtheiten. Dies gilt auch für den Unterschied von Wertvariablen und Referenzvariablen, Scala übernimmt auch für Zahlen die Regel, dass alle Datentypen mit großem Anfangsbuchstaben geschrieben werden (`Int`, `Double` usw.).[3]

In Scala wird die Gleichheit durch == überprüft (die Ungleichheit mit `!=`). In der jeweiligen Klasse wird die Gleichheit, wie in Java, durch die Methode `equals` definiert.

In Java gab es den Unterschied, dass Wertdaten durch Operatoren verknüpft werden, dagegen Objekte durch Methoden. Dieser Unterschied existiert in Scala ebenfalls nicht. Objekte, also auch Zahlen, werden mit Methoden verarbeitet.

Scala hat die Regel, dass Methoden, wie +, vom Parser wie die Operatoren in Java ihren Operanden zugeordnet werden, wobei auch ihre Präzendenz und ihre Assoziativität berücksichtigt wird.

In Scala kann man den Methoden beliebiger Klassen Operatornamen zuordnen. Damit kann man zum Beispiel eine Bruchklasse schreiben, mit der sich dann genauso „rechnen“ lässt wie mit Zahlen.

Grundsätzlich nutzt Scala die Elemente von Java, also z.B. die Wertdaten. Diese erscheinen in Scala aber immer in objektorientierter Form [4]. Die grundlegenden Java-Elemente, wie Arrays und Strings, stehen in einer deutlich erweiterten Form zur Verfügung. Ebenso werden die typischen Bibliotheksklassen in vereinfachter und verbesserter Form angeboten.

Alles sind Objekte? Ja, auch Funktionen sind Objekte. Darauf wird im nächsten Kapitel ausführlicher eingegangen.

4.2 Aufbau eines Scala-Programms

Wie ein Java-Programm, so gliedert sich ein Scala-Programm in Pakete. Diese wiederum enthalten Klassen, abstrakte Klassen, *Objekte* und *Traits* (diese übernehmen die Rolle von Interfaces).

In Scala entfällt die Forderung, dass eine Datei den gleichen Namen wie die (einzige) öffentliche Klasse tragen muss.

[1] *Inferenz*: Herleitung, von *inferre*: wörtl. *hineintragen*

[2] Wenn eine Funktion bereits ohne Klammern definiert wurde, darf man allerdings beim Aufruf auch keine Klammer setzen.

[3] Bei ganz wenigen Fällen macht Scala einen Unterschied zwischen Objekten die zur Klassenfamilie `AnyVal` im Unterschied zu `AnyRef` gehören. Die Oberklasse aller Objekte ist `Any`.

[4] Der Compiler entscheidet automatisch, wann die elementaren Typen benutzt werden können und wann man um die Wrapper-Objekte nicht herumkommt

4.2.1 Pakete

Pakete werden in Scala wie in Java durch die Packetanweisung deklariert. Die Import-Anweisung dient auch hier dem Zweck der Abkürzung von Namen. Die Regeln für Import-Anweisungen sind einfacher als in Java. Import kann an beliebiger Stelle im Programm stehen und es können beliebige Anteile von Paketnamen abgekürzt werden.

Nehmen wir die Klasse `scala.collection.mutable.List`. Die folgenden Beispiele stellen unterschiedlich weitgehende Imports dar:

```
import scala.collection
val a = collection.mutable.List(1,2,3)

import scala.collection.mutable
val a = mutable.List(1,2,3)

import scala.collection._  // _ entspricht dem * von Java
val a = mutable.List(1,2,3)
```

Die Klassen des Pakets `scala` sind schon automatisch bekannt gemacht. Dies gilt auch für die Funktionen einiger Objekte. Für die Java-Methode `println` braucht kein `System.out` angegeben zu sein.

Überhaupt können grundsätzlich alle Java-Klassen benutzt werden. Das gleiche gilt grundsätzlich auch umgekehrt. Scala-Klassen und ihre Objekte können in Java-Programmteilen auftauchen. Es gibt nur da Grenzen, wo es in Java kein Gegenstück zu einem Scala-Element gibt.

4.2.2 Variablendeklarationen und Typparameter

Variablen können als Parameter von Methoden, Objekten und Klassen und als lokale oder als Instanzvariablen erscheinen. Es gibt keine statischen Variablen.[5]

Zunächst die Deklaration von einfachen Variablen. Hierbei wird unterschieden zwischen unveränderlichen Variablen und veränderlichen Variablen. Unveränderliche Variablen (sie sind in Scala dar Normalfall) werden durch das Schlüsselwort `val` bezeichnet. Die veränderlichen Variablen sind durch `var` als solche zu erkennen. Auf die Kennzeichnung der Variablenart folgt der Name. Der Name wird gefolgt von der Typangabe und diese wird gefolgt von der Initialisierung der Variablen. Die Typangabe wird meist weggelassen, da sie sich fast immer aus der Initialisierung ergibt. In den Kommentaren sind die Java-Gegenstücke angegeben.

```
val a = 1.5               // final double a = 1.5;
val b = "abc"             // final String b = "abc";
var c = 0                 // int c = 0
val d = new Array[Int](5) // int[] d = new int[5]
```

In der letzten Zeile sehen Sie, dass Arrays in Scala wie eine parametrisierte Klasse verwendet werden. Typparameter werden in Scala in eckige Klammern eingeschlossen.

[5] Das Gegenstück zu Java-Klassenfunktionen sind die Methoden eines Singleton-Objekts.

Anstelle der eckigen Klammer werden für Indizierung und Größenangabe runde Klammern verwendet. Anstelle der Array-Literale gibt es eine entsprechenden Fabrikmethode (bei der unnötige Typangaben wieder weggelassen werden können):

```
val a: Array[Int] =
    Array[Int](1, 2, 3)     // vollstaendige Form
val a = Array(1,2,3)        // int[] a = {1, 2, 3};
```

Bei der Deklaration von Funktionsparametern steht nie ein `val`. Funktionsparameter sind in Scala *immer* unveränderlich.

Klassenparameter können, aber müssen nicht, mit `val` oder `var` deklariert sein (siehe unten).

Schließlich kann man festlegen, dass ein Wert nicht zu dem Zeitpunkt seiner Definition, sondern erst später bei der ersten Verwendung ausgewertet wird. Die Definition erfolgt durch die Schreibweise `lazy val`. Dies ist in der funktionalen Programmierung auch deshalb wichtig, weil es die Definition unendlicher Datenstrukturen erlaubt. Dies werden wir später besprechen.

Hier soll aber nur ein einfaches (prozedurales) Beispiel stehen:

```
var x = 4
lazy val y = 10 * x
x = 5
println(y) // Ausgabe: 50
```

4.2.3 Scala-Singleton-Objekte

In Scala gibt es keine statischen Funktionen (diese sind ja nicht objektorientiert).[6] Für global anzusprechende Funktionen gibt es singuläre Objekte, die mit ihrem global sichtbaren Namen angesprochen werden. Ein wichtiger Unterschied zwischen global bekannten Objekten und Klassen besteht auch darin, dass für das Verhalten dieser Objekte alle Regeln der Objektorientierung weiter gelten. Referenzen zu Scala-Objekten können in Variablen gespeichert werden.

Als Beispiel soll hier ein kleines Hello-World Programm stehen:

```
package beispiel

object HelloWorld {
    def main(args: Array[String]) {
        printHello()
    }

    def printHello() {
        println("hello world")
    }
}
```

[6] Die statischen Funktionen und Variablen der Java-Bibliothek lassen sich aber immer noch über den Klassennamen ansprechen.

Im Vorbeigehen sehen wir hier schon einmal zwei Funktionsdefinitionen, mehr darüber unten. Die Funktion `main` übernimmt die Funktion der Main-Funktion von Java, nämlich eine Anwendung zu starten. Mit `Array[String]` wird das Array der Kommandozeilenparameter deklariert.

Beim Aufruf eines fremden Objekts muss auch in Scala eine Objektreferenz stehen. Bei Singleton-Objekten ist dies der Objektname:

```
package beispiel

object HelloWord {
    def main(args: Array[String]) {
        Printer.printHello()
    }
}

object Printer {
    def printHello() {
        println("hello world")
    }
}
```

Bitte beachten Sie, dass `Printer.printHello()` mit `Printer` ein Objekt und nicht eine Klasse meint.

4.2.4 Scala-Klassen und Konstruktoren

Eine Scala-Klasse entspricht einer Java-Klasse. Aber auch hier sind ein paar Dinge vereinfacht. Zunächst einmal hat jede Klasse einen sogenannten *primären Konstruktor*. Dessen Parameter stehen unmittelbar im Kopf der Klasse, sein Körper steht als Anweisungsfolge im Klassenkörper.

Die folgende Java-Klasse definiert Personen als unveränderliche Objekte:

```
public class Person {
    private final String name;
    private final int alter;

    public Person(String n, int a) {
        if (a < 0) throw new IllegalArgumentException();
        name = n;
        alter = a;
    }

    public String name() {
        return name;
    }

    public int alter() {
        return alter;
    }
```

Die gleiche Klasse sieht in der Scala-Definition viel einfacher aus. Der Konstruktor erscheint als Klassenkörper. Den Instanzvariablen werden, wenn sie nicht `private` sind, automatisch Getter- und Setter-Methoden zugeordnet, sodass sich deren explizite Definition fast immer erübrigt.

```
class Person(n: String, a: Int) {
    require(a >= 0)
    val name = n
    val alter = a
}
```

Die Klasse lässt sich noch kürzer schreiben, wenn den Klassenparametern ein `val` oder ein `var` vorangestellt wird.

```
class Person(val name: String, val alter: Int) {
    require(alter >= 0)
}
```

Es sei hier kurz erwähnt, dass häufig einfache Klassen, die im Wesentlichen nur Information transportieren, als sogenannte Case-Klassen definiert sind. Bei diesen werden einige Methoden, wie `toString` und `equals` automatisch definiert. Case-Klassen sind sehr praktisch im Zusammenhang mit dem Pattermatching der Match-Case und der Receive-Case Ausdrücke (daher stammt auch der Name). Eine kleine Befehlsfolge verdeutlicht ihre Verwendung. Weitere Beispiele kommen später.

```
case class Person(name: String, alter: Int) {
    require(alter >= 0)
}

val pers = Array(
    Person("Karin", 17),
    Person("Hans", 9),
    Person("Karin",17))

println(pers(0))                // ergibt: Person(Karin, 17)
println(pers(0) == pers(2))     // ergibt true

val gefunden =
    pers.exists(_.name == "Karin")  // ergibt true
```

Vererbung wird in Scala, wie in Java, durch das Schlüsselwort `extends` ausgedrückt. Darüber hinaus gibt es eine Form der Mehrfachvererbung.

Überschriebene Elemente (Variable, Methoden) müssen durch dass Schlüsselwort `override` kenntlich gemacht werden.

4.2.5 Methodendeklaration

Die Methodendeklaration wird durch `def` eingeleitet. Darauf folgt der Name der Methode und dann die optionale, in Klammern eingeschlossene Parameterliste. Anschließend folgt der Rückgabetyp gefolgt von dem Methodenkörper. Anstelle des Schlüsselworts `void` fungiert in Scala der Typ `Unit`.

Hier ein paar Methodendeklarationen mit vollständiger Angabe aller Informationen:

```
def intMethode(x: Int): Int = 3 * x

def fakultaet(n: Int): Int =
    if (n == 0) 1 else n * fakultaet(n - 1)

def voidMethode(x: String): Unit =
    println(x)

def langeIntMethode(n: Int): Int = {
    var s = 0
    for (i <- 1 to n) s += i
    s
}

def langeVoidMethode(): Unit = {
    print("hello ")
    println("world")
}
```

Was fällt auf ?

- In Deklarationen steht der Typ immer hinter dem Variablennamen oder hinter der Parameterliste. Als Trennzeichen steht ein Doppelpunkt.
- Auf den Funktionskopf und Rückgabetyp folg ein Gleichheitszeichen gefolgt von dem Methodenkörper.
- Es gibt kein `return`.[7] Der Rückgabewert einer Methode ist der Wert des zuletzt stehenden Ausdrucks.
- Geschweifte Klammern sind nur nötig, wenn die Methode aus mehreren Anweisungen besteht.

Wie schon angedeutet, kann man das auch etwas kürzer schreiben. Der Rückgabetyp muss nur bei rekursiven Methoden zwingend angegeben werden.[8] Allerdings gilt die Regel, dass auch öffentliche Methoden einen vollständig getypten Funktionskopf haben sollten.

Fehlt im Methodenkopf das Gleichheitszeichen (die geschweiften Klammern sind dann aber zwingend notwendig) gilt automatisch die Angabe `: Unit`. Abgekürzt lauten die obigen Definitionen:

```
def intMethode(x: Int) = 3 * x

def fakultaet(n: Int): Int =
    if (n == 0) 1 else n + fakultaet(n - 1)

def voidMethode(x: String) {
    println(x)
}

def langeIntMethode(n: Int) = {
```

[7] Doch, gibt es schon. Das `return` hat in Scala aber ein andere Bedeutung als in Java. Sie sollten es möglichst nicht verwenden!

[8] Auch bei der (seltenen) Verwendung der Return-Anweisung muss der Rückgabetyp angegeben werden.

```
    var s = 0
    for (i <- 1 to n) s += i
    s
}

def langeVoidMethode() {
    print("hello ")
    println("world")
}
```

4.2.6 Kontrollstrukturen

Scala kennt die wichtigsten von Java her bekannten Kontrollstrukturen. Zum Teil ist ihre Form leicht verändert.

If-Ausdruck hat die gleiche Syntax wie das Java-if. Allerdings hat er die Semantik des bedingten Ausdrucks.

While-Anweisung entspricht exakt dem Java Gegenstück.

Do-While-Anweisung ist vorhanden, wird aber kaum verwendet.

Switch-Verzweigung ist (zum Glück) gestrichen.

Match-Ausdruck erlaubt die Muster-gesteuerte Auswahl von Ausdrücken. Die Verwendungsbeispiele von Java's Switch-Anweisung sind ein ganz einfacher Spezialfall.

Java For-Anweisung Die elementare For-Schleife ist in Scala nicht vorhanden.

For-Anweisung ist eine etwas mächtigere Variante der von Java bekannten Foreach-Schleife.

For-Ausdruck ist sehr nützlich im Umgang mit Datenstrukturen. In Java gibt es kein Gegenstück.

Der komplexere Match-Case-Ausdruck wird im nächsten Abschnitt vorgestellt. Hier sollen die anderen Kontrollstrukturen an einem kleinen Beispiel illustriert werden.

Es soll aber nicht verschwiegen werden, dass es sich dabei um prozedurale Lösungen handelt.

```
def summe(a: Array[Int]) = {
    var s = 0            // int s = 0;
    for (x <- a) s += x  // for (int x : a)  s += x;
    s                    // return s;
}

def quadriereElemente(a: Array[Int]) {
    for (i <- 0 until a.length) //  i = 0 .. a.length - 1
        a(i) *= a(i)
}

def maximum(a: Array[Int]) = {
    var m = a(0)
```

```
    for (x <- a) {
        m = m max x
    }
    m
}

def fakultaetIterativ(n: Int) = {
    var f = 1
    for (i <- 1 to n) f *= i
    f
}
```

Die folgenden Diskussionen vorwegnehmend, sollen hier schon mal funktionale Lösungen stehen:

```
def summe(a: Array[Int]) = a.sum
def quadriere(a: Array[Int]) = a.map(x => x * x)

def hoch3(a: Seq[Int] =
    for (x <- a) yield x * x * x

def maximum(a: Array[Int]) = a.reduce(_ max _)

def fakultaet(n: Int) = {
    @tailrec
    def f(i: Int, a: Int): Int =
        if (i > n) a else f(i + 1, i * a)
    f(1, 1)
}
```

Bei den ersten Funktionen werden Sie sicher (bei allen Unklarheiten) zustimmen, dass sie einfacher sind als die ursprüngliche Form. Bei der Fakultätsfunktion ist die endrekursive Lösung angegeben. Der Scala-Compiler übersetzt endrekursive Funktionen in effizienten Code. Wir werden später noch darauf eingehen.

4.3 Wichtige Erweiterungen

4.3.1 Funktionsobjekte

Als funktionale Sprache unterstützt Scala auch Funktionsobjekte. In der einfachsten Form speichert man eine Methode in einer Variablen.

```
object A {
    def m():Int {
        3
    }
}

object X {
    def main(args: Array[String]) {
        val hello_fkt1 = A.m _
        val hello_fkt2: ()=>Int = A.m
        val drei = A.m
    }
}
```

Das Beispiel zeigt ein Problem auf. Wegen der Regel, dass in Scala die Klammer beim Funktionsaufruf fehlen kann, ist es nicht klar, was der Ausdruck `A.m` bedeutet. Scala hat hier die Regel, dass zunächst ein „normaler" Funktionsaufruf gemeint ist. Will man dagegen eine Referenz auf ein Funktionsobjekt weitergeben, muss entweder aus dem Kontext zu erkennen sein, dass dies gemeint ist (z.B. durch eine Typangabe) oder man muss durch einen Unterstrich _ deutlich machen, dass nur die Referenz kopiert werden soll.

Es versteht sich von selbst, dass der Inhalt einer Funktionsvariablen an andere Variablen und auch an Methoden weitergereicht werden kann. Bei Funktions-Variablen gelten andere Regel für die Unterscheidung von Aufruf und Referenz: Nur wenn der Name von einer (evtl. leeren) Parameterliste gefolgt ist, ist ein Funktionsaufruf gemeint, sonst die bloße Referenz.

Hier wurde eine Methode in ein Funktionsobjekt umgewandelt. Man kann Funktionsobjekte aber auch durch sogenannte Funktionsliterale definieren.

Funktionsliterale (auch Methdoden allgemein) können überall, d.h. auch in Methoden stehen. Zu der Angabe eines Funktionsliterals gehört die Angabe der Signatur (die manchmal wieder „geraten" wird) und die Angabe des Funktionskörpers.

```
val addiereXundY = (x: Int, y: Int) => x + y
```

Die rechte Seite der Zuweisung stellt ein Funktionsliteral dar. Durch den angegebenen Ausdruck wird ein Funktionsobjekt definiert. Da diese Funktion keinen Namen trägt, nennt man sie auch *anonyme Funktion*. Ein anderer Name für anonyme Funktion ist *Lambda-Ausdruck*.[9]

anonyme Funktion ::= (*Parameterliste*) =>*Ausdruck*
| *Variable* =>*Ausdruck*
| *Ausdruck mit anonymen Variablen*

In dem Beispiel mussten der Typ von x und y angegeben werden. Wenn dieser aus dem Kontext hervorgeht, kann er weggelassen werden. Manchmal sind nicht einmal die Namen der Parameter notwendig. Dann verwendet man die anonyme Variable _. Es versteht sich von selbst, dass eine anonyme Funktion selbst keinen Namen hat und auch nicht zwingend in einer Variablen gespeichert wird.

```
val a = Array(1,2,3,4,5)
val summe = a.reduce((x:Int, y:Int) => x + y)
val summeKuerzer = a.reduce((x,y) => x + y)
val summeNochKuerzer = a.reduce(_ + _)
val summeGanzKurz = a.sum // sum ist halt vordefiniert
```

In diesem Beispiel wird die Summe aller Zahlen eines Arrays mittels der Funktion `reduce` berechnet. Diese Funktion kann eine Liste oder ein Array auf einen Wert

[9] Dieser Begriff geht auf A. Church zurück, der die Lambda-Notation zu Definition von Funktionen eingeführt hat. In der Folge wird der Begriff in vielen funktionalen Programmiersprachen verwendet.

reduzieren, indem von links nach rechts die Elemente mittels der angegebenen Funktion verknüpft werden.

Funktionsobjekte kennen die Variablenumgebung, in der sie definiert wurden. Sie nehmen diese Umgebung mit und werten die Werte äußerer Variablen bei ihrer Anwendung aus. Dieser Sachverhalt wird später wiederholt benutzt.

Definition:

*Eine Funktion, die ausschließlich lokale Variable und Parameter enthält, heißt **geschlossen**. Variable, die in der die Funktionsdefinition umfassenden Umgebung definiert sind, heißen **freie Variable**. Eine Funktion mit freien Variablen heißt auch **offen**. Die Vervollständigung der Funktion mit dem Bezug auf die freien Variablen wird als **closure** bezeichnet. Da in funktionalen Sprachen alle Funktionsobjekte mit freien Variablen die Closure-Eigenschaft haben, verwendet man den Begriff closure auch als Synonym für Funktionsobjekt oder Lambda-Ausdruck.*

In Scala bedeutet dies, dass Funktionsobjekte mit freien Variablen immer mit dem Objekt, dem sie entstammen, verbunden bleiben. Darüber hinaus gehören lokale Funktionen, die innerhalb einer anderen Funktion definiert wurden, auf Dauer zu dem lokalen Kontext dieser Funktion.[10]

4.3.2 Die Match-Case Anweisung von Scala

Scala kennt nicht die altmodische Switch-Case-Anweisung. Dagegen enthält es, ganz in der Tradition funktionaler Programmiersprachen, ein umfassenderes und mächtigeres Konstrukt für die Mehrfachauswahl. Die Syntax ist wie folgt

Match-Case ::= *Objekt* `match` {*Case-Fall** }
Case-Fall ::= `case` *Muster Guard*? =>*Aktionen*
Guard ::= `if` *Bedingung*

Die Case-Fälle können im einfachsten Fall einfache Werte darstellen, sie können aber auch durch komplexe Ausdrücke mit unbekannten Platzhaltern oder sogar durch reguläre Ausdrücke beschrieben sein. Hier sollen nur die einfacheren Fälle durch Beispiele dargestellt werden.

Zunächst soll eine switch-Anweisung aus Java nach Scala überführt werden.

```
String zifferZuName(int n) {
    switch(n) {
       case 0: return "Null";
       case 1: return "Eins";
       ...
       default: return "****";
   }
}
```

[10] Es mag ironisch klingen: Wenn man in Java Funktionsobjekte durch innere Klassen nachbildet, kann man freie Variablen einer lokalen Umgebung nicht verändern. In der funktionalen Sprache Scala (funktionale Sprachen wollen eigentlich keine Veränderung) ist das möglich.

Die äquivalente Scala-Form sieht fast gleich aus:

```
def zifferZuName(n: Int) = n match {
   case 0 => "Null"
   case 1 => "Eins"
   ...
   case _ => "****"
}
```

Scala hat hier ein paar Vorteile. Es gibt kein fall-through und jeder Fall stellt einen eigenen Block (mit eigenen Variablen) dar. Wie Sie sehen, wird Case als Ausdruck mit einem Ergebnis aufgefasst wie das auch bei dem If-Ausdruck geschehen ist.

Sie wissen, dass Case in Java keine Bedingung enthalten darf. Das ist in Scala anders. Hinter jedem Case darf optional eine Bedingung stehen. In dem folgenden Beispiel ist auch demonstriert, dass in dem Case-Muster Variablen vorkommen dürfen.

```
def signum(n: Int) = n match {
   case 0 => 0
   case x:Int if x > 0 => 1
   case _ => -1
}
```

In dem Beispiel ist Verschiedenes zu erkennen. So kann das Muster eine Typangabe enthalten (diese ist hier nicht notwendig). Die Reihenfolge der Fälle spielt eine Rolle. Der letzte der drei Fälle trifft nur auf negative Zahlen zu. Der Unterstrich spielt in den Musterausdrücken die Rolle einer beliebigen anonymen Variablen.

Die Typangabe kann für die Fallunterscheidung relevant sein. Scala verfährt dann so, dass eventuell nötige Typanpassungen automatisch vorgenommen werden.

Sie erinnern sich an `equals` aus der Java-Klasse Bruch?

```
public boolean equals(Object that) {
   if (! (that instanceof Bruch)) return false;
   Bruch b = (Bruch) that;
   return this.zaehler == b.zaehler &&
          this.nenner == b.nenner;
}
```

In Scala sieht das so aus:

```
override def equals(that: Any) = that match {
   case b: Bruch =>
      this.nenner == b.nenner && this.zaehler == b.zaehler
   case _ => false
}
```

In Scala hat die geklammerte Folge der Case-Fälle eine eigenständige Bedeutung. Sie kann auch ohne `match` in ganz anderem Kontext auftreten. Es handelt sich genau genommen um die Definition einer *partiell definierten Funktion*. In Scala ist die Case-Folge daher eine Instanz von `PartialFunction`.

Kapitel 5

Funktionale Programmierung

5.1 Was zeichnet den funktionalen Programmierstil aus?

Funktional definierte Programme sind erheblich kürzer als die äquivalenten prozeduralen Gegenstücke. Schauen Sie sich einmal dieses kleine Programmbeispiel an, das einer Praktikumsaufgabe von AP1 nachempfunden ist.

Bei der Aufgabe geht es darum, aus einer Datei einige Zahlen einzulesen, diese in einem 2-dimensionalen Feld [1] zu speichern und herauszufinden, welche Zahlen davon „Schnapszahlen“ sind (d.h. durch 11 teilbar sind) und diese Zahlen ebenso wie die Anzahl der Schnapszahlen auszugeben. Dabei sollen mehrfach vorkommende Zahlen aber nur einmal gezählt werden.

Ein funktionales Scala-Programm sieht so aus:

```
import java.util.Scanner
import java.io.FileReader
import collection.SortedSet

object Schnapszahlen {
  def main(args: Array[String]) {
    val in = new Scanner(new FileReader("zahlen"))
    val a = Array.tabulate(16, 16)((i,j) => in.nextInt)
    println("Programm zur Ueberpruefung auf Schnapszahlen")
    println("\nWerte der Testmatrix:")
    for(zeile <- a) println(zeile.mkString(" "))
    val zz = a.flatten.filter(z => z%11==0).to[SortedSet]
    printf("%nSchnapszahlen: %s%n", zz.mkString(" "))
    printf("Es sind %d Schnapszahlen.%n", zz.size)
  }
}
```

Versuchen Sie das Problem in Java zu lösen! Das Java-Programm wird sicher deutlich länger sein. Funktionale Programmierung ermöglicht nämlich ein besonders hohes Maß an Modularisierung. Nehmen wir die folgende Zeile:

```
val quad = Array.tabulate(16, 16)((i,j) => in.nextInt)
```

[1] Zur Lösung der Aufgabe braucht man kein 2-dimensionales Feld. Aber das war im Praktikum gefordert.

Die Funktion `Array.tabulate` erstellt eine 16 x 16 Matrix. Das Besondere ist hier das Funktionsliteral

```
(i,j) => in.nextInt
```

Gefordert ist eine Funktion, die für das Element i, j bei der Initialisierung aufgerufen wird. In diesem Fall wird für jedes Element einfach ein neuer Wert aus der Datei gelesen.

Die nächste interessante Zeile ist

```
val zz = a.flatten.filter(z => z%11==0).to[SortedSet]
```

Hier wird das 2-dimensionale in ein 1-dimensionales Feld verwandelt (`flatten`), aus diesem werden die durch 11 teilbaren Elemente in ein neues Feld übertragen (`filter`) und schließlich werden doppelte Elemente entfernt (`to[SortedSet]`) und die Zahlen werden gleichzeitig sortiert.

Es gibt hier einen Unterschied zur „normalen" funktionalen Schreibweise. Scala implementiert Datenstrukturen, wie Arrays und Listen, objektorientiert. In einer rein funktionalen Sprache würde die Zeile vielleicht so aussehen:

```
val zz = SortedSet(filter(z=>z%11==0, flatten(quad))))
```

Funktionen stehen ja immer vor der jeweiligen Parameterliste. In der Objektorientierung folgt dagegen der Methodenaufruf auf die Objektreferenz. In der Vorlesung werden wir beide Schreibweisen verwenden. Die funktionale Schreibweise verwenden wir bei selbst geschriebenen Funktionen. Den objektorientierten Stil verwenden wir dagegen vor allem bei den vordefinierten Bibliotheksklassen.[2]

Unabhängig von dem Reihenfolgeproblem sehen Sie aber auch wieder bei dieser Zeilen mehrere Gründe warum der Algorithmus in Java nicht so einfach zu implementieren ist:

- Funktionale Datenstrukturen sind unveränderlich. Alle Operationen werden durch Funktionen implementiert, die ein Result zurückgeben. Dies erlaubt die direkte Verknüpfung mehrerer Funktionsaufrufe. Der prozeduralen Programmierstils hat dagegen oft Seiteneffekte und dann keine expliziten Rückgabewerte.
- Scala erlaubt die bequeme Definition von *Funktionsliteralen*. Funktionsliterale ermöglichen die Verwendung von Funktionen, die allgemeine Aktionen auf Datenstrukturen ausführen. In dem Beispiel sucht `filter` alle die Zahlen zusammen, die die angegebene logische Bedingung erfüllen.
- Diese beiden Gründe (Funktionsliterale und unveränderliche Datenstrukturen) ermöglichen die modulare Definition von Bibliotheksfunktionen. Daher verfügen funktionale Sprachen oft über eine perfekt ausgebaute Bibliothek für Datenstrukturen. `mkString` ist dafür ein Beispiel. Diese Funktion

[2] Fallen Ihnen die vielen Funktionsklammern auf? Das ist ein Problem der funktionalen Schreibweise. Das fällt besonders in der Sprache Lisp auf. In modernen Sprachen, wie Haskell oder Scala, begegnet dem auch dadurch, dass man unnötige Klammern weglassen kann.

fasst die Elemente eines Arrays (oder einer beliebigen anderen sequentiellen Datenstruktur) zu einem String zusammen. Der übergebene String dient als Trennzeichen.

Der Begriff *Funktionsliteral* wurde zwar bereits im letzten Kapitel definiert. Wegen der zentralen Bedeutung der Begriffe schadet es aber nicht, den Begriff nochmals zu erläutern.

Definition:
*Ein **Funktionsliteral** beschreibt eine Funktion. Funktionen können in Variablen gespeichert, an Funktionen übergeben und von Funktionen zurückgegeben werden. Die in der Funktion angesprochenen freien Variablen (in der Umgebung der Funktion definierten Variablen) bleiben auch bei der Weitergabe der Funktion an diese gebunden (*lexical closure*). Funktionsliterale werden auch **Lambda-Ausdruck**, **anonyme Funktion**, **Closure** oder **Funktionsobjekt** genannt.*

Die Vielfalt der Namen ist vielleicht typisch für den ungewohnten Umgang mit Funktionsliteralen. In C definieren Sie Funktionen in der bekannten syntaktischen Form. Eine Funktion hat immer einen Namen. Funktionen haben zwar auch eine Speicheradresse, die man weitergeben kann, mit der Funktion als solcher wird aber konzeptionell nicht operiert. In der funktionalen Programmierung ist das genau anders.

Natürlich kann man an dem kleinen Beispiel nicht alles sehen. Aber es gibt noch weitere Punkte die bei der Frage der Verwendung des funktionalen Stils eine Rolle spielen:

- Funktionale Sprachen sind theoretisch besser fundiert und von der Ausdrucksmöglichkeit her vollständiger als prozedurale Abläufe. Sie ermöglichen ein sehr hohes Maß an Abstraktion. Abstraktion ermöglich erst die hochgradig wiederverwendbaren Bibliotheken funktionaler Sprachen.
- Der hohe Abstraktionsgrad funktionaler Formulierungen ermöglicht automatische Optimierungen, insbesondere auch die einfache Ausnutzung von paralleler Hardware.
- Der modulare Ansatz funktionaler Sprachen hat immer wieder neue Paradigmen und Mechanismen in die Sprachen des Mainstreams eingeführt (Objektorientierung. Datenstrukturen, garbage collection).
- Der hohe Abstraktionsgrad hat häufig seinen Preis in höheren Speicher- und Laufzeitanforderungen.

5.2 Das Paradigma der funktionalen Programmierung

Der Funktionsbegriff ist viel älter als das Nachdenken über Programmierung. Man hat seit Beginn des 20. Jahrhunderts darüber nachgedacht, wie sich Berechnungen mathematisch streng beschreiben lassen. Hierfür stehen Namen wie *Gödel*, *Turing* und *Church*. Church hat durch den von ihm entwickelten *Lambda-Kalkül* die Brücke zur funktionalen Programmierung geschlagen. Nach der Entwicklung des elektronischen Computers hat dann die Arbeitsgruppe von Marvin Minsky

in den 1950er Jahren, ausgehend von dem Lambda-Kalkül, die erste nach funktionalen Grundsätzen gestaltete Programmiersprache, nämlich *LISP*, entwickelt. Auch wenn die Syntax dieser Sprache vielleicht etwas ungewohnt aussieht, so muss man doch feststellen, dass es sich dabei um die erste (immer noch) moderne Programmiersprache handelt.

LISP wird nach wie vor in verschiedenen Varianten als aktuelle Programmiersprache genutzt. Das soll uns aber hier nicht interessieren. Immerhin sind fast alle Sprachmerkmale der LISP-Welt inzwischen auch in anderen funktionalen Programmiersprachen (und teilweise auch in Scala) verfügbar. Vor der Besprechung der konkreten Realisierung in Scala kommen wir aber nicht daran vorbei, zunächst auf die Begriffsbildung durch die Mathematik einzugehen.

5.2.1 Funktionen in der Mathematik

Der Funktionsbegriff

Die mathematische Begriffsbildung ist im Vergleich zur Informatik sehr alt. Ihre Grundbegriffe und Methoden gelten als weitgehend etabliert. Die Ausdrucksfähigkeit der Mathematik ist schier unendlich. Warum soll man also die Mathematik nicht als Vorbild für die Programmierung nehmen?

Zunächst die Definition:

Definition:

> *Eine **Funktion** ordnet den Elementen eines **Definitionsbereichs** (englisch:* domain*) jeweils ein Element eines **Wertebereichs** (englisch:* codomain*) zu. Eine **partielle Funktion** ist eine Zuordnung, die nur für Teile des Definitionsbereichs definiert ist.*

Diese Definition ist für sich alleine noch nicht sehr hilfreich. Wichtiger ist, wie man Funktionen definieren, und wie man damit umgehen kann. Auch hier hilft uns die Mathematik weiter.

Die einfachste Möglichkeit, eine Funktion zu definieren, besteht darin, fürendrekursiven alle möglichen Ausgangswerte die Funktionsresultate aufzulisten. Wegen der Vielzahl der Zuordnungen ist dieses Vorgehen aber in der Regel nicht praktikabel.

In Normalfall ist es besser, eine Funktion durch andere Funktionen zu erklären. Gegebenenfalls müssen dabei verschiedene Bereiche des Definitionsbereichs durch unterschiedliche partielle Funktionen definiert werden. Wenn die zu definierende Funktion in der Definition durch sich selbst erklärt wird, spricht man von einer *rekursiven Funktionsdefinition*.

Die Definition von Funktionen durch einfachere Funktionen setzt das Vorhandensein elementarer vordefinierter Operationen (z.B. Grundrechenarten) voraus.

Un eine zu tiefgehende mathematische Darstellung zu vermeiden, soll hier das Gesagte an dem einfachen Beispiel der Definiton der Fakultät erläutert werden.

$$\text{fac: } \mathbb{N} \longrightarrow \mathbb{N} \tag{5.1}$$

$$\text{fac: } n \longrightarrow \text{fac}(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot \text{fac}(n-1) & \text{falls } n > 0 \end{cases} \tag{5.2}$$

Die erste Zeile gibt die Signatur, d.h. den Definitions- und den Wertebereich der Funktion an. Auch in der Informatik heißt die Typfestlegung ener Funktion oder Methode so. Es folgt dann die Definition der Funktionsgleichung. Sie baut auf den elementaren Funktionen der Multiplikation und Subtraktion und auf der rekursiven Anwendung der Fakultätsfunktion selbst auf. Hier ist eine Fallunterscheidung nötig, da für die $n = 0$ eine besondere Festlegung getroffen werden muss.

Programmiersprachen legen wenig Wert auf die genaue Festlegung von Definitions- und Wertebereich. Bei dynamisch getypten Sprachen fehlt sogar überhaupt jede Typdeklaration im Programm. Bei anderen Sprachen, wie Java oder Scala, ist eine ungefähre Typangabe möglich. Scala und Java kennen aber nicht die Möglichkeit, den erlaubten Zahlenbereich durch eine Typangabe einzugrenzen. Zum Beispiel kann man in Java negative Argumente nicht durch eine Typangabe verbieten. Solche Vorbedingungen können erst zur Laufzeit geprüft werden. Fehler werden dann durch das Werfen einer Ausnahme „geahndet".

Gebundene und freie Variable

Betrachten wir die folgende formale Funktionsdefinition:

$$f(x) = ax^2 + bx + c$$

In dieser Formel ist x auf der linken Seite der Gleichung als Funktionsparameter kenntlich gemacht. Es ist klar, dass auf der rechten Seite der jeweilige Wert von x gemeint ist. Diese Variable ist an den Wert des jeweiligen Funktionsarguments *gebunden*. Was aber sind a, b und c? Von der Mathematik her wird man sagen, dass dies drei Konstanten sind. Erst bei konkret bekannten Werten ist die Funktion wirklich definiert.

In der Sprache der Mathematik spricht man hier auch von *freien Variablen*, die noch nicht an Werte gebunden sind. Zum Zwecke der Evaluierung der Funktionswerte muss dann aber eine vollständige Bindung vorliegen.

Definition:

Eine Variable, die innerhalb einer Formel definiert ist, heißt ***gebundene Variable****. Variable, die der äußeren Umgebung entnommen sind, heißen* ***freie Variable****.*

Operationen mit Funktionen

Die Höhere Mathematik zeichnet sich dadurch aus, dass in ihr nicht nur Funktionen definiert und berechnet werden, sondern dass auch die Eigenschaften von Funktionen und von Operationen auf Funktionen untersucht werden.

Ein naheliegendes Beispiel ist der Differentialoperator. Dieser ordnet einer (z.B. reellen) Funktion eine andere Funktion zu. Der reellen Funktion $\sin(x)$ ist so die reelle Funktion $\cos(x)$ zugeordnet, der Funktion $\log(x)$ die Funktion $1/x$. Jeder differenzierbaren reellen Funktion ist eine andere reelle Funktion zugeordnet.

Daneben gibt es aber beliebig viele weitere Funktionen von Funktionen. Dies geht soweit, dass sogar die Definition einer Funktion (dies ist ja in erster Linie eine Komposition von Funktionen) selbst als Funktion aufgefasst werden kann.

Man nennt Funktionen, die Funktionen als Argumente oder Ergebnisse haben, „Funktionen höherer Ordnung“.

Operationen auf Datenstrukturen

In der Mathematik kennt man Operationen auf Datenstrukturen, die in dieser Eleganz in imperativen Programmiersprachen nicht vorhanden sind. Wenn ich z.B. in Java einer Menge von Zahlen die Menge ihrer Quadrate zuordnen will, muss ich dies mit einer for-Schleife machen wie z.B. in dem folgenden Programm:

```
Set<Double> quadriere(Set<Double> menge) {
    Set<Double> ergebnis = new HashSet<Double>();
    Iterator<Double> iter = menge.iterator();
    while (iter.hasNext()) {
        double zahl = iter.next();
        ergebnis.add(zahl * zahl);
    return ergebnis;
}
```

In diesem Programm *sieht* man förmlich den Ablauf.[3] Man kann genau erkennen, was der Computer tut. Anders die Mathematik;

$$Q(M) = \{y \mid y = x * x \text{ für } x \in M\} \tag{5.3}$$

In der funktionalen Programmierung werden uns „Formeln“ begegnen, die der mathematischen Form verwandt sind, z.B. wie:

```
def mengeDerQuadrate(menge: Set[Double]): Set[Double] =
  menge.map(x => x * x)
```

oder in der Formulierung mittels „for comprehension“: ermöglicht weitere Optimierungen, weil

```
 def mengeDerQuadrate(menge: Set[Double]): Set[Double] =
  for (x <- menge) yield x * x
```

Als zweites Beispiel nehmen wir die Fakultätsfunktion. Man kann sie definieren als „das Produkt der Zahlen von 1 bis n“.

Die prozedurale Formulierung in Java oder C lautet:

[3] Das sieht nicht ganz so schlimm aus, wenn man die Foreach-Schleife verwendet,

```
int fakulaet(int n) {
  int f = 1;
  for (int i = 1; i <= n; i++)
    f *= i;
  return f;
}
```

Die formale mathematische Definition:

$$n! = \Pi_{\nu=1}^{n} \nu \tag{5.4}$$

lässt sich exakt in Scala nachbilden:

```
def fakultaet(n: Int) = (1 to n) product
```

Mathematische Objekte

Der naheliegenste, und damit am leichtesten zu übersehende, Unterschied von Mathematik und prozeduraler Programmierung liegt in dem Begriff der Variablen. Die Tatsache, dass der Inhalt einer prozeduralen Variablen zu verschiedenen Zeiten verschieden ist, bedeutet, dass ich nie genau sagen kann, was eine Programmanweisung bedeutet und bewirkt. Es ist der tiefere Grund, dass prozedurale Programme schwer zu verstehen und testen sind.

Mathematische Aussagen sind dagegen allgemeingültig. Daraus ergibt sich die Beweisbarkeit. Dies gilt aber wirklich nur unter der Voraussetzung, dass Werte nicht verändert werden. Eine Funktion verändert nicht einen Wert, sondern sie ordnet einem oder mehreren Werten einen neuen Werte zu.

Weitere Operationen

Es ist an dieser Stelle nicht der Platz, alle Operationen mit Funktionen anzuführen. In der Mathematik gibt es große Teilgebiete, die dies tun. Für die Programmierung sind andere Aspekte von Interesse.

5.2.2 Grundelemente der funktionalen Programmierung

Merkmale

Die besonderen Merkmale der funktionalen Programmierung ergeben sich aus dem mathematischen Vorbild und sind hier im Unterschied zum prozedural-imperativen Modell dargestell:

- Funktionale Programmierung ist *deklarativ*. Die Bedeutung ergibt sich aus dem statischen Zusammenhang, ohne die Simulation eines Ablaufs. Der genaue Ablauf ist irrelevant.
- Funktionen haben *keine Seiteneffekte*.
- Funktionale Programme kennen *keinen veränderlichen Zustand*.

- Funktionen sind *erstklassige Objekte*. Dies bedeutet, dass man sie genauso wie andere Werte an Variablen binden oder an andere Funktionen übergeben kann.
- Im funktionalen Sinne bedeutet die *Fallunterscheidung* die Auswahl der für ein Element des Definitionsbereichs zuständigen *partiellen Funktion*.
- An die Stelle der Iteration mittels `while`, die auf veränderlichen Variablen beruht, treten die *Rekursion* und Funktionen höherer Ordnung.
- Funktionale Programmierung basiert auf *unveränderlichen Datenstrukturen*. Operationen auf diesen Datenstrukturen erzeugen – wenn nötig – neue Datenstrukturen.
- Die funktionale Programmierung legt einen anderen Programmierstil nahe. Dieser tendiert dazu, für Operationen auf Datenstrukturen *Funktionen höherer Ordnung* anzubieten. Eine Funktion höhere Ordnung auf Datenstrukturen ordnet einer Datenstuktur und einer Funktion einen Ergebniswert zu. Das Ergebnis kann selbst wieder eine Datenstruktur oder eine Funktion sein.
- Funktionen stellen eine Zuordnungsvorschrift dar. Sie haben nicht zwingend einen Namen (*anonyme Funktion*).
- Funktionen können neben den gebundenen auch freie Variablen enthalten (*closure*).

Vorteile

Grundsätzlich haben funktionale Programme das Problem, dass sie fast immer in einer Umgebung ablaufen die, gelinde gesagt, unfreundlich ist. Der von Neumann'sche Universalrechner ebenso wie die JVM sind für andere Programmierparadigmata optimiert. Trotzdem ergeben sich selbst in diesen Umgebungen einige Vorteile:

- Dadurch dass es keine veränderlichen Werte gibt, ermöglicht die funktionale Programmierung zusätzliche Optimierungen. Ein Beispiel sind die unveränderlichen String-Objekte in Java. Da sie unveränderlich sind, kann der Compiler die Anzahl der Objekte verringern (gleichlautende Strings werden durch ein einziges Objekt repräsentiert).
- Einige Optimierungen sind erst dadurch möglich, dass in der funktionalen Programmierung der Ablauf nicht im Detail festgelegt ist. Ein Beispiel hierfür ist die einfache automatische Parallelisierung. Dies kann mit zunehmender Bedeutung von paralleler Hardware wichtig werden.
- Funktionale Programme können problemlos für Multithreading programmiert werden. Bei prozeduraler Programmierung ist dies extrem fehleranfällig (Wettlaufbedingungen).
- Funktionale Programmierung fördert die Datenkapselung. Datenstrukturen können unbedenklich an andere Programmeinheiten weitergegeben werden. Sie können ja nicht verändert werden. In prozeduralen Programmen muss man hierfür besondere Vorkehrungen treffen.

- Auf Funktionen höherer Ordnung aufgebaute Programme sind kürzer als äquivalente prozedurale Programme.
- Dadurch, dass Funktionen erstklassige Objekte sind, lassen sich *Kontrollabstraktionen* implementieren. Ein prominentes Beispiel ist die Implementierung der Receive-Case-Anweisung in Scala. Die Formulierung von Kontrollabstraktionen ermöglicht die Implementierung (interner) Domänen--spezifischer Sprachen (DSL).
- Wie die Beispiele Smalltalk und Scala zeigen, lässt sich funktionale Programmierung gut mit Objektorientierung kombinieren.

Grenzen

Die Nachteile und Grenzen des funktionalen Paradigmas liegen in unterschiedlichen Bereichen.

- Die funktionale Programmierung verfolgt Konzepte, die nicht unmittelbar von den Eigenschaften eines Computers abgeleitet sind. Es ist kein Wunder, dass es nicht so einfach ist, eine effiziente Implementierung vorzunehmen.
- Die Unveränderlichkeit von Datenstrukturen bedingt, dass häufig größere Datenmengen kopiert werden müssen.
- Die meisten Programmierer sind „imperativ“ erzogen. Es fällt ihnen nicht leicht, effiziente funktionale Programme zu schreiben.
- Die Welt ist nicht vollständig funktional! Viele Sachverhalte lassen sich leichter mit veränderlichen, zustandsbehafteten Objekten modellieren.

Diese Begrenzungen haben immer schon dazu geführt, dass die „reine“ funktionale Programmierung um andere Konzepte ergänzt wurde. In der hier behandelten Programmiersprache Scala ist das auch so. Neben der funktionalen Programmierung steht der volle Umfang objektorientierter Programmierung einschließlich aller prozeduralen Aspekte zur Verfügung.

5.3 Funktionale Programmierung am Beispiel Scala

Die wichtigsten Eigenschaften von Scala wurde im vorigen Kapitel gesprochen. Hier geht es demnach nicht um eine Sprachbeschreibung sondern um die Erläuterung der wichtigsten Merkmale funktionaler Programmierung.

5.3.1 Funktionsdefinition und Funktionsanwendung

Die Syntax der Funktionsdefinition wurde auch schon im letzten Kapitel besprochen. In diesem Abschnitt sollen die Grundelemente der funktionalen Programmierung verdeutlicht werden. Dazu braucht man Beispiele. Diese sind in Scala formuliert. Es geht dabei stets um das Konzept der funktionalen Programmierung. Es geht nicht um Scala!

Zunächst wird an einem Beispiel erläutert, wie man in der funktionalen Programmierung *Funktionsanwendungen* verstehen kann, ohne unbedingt einen Ablauf nachvollziehen zu müssen. Der Kern der Funktionsanwendung besteht dabei in dem Umschreiben des „Funktionsaufruf“ in den „Funktionskörper“. In welcher Reihenfolge man die gleichzeitig möglichen Ersetzungen vornimmt, hat bei einer funktionalen Sprache keine Auswirkung auf das Ergebnis. Ich gehe *hier* wie bei einem prozeduralen Ablauf so vor, dass ich zunächst die Parameterausdrücke auswerte und dann die Ergebnisse an die einzelnen Funktionsparameter binde.

```
object MeinProgramm {
  def main(args: Array[String]) {
    println(f(7))
  }
  def f(n: Int) = g(n-2) * g(n+1)
  def g(n: Int) = n * n
}
```

Es soll herausgefunden werden, welche Ausgabe ausgegeben wird oder was der Wert von `f(7)` ist. Dazu wird eine Folge von Funktionsanwendungen durchgeführt. Die geschweiften Klammern sind hier nur eine andere Schreibweise der Klammerung. Sie sollen daran erinnern, dass es sich um die Auswertung eines Funktionskörpers handelt. Wenn man keinen Wert auf die Verdeutlichung des exakten Ablaufs legt, kann man sie getrost weglassen oder durch einfache Klammern ersetzen.

```
f(7) =
{g(7-2) * g(7+1)} =
{g(5) * g(7+1)} =
{{5 * 5} * g(7+1)} =
{25 * g(7+1)} =
{25 * g(8)} =
{25 * {8 * 8}}=
{25 * 64} =
1600
```

In dem Beispiel wurde in jeder Zeile genau eine einzige Funktionsanwendung vorgenommen. Dabei wurde die willkürliche Strategie verfolgt, immer zunächst die linkeste Funktion auszuwerten. Man hätte aber auch ganz anders vorgehen können, nämlich zunächst die rechteste Funktion auszuwerten:

```
f(7) =
{g(7-2) * g(7+1)} =
{g(7-2) * g(8)} =
{g(7-2) * {8 * 8}} =
{g(7-2) * 64} =
{g(5) * 64} =
{{5 * 5} * 64}=
{25 * 64} =
1600
```

Es ist typisch für die funktionale Programmierung, dass die Auswertungsreihenfolge keine Rolle spielt. Am kürzesten und am verständlichsten ist vermutlich, wenn man stets alle Anwendungen einer Schachtelungsebene gleichzeitig vornimmt.

```
f(7) =
{g(7-2) * g(7+1)} =
{g(5) * g(8)} =
{{5 * 5} * {8 * 8}} =
{25 * 64} =
1600
```

Allen Formen der Darstellung der Auswertung eines funktionalen Ausdrucks ist gemein, dass sie einfach als eine Folge von Umformungen geschrieben werden können. Das sieht nicht zufällig so aus wie die Umformung mathematischer Formeln!

5.3.2 Funktionen

Funktionsliterale

Funktionliterale, auch Closure, oder Funktionsobjekt genannt, wurden ja schon mehrfach angesprochen. Hier sollen ein paar Beispiele zu ihrer Verwendung stehen. Es wird auch gezeigt, wie sie in Java (bis Java 7) durch anonyme Klasse nachgebildet werden.

In funktionalen Sprachen sind alle Funktionsobjekte letztlich Closures. Das Konzept ist laut Definition umfassender als die bloße Verwendung von Funktionsobjekten. Es bedeutet, dass ich Funktionen in jedem beliebigen Kontext definieren kann, also auch „lokale“ Funktionen. Egal wo diese Funktionen später aufgerufen werden, sie „erinnern“ sich immer an den Ort ihrer „Geburtsumgebung“.

Die folgende Definition erlaubt es, quadratische Funktionen per Funktion zu definieren:

```
def quadratic(a: Double, b: Double, c: Double) =
  (x: Double) => (a * x + b) * x + c
```

Durch den Aufruf von `quadratic` wird ein anonymes Funktionsobjekt erzeugt und als Resultat zurück gegeben. Die Closure-Eigenschaft kommt darin zum Ausdruck, dass dieses Funktionsobjekt die bei der Definition verwendeten Werte von `a`, `b` und `c` mit sich trägt.

Der folgende Ablauf des Scala-Interpreters macht das deutlich:

```
scala> val a = quadratic(1, 0, 0)
a: (Double) => Double = <function1>

scala> a(3)
res0: Double = 9.0

scala> val b = quadratic(1,0,10)
b: (Double) => Double = <function1>

scala> b(3)
res1: Double = 19.0

scala> a(3)
res2: Double = 9.0
```

Auch wenn die Funktionsobjekte der Variablen `a` und `b` durch Aufruf der Funktion `quadratic` die gleichnamigen lokalen Variablen nutzen, so handelt es sich bei den freien Variablen `a`, `b` und `c` jeweils ja doch um eine neue Variablenumgebung, die zu der jeweiligen Ausführung der Funktion `quadratic` gehört.

Anonyme Klasse als Closure

Zur Abgrenzung soll das letzte Beispiel in Java formuliert werden.[4]. Das Beispiel soll den Zusammenhang zwischen Closure und anonymer Klasse verdeutlichen.

In Java benötigen wird zunächst ein Interface (das ist in Scala halt schon so vordefiniert).

```
public interface Function1<T,R> {
  public R apply(T x);
}
```

Damit können wir nun unser `quadratic`-Objekt definieren:

```
Function1<Double, Double> quadratic(
  final double a, final double b, final double c)
{
  return new Function1<Double, Double>() {
    public Double apply(Double x) {
      return (a * x + b) * x + c;
    } };
}
```

Hier sind ein paar Kleinigkeiten zu beachten. Die Typparameter müssen Referenztypen sein, deshalb steht dort `Double`. Die in der anonymen Klasse verwendeten lokalen Variablen müsssen `final` sein. Die Parameter `a`, `b`, `c` der Parabelfunktion können `Double` oder `double` sein. Die Unterschiede werden in jedem Fall durch Autoboxing verdeckt.

Und schließlich können wir dies anwenden:

```
Function1<Double, Double> a = quadratic(1, 0, 0);
Function1<Double, Double> b = quadratic(1, 0, 10);
System.out.println(a.apply(3));
System.out.println(b.apply(3));
```

Man erkennt hier auch den Ballast, den vollständige statische Typangaben und Typparameter mit sich bringen. Nicht umsonst sind ungetypte Sprachen populär.

Es ist nicht ganz verkehrt, wenn Sie in Scala nur ein verbessertes Frontend zu Java sehen. Der Scala-Compiler erzeugt für Closures letztlich Klassen, die so ähnlich wie dieses Java-Beispiel aussehen.

[4] Es geht nicht darum, Java schlecht aussehen zu lassen. Java unterstützt funktionale Programmierung und Closures ja ganz bewusst nicht.

Currying

Der Begriff Currying geht auf die Mathematiker Moses Schönfinkel und Haskell Curry zurück. Vereinfacht geht es darum, eine Paramerliste mit mehreren Parameter durch mehrere Funktionen mit jeweils einem Parametern darzustellen. Eingeführt wurde diese Technik für theoretische Untersuchungen über berechenbare Funktionen. In funktionalen Programmiersprachen, wie in Scala, wird Currying aber häufig auch dazu verwendet, eine bewusst gewollte Schreibweise zu erreichen.

Definition:

Unter Currying versteht man die Darstellung einer mehrparametrigen Funktion durch einparametrige Funktionen, die jeweils eine weitere Funktion definieren. Durch eine Verkettung mehrerer Funktionen kann man schließlich den gleichen Effekt wie bei der Anwendung einer einzigen mehrparametrigen Funktion erreichen.

Beispiel:

```
// normale Funktion mit 2 Parametern.
def summe(a: Int, b: Int) = a + b

// Anwendung
val sum_1_plus_3 = summe(1, 3)

// Currying = definiert über Funktionsobjekte
def summe(a: Int) =  (b: Int) => a + b

// Anwendung
val sum_2_plus_4 = summe(2)(4)

// abgekuerzte Definition
def summe(a: Int)(b: Int) = a + b

// Anwendung
val sum_7_plus_3 = summe(7)(3)
```

Eine der Anwendungen des Currying liegt darin, zunächst nicht alle Parameter festzulegen und so eine Funktion der restlichen Parameter zu definieren. Dies wird dann im nächsten Abschnitt erläutert.

In Scala bietet Currying den Vorteil, dass dadurch die Typinferenz unterstützt wird. Datentypen, die der Compiler bei den ersten Parameterlisten erkannt hat k"önnen in den später anzuwendenden Parameterlisten zur Typinferenz herangezogen werden.

Partielle Evaluierung einer Funktion

Es ist eine wichtige Grundlage der funktionalen Programmierung, dass man Operationen auf Funktionen selbst besitzt. Eine solche Operation ist die Möglichkeit, aus vorhandenen Funktionen neue Funktionen herzuleiten, indem man einige Parameter festlegt. Umgekehrt kann man das auch so beschreiben, dass bei einem Funktionsaufruf nur ein Teil der Parameter ausgewertet wird.

Definition:

Unter partieller Evaluierung versteht man die teilweise Festlegung der Funktionsparameter. Das Resultat ist eine neue Funktion, die den restlichen Parametern einen Ergebniswert zuordnet.

In dem folgenden Beispiel leiten wir aus der Summenfunktion eine Teilfunktion her. Zunächst können wir das Ziel durch Currying erreichen.

```
def defIncrement(a: Int) =  (b: Int) => a + b
val plus3 = defIncrement(3)

// die Anwendung ergibt den Wert 10 = 3 + 7
plus3(7)
```

Hierbei ging es noch nicht um partielle Auswertung. Diese entsteht erst bei der Ersetzung nicht festgelegter Parameter durch einen Wildcard-Ausdruck:

```
def summe(a; Int, b: Int) = a + b
val plus3 = summe(3, _:Int)
plus3(7)
```

Mittels dieser Methode können wir beliebige Parameter als nicht evaluiert festlegen.

```
def addMult(x: Int, y: Int, z: Int) = x + y * z
val addTwice = addMult(_:Int, 2, _:Int)
println(addTwice(7, 5))  // Ausgabe = 15
```

In ähnlicher Form kann man auch mit dem Currying verfahren, um erst teilweise festgelegte Funktionen zu definieren.

```
def summe(a: Int)(b: Int) = a + b
println(summe(2)(7))     // Ausgabe = 9
val plus3: Int=>Int = summe(2)
val plus2 = summe(2) _   // Syntax mit _ !
println(plus2(7))        // Ausgabe = 9
println(plus3(7))        // Ausgabe = 10
```

5.3.3 Rekursion

Rekursion ist die Zurückführung einer Funktionsdefinition auf sich selbst. Die Rekursion muss bei der Programmierung zu einer berechenbaren Vorschrift führen. Als Beispiel diene die bekannte Definition der Fakultätsfunktion.

```
def f(n: Int): Int  = if (n == 0) 1 else n * f(n - 1)
```

Auch hier können wir mit Funktionsanwendungen „per Hand“ das Ergebnis ermitteln:

```
f(3) =
{if (3 == 0) 1 else 3 * f(3 - 1)} =
{3 * f(3 - 1)} =
{3 * f(2)} =
{3 * {if (2 == 0) 1 else 2 * f(2 - 1)}} =
{3 * {2 * f(2 - 1)}} =
{3 * {2 * f(1)}} =
{3 * {2 * {if (1 == 0) 1 else 1 * f(1 - 1)}}} =
{3 * {2 * {1 * f(1 - 1)}}} =
{3 * {2 * {1 * f(0)}}} =
{3 * {2 * {1 * {if (0 == 0) 1 else 0 * f(0 - 1)}}}} =
{3 * {2 * {1 * 1}}} =
{3 * {2 * 1}} =
{3 * 2} =
6
```

Hier habe ich mal wieder minutiös jeden Schritt dargestellt. Wir können das extrem kürzen, wenn wir die Evalution von einfachen Ausdrücken direkt komplett durchführen und die if's direkt im Kopf auswerten:

```
f(3) =
{3 * f(2)} =
{3 * {2 * f(1)} =
{3 * {2 * {1 * f(0)}}} =
{3 * {2 * {1 * 1}}} =
{3 * {2 * 1}} =
{3 * 2} =
6
```

Man erkennt an dieser Reihenfolge deutlich den „doppelten" Weg der Rekursion. Auf dem „Hinweg" (in dem Beispiel von `f(3)` zu `f(0)` wird der auszuwertende Ausdruck immer länger, da ja noch nichts berechnet wird. Erst nachdem die Abbruchbedingung erreicht wurde, wird auf dem „Rückweg" das Ergebnis nach und nach aufgebaut.

Wie Sie wissen, hat diese Form der Rekursion den Nachteil, dass eine Reihe von Stackframes aufgebaut werden müssen, die Kopien der Funktionsargumente und eventuell auch lokale Variablen vorhalten.

In einer besonderen Form der Rekursion, nämlich der *Endrekursion*, kann der Compiler das Programm ohne den Aufbau von zusätzlichen Stackframes übersetzen. Der Binärcode und die Ausführung eines solchen Programms sind von der Ausführung eines iterativen Programms nicht zu unterscheiden.

Definition:

*Eine rekursive Funktion ist **endrekursiv**, wenn nach dem rekursiven Aufruf innerhalb der Funktion keine Operation mehr auszuführen ist. Die **Endrekursionsoptimierung** bewirkt die Übersetzung einer endrekursiven Funktion in einen iterativen Ablauf.*

Die eben beschriebenen Fakultätsfunktions ist *nicht* endrekursiv, denn nach dem Aufruf muss noch eine weitere Vereinfachung, nämlich die Multiplikation, angewendet werden.

Die Umwandlung einer rekursiven Funktion in eine endrekursive Form ist meist relativ einfach. Die Grundidee ist (mindestens) eine weitere Variable einzuführen,

in der auf dem Hinweg das Ergebnis nach und nach aufgebaut wird. Diese Variable, heißt auch *Akkumulatorvariable* (akkumulieren = sammeln).

In aller Regel entstehen aus einer einzigen rekursiven Funktion zwei Funktionen. Ein davon ist die endrekursive Form und die zweite Funktion wird als öffentlich sichtbare Aufruf-Schnittstelle und zur Initialisierung des Rekursionsanfangs verwendet.

```
// fac(n) = n!
def fac(n: Int) = f(n, 1)

// f(n, p) = n! * p
def f(n: Int, p: Int): Int =
  if (n == 0) p else f(n - 1, n * p)
```

```
fac(3) =
f(3, 1) =
f(2, 3) =
f(1, 6) =
f(0, 6) =
6
```

In diesem Fall wurden die unnötigen geschweiften Klammern weggelassen. Übrig bleibt eine bloße Gleichungsumformung. Die Funktion f kann auch mathematisch (nicht programmtechnisch) beschrieben werden durch die Gleichung:

$$f(n, p) = n! \cdot p$$

Weiter gilt dann auch:

$$f(n, p) = n! \cdot p = f(n - 1, n \cdot p) = (n - 1)! \cdot n \cdot p$$

Wenn man das weiß, lässt sich die Korrektheit der Gleichungsumformungen und des Programms leicht einsehen.

Schließlich kann ich die Funktion so umschreiben, dass sie der „normalen“ Iteration entspricht. Hierbei wird anstelle dem abwärtszählenden `n` eine ab `1` aufwärtszählende Variable `i` verwendet. Gleichzeitig wird durch das Beispiel illustriert, dass man in funktionalen Sprachen eine Funktion lokal definieren kann. Sie ist dadurch einerseits nach außen nicht sichtbar. Andererseits kann man in der eingebetteten Funktion auf die Variablen der Umgebung zugreifen.

Zum Vergleich sind neben der funktionalen, rekursiven Definition anschließend eine iterative Formulierung und die Formulierung mit höheren Datenstrukturen angegeben.

```
// endrekursiv (funktional)
def fac(n: Int) = {
  // f(i, p) = p * n! / (i - 1)!
  // d.h. wenn p == (i - 1)!, dann f(i,p) = n!
  @tailrec
  def f(i: Int, p: Int): Int =
    if (i <= n) f(i + 1, i * p) else p
  f(1, 1)
```

```
}

// Iteration mittels while (imperativ)
def facWhile(n: Int) = {
  var i = 1
  var p = 1
  // Invariante: p = (i-1)!
  while (i <= n) {
    p *= i
    i += 1
  }
  p
}

// Definition mittels Range-Objekt
def facRange(n: Int) = (1 to n) product
```

Man kann der endrekursiven Funktion `f` auch eine deutliche Erklärung geben:

$$f(i, p) = n!/(i-1)! \cdot p$$

.

Daraus folgt dann auch die Initialisierung fac$(n) = f(1, 1)$. Klingt kompliziert? Ist letztlich aber einfacher, als das Nachvollziehen der Schleifeninvariante in der iterativen Fassung.

Beachten Sie das Auftreten der `var`-Deklarationen in der iterativen Fassung. Veränderliche Variable sind immer ein Zeichen eines imperativen Vorgehens[5]

Nehmen wir ein letztes einfaches Beispiel, nämlich die Fibonacci-Funktion:

```
// rekursive Fassung (funktional)
def fibRec(n: Int): BigInt =
  if (n <= 1) 1 else fib_rec(n - 1) + fibRec(n - 2)

// endrekursive Fassung (funktional)
def fibTailrec(n: Int) = {
  @tailrec
  def fib(i: Int, f: BigInt, g: BigInt): BigInt =
    if (i <= n) fib(i + 1, f + g, f) else f
  fib(2, 1, 1)
}

// iterative Fassung (imperativ)
def fibFor(n: Int) = {
  var g = BigInt(1)
  var f = BigInt(1)
  for (i <- 2 to n) {
    val t = f
    f = f + g
    g = t
  }
  f
}
```

[5]Die Ausnahme sind manchmal mögliche Optimierungen, wie das „Caching“ von bereits gefundenen Funktionswerten.

Da bei der Fibonacci-Funktion sehr große Ergebnisse vorkommen können, habe ich den Datentyp `BigInt` verwendet.[6]

Sie werden bemerkt haben, dass `fibRec` die einfache extrem ineffiziente Variante aus dem ersten Semester ist. Die endrekursive Lösung steht dagegen (auch ohne die Compileroptimierung) der imperativen Lösung hinsichtlich der Laufzeit in nichts nach.

An dem Beispiel kann man einen weiteren Unterschied zwischen funktionaler und imperativer Programmierung erkennen. Imperative Programme bestehen immer aus einer Folge von Anweisungen. Funktionen sind oft durch eine einzige Gleichung, manchmal ergänzt um die Definition von Hilfsfunktionen, zu beschreiben.

5.3.4 Operationen auf Funktionen

Funktionen sind Objekte. Die Konsequenz ist, dass man Funktionen schreiben kann, die aus bestehenden Funktionen neue Funktionen erzeugen. Ein einfaches Beispiel ist die folgende Funktion `compose`. Wenn man ihr zwei Funktionen f und g übergibt, erhält man als Resultat eine neue Funktion, deren Vorschrift darin besteht, dass zunächst g und dann auf das Ergebnis f angewendet wird (compose$(f, g)(x) = f(g(x))$).

```
def compose[R, S, T](f2: R=>T, f1: S=>R) =
  (x: R) => f2(f1(x))

val quadrat_plus_1 =
  compose((x:Double) => x + 1, (x:Double) => x * x)
```

5.3.5 Partielle Funktion

Totale Funktionen (das sind die „normalen“ Funktionen) sind für jedes Element des Definitionsbereichs definiert. Dabei kann es nötig sein, verschiedene Bereiche hinsichtlich der Funktionsgleichung zu unterscheiden. Nehmen wir als Beispiel die Definition des Absolutbetrags:

```
def abs(x: Double) = x match {
  case 0          =>  0
  case a if a > 0 => +a
  case a if a < 0 => -a
}
```

Die Funktion wurde bewusst unnötig ausführlich geschrieben um die Abdeckung des Definitionsbereichts zu verdeutlichen. Was ist, wenn einer der drei Fälle fehlt? Dann ist die Funktion nicht mehr für alle Gleitkommazahlen definiert. Wir haben dann eine partielle Funktion.

Als Beispiel für eine partielle Funktion mag die Fakultät herhalten, die bekanntlich nur für natürliche, d.h. für positive reelle Zahlen definiert ist.

Die übliche Definition einer Fakultätsfunktion:

[6] Das ist letztlich eine Verpackung der Java-Klasse `BigInteger`.

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

ist formal eine totale Funktion, also für alle ganzen Zahlen definiert. Natürlich macht der Aufruf für negative Zahlen keinen Sinn, wird aber halt nicht überprüft. Anders sieht es bei der folgenden Formulierung aus.

```
def fac(n: Int): Int = n match {
  case 0 => 1
  case x if x > 0 => n * fac(n - 1)
}
```

Die Match-Anweisung deckt nicht alle möglichen Fälle ab. Nur für die sinnvollen Fälle wird ein Funktionswert angegeben. Die Funktion ist demnach nur partiell definiert.

Wir haben in diesem Beispiel aber noch kein Funktionsobjekt. Dieses können wir z.B. durch die folgende Anweisung erzeugen:

```
def fac: PartialFunction[Int,Int] = {
  case 0 => 1
  case n if n > 0 => n * fac(x)
}
```

Die geklammerten Case-Anweisungen stellen ein Funktionsobjekt dar. Genauer ist es in unserem Fall eine partielle Funktion (`PartialFunction`), die einen Teilbereich von `Int` auf `Int` abbildet.[7]

Partielle Funktionen können für die Argumente, für die sie definiert sind, ganz normal aufgerufen werden. Für undefinierte Fälle wird eine `MatchException` geworfen. Das Besondere der Objekte von `PartialFunction` ist aber, dass wir jetzt im Voraus abfragen können, ob die Funktion definiert ist. Die entsprechende Methode heißt `isDefinedAt`.

```
object Compute {
  def fac: PartialFunction[Int,Int] = {
    case 0 => 1
    case n if n > 0 => n * fac(n - 1)
  }

  def main(args: Array[String]) {
    print("Eingabe einer ganzen Zahl: ")
    val zahl = readInt
    berechne(fac, zahl)
  }

  def berechne[T](f: PartialFunction[T,T], n: T) =
    if (f isDefinedAt n)
      println("Der Funktionswert ist " + f(n))
    else
      println("Die Funktion ist nicht definiert")
}
```

[7] Die Funktion `fac` ist eine Funktion, die als Ergebnis ein Objekt einer partiellen Funktion zurückgibt.

Das etwas längere und vollständige Beispiel zeigt auch wie Funktionsobjekte einfach übergeben, auf ihre Definition abgefragt und schießlich ausgewertet werden können. Ein besonderer match-Ausdruck erscheint nirgends.

Merksatz:

Partielle Funktionen werden uns noch bei der Programmierung von Nebenläufigkeit mit Aktoren begegnen!

5.3.6 Call by Name und Kontrollabstraktion

Sie kennen aus Algorithmen und Programmierung zwei Mechanismen für die Parameterübergabe. Der Grund ist, dass es diese beiden Mechanismen in C gibt. Grundsätzlich gibt es noch weitere Mechanismen. Ohne Anspruch auf Vollständigkeit kann man die folgenden unterscheiden:

call by value Dabei werden Kopien der Parameter in der Funktion verwendet. Funktionale Sprachen wie Scala erlauben nicht einmal eine lokale Veränderung dieser Parameter. In der prozeduralen Programmierung heißt der Mechanismus auch *copy in*.

call by reference Hier wird die Adresse einer Variablen übergeben, so dass ihre Inhalte auch von der Funktion verändert werden können. Typisch prozedural.

copy out, copy inout Hier wird ebenfalls eine Variable übergeben. Der Compiler hat etwas bessere Kontrolle als bei der Referenzübergabe. Außerdem sind die sehr technisch aussehenden Dereferenzierungen usw. wie bei C nicht nötig. Dies ein etwas neuerer prozeduraler Mechanimus.

call by name Hier steht der Parameter für einen übergebenen Ausdruck. Der Ausdruck wird jedesmal erneut ausgewertet, wenn auf den Parameter zugegriffen wird. Im funktionalen Sinn kann man call by name als Übergabe eines Funktionsobjekts ansehen.

Als Fazit bleibt von der Liste, dass die funktionale Programmierung genau zwei Mechanismen unterstützt, nämlich call by value und call by name.

Definition:

*Unter **call by value** versteht man einen Übergabemechnismus für Funktionsparameter. Vor dem Aufruf der Funktion werden alle By-Value-Argumente ausgewertet. Die Ergebniswerte werden an die formalen Funktionsparameter gebunden.*

*Unter **call by name** versteht man einen Übergabemechanismus für Funktionsparameter. Beim Aufruf der Funktion werden By-Name-Argumente nicht ausgewertet. Statt dessen werden sie als Funktionsobjekt an die Funktionsparameter gebunden. Bei jeder Verwendung eines By-Name-Parameters findet dann eine erneute Auswertung des Argumentausdrucks statt.*

Man kann call by name als eine Optimierungsstrategie ansehen. Die Auswertung des Argumentausrucks wird nämlich auf später verschoben. Wenn der Wert schließlich nicht benötigt wird, kann die Auswertung unterbleiben. Andererseits

kann call by name aber auch den Nachteil haben, das der gleiche Werte wiederholt ermittelt werden muß.

Den Optimierungsaspekt kann man an dem folgenden Beispiel nachvollziehen.

```
var Debug = true

def log(meldung: String) {
  if (Debug) println(meldung)
}

...
log("Liste: " + liste.toString)
...
```

In diesem Szenario ist angenommen, dass wir in einem Logging-System (das kann natürlich eine Bibliotheksklasse sein) über Methoden verfügen, die es erlauben, Meldungen auszugeben. Die Meldungen sollen nur ausgegeben werden, wenn die Variable `Debug` auf `true` steht.

Das Problem ist nun, dass dieser Mechanismus selbst dann erhebliche Rechenzeit kosten kann, wenn wir keine Meldungen haben wollen. Wir können natürlich die Ausgabe unterdrücken, wenn wir `Debug` gleich `false` setzen. Allerdings wird dann immer noch `log` aufgerufen und, noch schlimmer, es werden zeitaufwändige Berechnungen, wie die Umwandlung von Datenstrukturen in Strings ausgeführt. In Scala können wir dies mit call by name lösen:

```
var Debug = true

def log(meldung: =>String) { // by name !!
  if (Debug) println(meldung)
}

...
log("Liste: " + liste.toString)
...
```

Der einzige Unterschied besteht in dem Datentyp des Parameters `meldung`. Die Schreibweise `=>String` kennzeichnet die Übergabe als call by name. Die Syntax legt nahe, den Parameter als eine Funktion aufzufassen, die bei Aufruf einen String liefert. Wir erhalten also eine Optimierung, da jetzt bei ausgeschaltetem Debugging das Argument des Aufrufs von `log` nicht mehr ausgewertet wird.

Dies ist eine bloße Optimierung. Weitaus wichtiger ist die durch call by name gegebene Möglichkeit der Formulierung eigener Kontrollabstraktionen.

Definition:

*Unter **Kontrollabstraktion** versteht man die Implementierung von Kontrollstrukturen durch Bibliotheksfunktionen. Kontrollabstraktion erlaubt die spezialisierte Einführung von besonderen Kontrollstrukturen. Damit ist oft eine Modularisierung des Codes möglich, die in anderen Programmiersprachen nur durch spezialisierte Konstrukte erreicht werden können (oder auch nicht). Sie stellt auch die Grundlage für die Formulierung von domänen spezifischen Spracherweiterungen dar (DSL).*

Mittels Kontrollabstraktion lassen sich grundsätzlich sogar die bereits vorhandenen Kontrollstrukturen in Scala selbst programmieren. Ein Beispiel ist die Definition der While-Anweisung. Ich nenne sie hier `solange`:

```
def solange(bedingung: => Boolean)(anweisungsBlock: => Unit)
{
  if (bedingung) {
    anweisungsBlock
    solange(bedingung)(anweisungsBlock)
  }
}

 ...
 var i = 1
 solange (i <= 10) {
   println(i)
   i = i + 1
 }
```

Sieht schön aus, ist aber so nicht wirklich notwendig, da es ja schon das While gibt (ist natürlich optimaler implementiert). Abgesehen davon, dass man ähnliche Erweiterungen selbst hinzufügen kann, verfügt Scala bereits über ein wichtiges Beispiel der Anwendung der Kontrollabstraktion.

Wie wir bei der Besprechung von Nebenläufigkeit sehen werden, ist das Actor-Konzept ausschließlich durch Bibliotheksfunktionen und dort definierte Kontrollabstraktionen implementiert.

Call by name und Kontrollabstraktionen waren im Umfeld der funktionalen Sprachen schon immer bekannt (spätestens seit Ende der 50er). Auch Smalltalk, die erste wirklich objektorientierte Sprache (1980) verfügt darüber.

5.4 Algebraische Datentypen in Scala

5.4.1 Algebraische Datentypen

Der Begriff *algebraischer Datentyp* entstammt aus der Typtheorie. Algebraische Datentypen werden häufig als ADT abgekrürzt. Das ist insofern missverständlich als der Begriff Äbstrakter Datentyp"ganz was anderes bedeuten, nämlich die abstrakte Beschreibung von Datentypen über ihr Verhalten. Demgegenüber beschreiben algebraische Typdefinitionen eher die Struktur zusammengesetzter Typen.

Definition:

Ein algebraischer Datentyp definiert strukturierte Typen auf der Basis von Grundtypen. Die Typdefinition verwendet die Grundoperationen Produkt *und* Summe*. Die Produktoperation kann man durch einen Konstruktor beschreiben, der mehrere Elemente von anderen Datentypen als kartesisches Produkt zusammenfasst. Die Summenoperation fasst unterschiedliche Typen zu der Einheit eines Obertyps zusammen. Eine besondere Rolle spielen rekursive Typdefinition (hier kommt der Obertyp als Produktelement eines Untertyps wieder vor). Auf algebraischen Datentypen definierte Funktionen machen starken Gebrauch von* Pattern-Matching*.*

Ehe ich im nächsten Abschnitt auf die Realisierung in Scala eingehe, möchte ich hier die prägnanteren Formulierungen aus streng funktionalen Sprachen vorstellen.

5.4.2 Realisierung algebraischer Datenstrukturen mit Case-Klassen

Die Realisierung algebraischer Datenstrukturen in einer Programmiersprache verlangt mehrere Eingenschaften, die in Scala (bewusst) alle gegeben sind:

- Die bequeme Definition eines Objekts über einen Konstruktor (*Produktdefinition*)
- Die bequeme Darstellung von Untertypen mittels Vererbung (*Summe*)
- Pattern-Matching mittels Case-Klassen.
- Unveränderlichkeit der Daten, Vergleichsoperation etc. durch Case-Klassen.

Der einfachste Datentyp (ohne) Werte heißt in Scala `Unit`.

Das grundlegende Muster einer algebraischen Datenstruktur sei am Beispiel des nachgebildeten Boole'schen Typs gezeigt[8].

```
sealed abstract class Bool
case object True extends Bool
case object False extends Bool

def or(a: Bool, b: Bool): Bool = (a, b) match {
    case (False, False) => False
    case _              => True
}
```

Wie gesagt, das Beispiel dient nur der Illustration. Es sind die folgenden Anmerkungen zu machen:

- Eine `sealed abstract class` ist eine abstrakte Oberklasse, deren sämtliche Unterklassen in der aktuellen Übersetzungseinheit stehen müssen. Dadurch wird die Anzahl der Varianten bleibend festgelegt, so dass der Compiler Fehler melden kann, wenn man beim Patternmatching eine Variante vergisst.
- Da die Konstruktoren `True` und `False` über keine Parameter verfügen, gibt es jeweils nur ein Objekt.
- Der Präfix `case` besorgt den nötigen Komfort.
- An dem Beispiel sehen wir, wie die Vererbung die „Summenoperation" ausdrückt.
- Die nötigen Operationen des Datentyps werden durch Funktion ausgedrückt.

[8] Die Nachbildung dient nur als Beispiel; sie bringt keine Vorteile.

An dieser Stelle sei angemerkt, dass Scala einigen syntaktischen „Zucker“ bereitstellt. Wenn man Operationen durch Methoden implementiert ist nämlich ebenso wie in anderen funktionalen Sprachen am Ende die Operatorschreibweise möglich.

Dies zeigt das leicht abgeänderte Beispiel:

```
sealed abstract class Bool {
    def or(b: Bool): Bool
}

case object True extends Bool {
    def or(b: Bool): Bool = True
}
case object False extends Bool {
    def or(b: Bool): Bool = b
}

// Verwendung als Operator
Bool x = a or b // = a.or(b)
```

Sinnvollere Beispiele sind Listen und Bäume. Beide sind rekursive Datenstrukturen. Damit die Beispiele gleich sinnvoller sind, werden hier auch gleich Typparameter mitverwendet.

Der oben angegebene Baumtyp lässt sich in Scala so schreiben[9]:

```
sealed abstract class Tree[+V]
case object Empty extends Tree[Nothing]
case class Leaf[V](value: V) extends Tree[V]
case class Node[V](left: Tree[V], right: Tree[V]) extends
   Tree[V]

def sumTree(t: Tree[Int]): Int = t match {
    case Empty      => 0
    case Leaf(v)    => v
    case Node(l, r) => sumTree(l) + sumTree(r)
}

val baum = Node(Leaf(17),
                Node(Leaf(2), Empty) )

val n = sumTree(baum) // n = 19
```

Listen und sequentielle Datenstrukturen spielen in der funktionalen Programmierung eine ganz zentrale Rolle. Sie werden daher in eine weiteren Abschnitt separat behandelt.

5.4.3 Algebraische Datenstrukturen und Objektorientierung

Die Tatsache, dass algebraische Datenstrukturen in Scala durch Klassen implementiert werden, wirft die Frage auf, was Klassen und algebraische Datenstrukturen miteinander zu tun haben.

Zunächst gibt es zwar keine Identität aber weitgehende Ähnlichkeiten:

[9] Der Typparameter `+V` ist als kovariant gekennzeichnet, dass bedeutet, dass Teilbäume auch einen Untertyp von `V` haben dürfen

- Die *Summe-Operation* wird in einer Klasse durch die Menge der Attribute erreicht.
- Die *Produkt-Operation* wird durch die mit der Klassenvererbung verbundene Typhierarchie ausgedrückt.
- Es ist in beiden Fällen möglich polymorphe Algorithmen zu formulieren.

Gleichzeitig ist es sehr instruktiv, die Unterschiede herauszustellen:

- Unterschiedliche Teiltypen erfordern eine unterschiedliche Implementierung auf der Operationen. In der Objektorientierung wird dies durch die Methodenauswahl durch die späte Bindung bewirkt. Algebraische Datentypen formulieren die Typauswahl explizit durch Mustererkennung.
- Algebraische Datentypen beschreiben unveränderliche Werte, Objekte kapseln einen (eventuell) veränderlichen Zustand.
- Die reguläre Struktur algebraischer Datenstrukturen ermöglicht einige Vereinfachungen (automatische Erzeugung der Gleichheit, des HashCodes und der Stringdarstellung (`toString`).
- Die objektorientiere Methodenauswahl ist sehr effizient und unterstützt sehr gut die Modularisierung des Programmcodes, dagegen ist das Pattern-Matching grundsätzlich mächtiger.

In rein funktionalen Sprachen, wie Haskell, nimmt das algebraisch strukturierte Typssystem viele Aufgaben der Objektorientierung. Haskell erlaubt es zudem eine zweiparametrige Funktion in Operatorschreibweise zu schreiben `f x y` kan auch als `x f y` werden. Scala erlaubt es den Methodenaufruf `x.f(y)` ebenfalls in der selben Form `x f y` zu schreiben. Wenn der erste Funktionsparamer gleich der Datenstruktur ist, sehen ähnliche Ausdrücke gleich aus.

Die wichtigste Besonderheit der Objektorientierung ist die größere Freiheit im Umgang mit Objekten und die enge Bindung der zulässignen Methoden an die Objekte. Der Vergleich macht aber auch deutlich, dass die Grenzen zwischen Objektorierntierung und Funktionaler Programmierung fließend sind. Scala macht sich diesen Umstand zunutze.

5.5 Funktionale Datenstrukturen

In C und Java haben Sie Arrays als grundlegende Datenstruktur kennen gelernt. Arrays sind eine Ansammlung von veränderlichen Variablen. Viele Algorithmen – ein Beispiel sind die Sortieralgorithmen – bestehen darin, einfach die Inhalte eines Arrays zu verändern. Arrays sind eine typische Datenstruktur der imperativen Programmierung.

In der funktionalen Programmierung haben die Arrays nur eine geringe Bedeutung. Grundsätzlich kann man auch mit Arrays alle möglichen Algorithmen funktional ausdrücken. Es gibt aber dabei das Problem, dass man dann bei jeder Veränderung das gesamte Array kopieren muss.

Anders sieht dies bei verketteten Listen aus. Wenn Listenobjekte nie verändert werden, lässt sich eine wichtige Optimierung einführen. Diese geht davon aus, dass Operationen am Listenanfang in $O(1)$ durchgeführt werden können.

Definition:

Die grundlegende Operation zum Erzeugen erweiterter Listen, ist die Cons-Operation (`::`). Logisch gesehen, wird der alten Listen eine neue Liste zugeordnet, die ein neues Element vorangestellt hat. Von der Implementierung her, haben die neue und die alte Liste alle Elemente, bis auf das erste, gemeinsam. Die grundlegenden Operationen zum Zerlegen von Listen sind `head` *(erstes Element) und* `tail` *(die Restliste ohne das erste Element). In Scala steht* `Nil` *für das Objekt der leeren Liste.*

```
val liste = List(1,2,3)
val ersteElement = liste.head
val restlicheElemente = liste.tail
val listeMitNull = 0::liste
```

Die Abbildung 5.1 zeigt, wie die grundlegenden Listenoperationen sich auf unterschiedliche Teile einer Liste beziehen. Dabei können verschiedene Listen Teile der Daten gemeinsam nutzen.

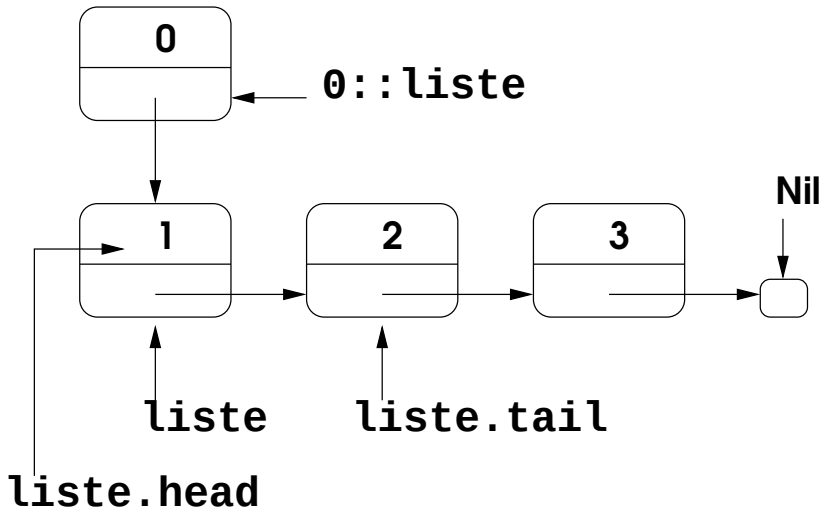

Abbildung 5.1: Funktionale Listenoperationen

Die eleganteste Form nehmen die Listenoperationen zusammen mit der Musterunterscheidung ein. Im Folgenden sind ein paar Beispiele in verschiedenen Varianten programmiert. Die Beispiele dienen der Illustration.

```
def isEmpty(liste: List) = liste == Nil

def length(liste: List[Any]): Int = liste match {
  case Nil => 0
  case _::tail => 1 + length(tail)
}
```

```
def length(liste: List[Any]): Int =
  if (liste == Nil) 0 else 1 + length(liste.tail)

def contains[T](liste: List[T], x: T): Boolean =
  liste match {
    case head::tail => (head == x) || contains(tail, x)
    case _ => false
  }

def contains[T](liste: List[T], x: T): Boolean =
  if (liste == Nil) false
  else if (liste.head == x) true
  else contains(liste.tail, x)

def append[T](liste1: List[T], liste2: List[T]): List[T] =
  liste1 match {
    case Nil => liste2
    case h::t => h::append(t, liste2)
  }

def append[T](liste1: List[T], liste2: List[T]): List[T] =
  if (liste1 == Nil) liste2
  else liste1.head::append(liste1.tail, liste2)
```

Alle Operationen sind bereits in der Bibliothek vorhanden. Ihr Aufruf sieht meist etwas anders aus, da in der Scala-Bibliothek alle Listen objektorientiert implmementiert sind. Die Append-Funktion wird durch den Operator `:::` ausgedrückt. Bei der Anwendung der Append-Funktion muss eine Kopie der Liste erstellt werde. Diese Operation ist damit – genauso wie `length` oder `contains` – in $O(n)$.

Die objektorientierte Darstellung von Listen in Scala ermöglich eine erhebliche Optimierung. Bei unveränderlichen Listen verhalten sich nämlich alle Listenoperationen streng funktional. Dies hindert die Scala-Implementierung jedoch nicht daran, die Methodenkörper prozedural zu realisieren.

Bei der Implementierung funktionaler Ausdrücke durch prozdeurale Abläufe ist das Aufrechterhalten der *referentiellen Integrität* zu beachten.

Definition:

*Ein Programm hat die Eigenschaft der **referentiellen Integrität**, wenn alle Vorkommen von Variablen durch den sie definierenden Ausdruck ersetzt werden können ohne das Ergebnis des Programms zu ändern.*

Referentielle Integrität bedeutet insbesondere, dass Funktionen keine Seiteneffekte haben dürfen. Wie sie intern funktionieren, ist dagegen egal.

Jedenfalls ergeben sich aus dem objektorientieren Konzept in Scala keine Nachteile. Anstelle des Funktionsaufrufs `contains(liste, "abc")` schreibt man halt `liste.contains("abc")`.

Hier sind ein paar typische Beispiele:

```
Nil.isEmpty           // ergibt true
val abc = List(1,2,3)
abc.isEmpty           // ergibt false
abc.length            // ergibt 3
List(3,4)::abc        // ergibt (3,4,1,2,3)
abc.exists(_ == 3)    // ergibt true
```

```
(List(3,4)::abc.count(_ == 3)
                        // ergibt 2
abc.filter(_ > 1)       // ergibt (2,3)
abc.map(x => x * x)     // ergibt (1, 4, 9)
```

Die Funktionen `exists`, `count`, `filter` und `map` haben als Argument ein Funktionsobjekt. In den Beispielen verwende ich auch die abgekürzte Schreibweise einer anonymen Funktion mit anonymen Variablen. Bei `map` ist das nichts möglich, da `x` zweimal auftritt. Immerhin braucht die Variable aber nicht deklariert zu werden, da der Typ aus dem Kontext hervorgeht.

5.6 Funktionen höherer Ordnung

Das letzte Beispiel hat es schon angedeutet: Die funktionale Programmierung bietet ganz andere Möglichkeiten der Programmierung von Operationen auf Datenstrukturen. Anstelle eine Operation auf allen Elementen durch Iteration oder Rekursion zu auszudrücken, rufen wir einfach eine Funktion auf, der wir eine Funktion mitgegeben, die auf jedes Element anzuwenden ist.

Definition:

*Eine **Funktion höherer Ordnung** ist eine Funktion, die ihrerseits Funktionen als Parameter oder als Ergebnis hat. Funktionen höherer Ordnung dienen oft dazu komplexe Operationen auf Datenstrukturen durchzuführen. Funktionen höherer Ordnung bieten auch die Grundlage für die Formulierung von* Kontrollabstraktionen.

Von Java her kennen Sie das eigentlich auch schon. Java hat aber nicht zum Ziel die funktionale Programmierung unterstützen. Solche Anwendungen sehen dort etwas schwerfällig aus und werden nur in besondere Fällen verwendet. In der Konsequenz werden in Java Funktionen höherer Ordnung auch nur dann genutzt, wenn sie deutliche Vorteile bieten.

Ein Beispiel ist das Sortieren nach besonderen Kriterien. Hier sollen mal Strings absteigend, statt aufsteigend sortiert werden. In Java können wir dazu eine anonyme `Comparator`-Klasse verwenden.

```
String[] a = { "Hans", "Karin", ... };
Arrays.sort(a, new Comparator<String>() {
  public int compare(String a, String b) {
    return - a.compareTo(b);
  }
});
```

Die anonyme Klasse dient dazu, eine Funktion (`compareTo`) zu verpacken. Das ist grundsätzlich nicht schlimm, es sieht halt nur etwas kompliziert aus. Gleichzeitig ist diese Anwendung aber immer noch nicht funktional, da ja die Inhalte des Arrays verändert werden.

In Scala lässt sich das Beispiel so schreiben.

```
val a = List{ "Hans", "Karin", ... }
```

```
val sortiert = a.sortWith(_ > _)
```

`sortWith` ist eine Methode der Klasse `List`. Ihr muss ein Funktionsobjekt übergeben werden, dass den Vergleich übernimmt. Wenn wir wollen, können wir so ein Objekt eigens definieren. Wir können aber auch, so wie hier, einfach eine `anonyme Funktion` übergeben. Die volle Schreibweise der anonymen Funktion ist etwas länger. Scala erlaubt halt, und das ist auch für andere funktionale Sprachen typisch, diesen Ausdruck kürzer zu schreiben. Die lange Form sieht so aus:

```
val a: List[String] = List[String]{"Hans", "Karin", ... }
val sortiert: List[String] =
  a.sortWith((x:String, y: String) => x > y)
```

Auch in Java sind solche funktionalen Anwendungen nicht so selten, wie vielleicht man denken mag. Denken Sie doch z.B. an die Aktionen, die man den GUI-Elementen zuordnet. Dort werden regelmäßig anonyme Klassen zum „Verpacken“ von Funktionen verwendet.

Allerdings bleibt in Java die Verwendung höherer Funktionen doch eine etwas kompliziert wirkende Struktur, die dann doch viel seltener verwendet wird, als dies bei der funktionalen Programmierung der Fall ist. Das folgende Beispiel zeigt einige typische Konstrukte.

```
// sum, product, max gibt es schon in der Scala-Library
def summe(liste: List[Double]) = liste.reduceLeft(_ + _)
def fakultaet(n: Int) = (BigInt(1) to n).reduceLeft(_*_)
def maximum(liste: List[Double]) = liste.reduceLeft(_ max _)

val quadrate = List(1, 2, 3, 4).map(x=>x*x)

val geradeZahlen = List(...).filter(_ % 2 == 0)

val enthaeltUngeradeZahl = List(...).exists(_ % 2 == 1)

def dotProduct(v1: List[Double], v2: List[Double]) = {
  require (v1.length == v2.length)
  (v1, v2).zipped.map(_ * _).sum
}

def anzahlBuchstaben(s: String, c: Char) =
  s.count(_ == c)

def ausgabe(liste: List[Any]) {
  liste.foreach(println(_))
}
```

Hier wurden die folgenden Funktionen verwendet:

- `reduceLeft`: Fasse von links beginnend alle Elemente mit der angegebenen Operation zusammen.
- `map`: Erzeugt eine neue Liste mit den Ergebnissen der Funktionsanwendung auf die einzelnen Listenelemente.

- `filter`: Erzeugt eine Liste mit den Elementen für die die angegebene Bedingung erfüllt ist.
- `exists`: gibt es ein Element, das die boole'sche Funktion erfüllt?
- `count`: gibt die Anzahl der Elemente zurück, die die angegebene Bedingung erfüllen.
- `zipped`: gruppiert die Listenelemente paarweise, so dass sie einfach verknüpft werden können.
- `sum`: summiert alle Elemente einer Liste oder eines Arrays.
- `foreach`: führt für jedes Element die Seiteneffekt behaftete Operation aus.

Wenn Sie in der Scala-API nachschauen, werden Sie noch mehr standardmäßig vordefinierte Listenfunktionen finden. Wenn man sich einmal daran gewöhnt hat, kann man viele Algorithmen kürzer und lesbarer schreiben. In manchen Fällen bevorzugt man aber auch gerne die For-Schleife. In Scala ist diese genau genommen nur eine andere Schreibweise für den Aufruf höherer Listenfunkten, `foreach` und `map`. Man spricht daher auch von der *for comprehension* (*comprehension = Abkürzung*.

Beispiele für die funktionale Anwendung von `for` sind:

```
// diese Iterationen entsprechen der funktionalen Form

val quadrate = for(x <- List(1, 2, 3, 4)) yield x * x

def ausgabe(liste: List[Any]) {
  for(x <-liste) println(x)
}
```

Andere Anwendungen von `for` sind aber prozedural, weil wir dabei in einer Folge von Anweisungen Veränderungen an Variablen durchführen.

```
// diese Iterationen sind prozedural

def summe(liste: List[Double]) = {
  var s = 0.0
  for (x <- liste) s += x
  x
}
```

Anmerkung:

Es sollte nicht unerwähnt bleiben, dass Funktionen höherer Ordnung in der Zukunft vielleicht allein aus Effizienzgründen in Mode kommen. Sie unterstützen nämlich ganz besonders die Formulierung von datenparallelen Anwendungen bei denen Rechenoperationen gleichzeitig auf möglichst vielen Datenelementen gleichzeitig ausgeführt werden. Datenparallelität (SIMD-Parallelität) unterstützt die sichere und effiziente Ausnutzung paralleler Hardware.

Im nächsten Kapitel werden einige der erwähnten Funktionen zusammen mit den Datenstrukturen in einem etwas allgemeineren Schema dargestellt.

Schließlich will ich noch zeigen, wie komplexere Algorithmen aussehen können. Hier der Quicksort (z.B. für Gleitkommazahlen):

```
def sort(liste: List[Double]): List[Double] = liste match {
  case Nil => Nil
  case pivot::rest =>
    val (small,large) = rest.partition(_ <= pivot)
    sort(small):::pivot::sort(large)
}
```

`partition` ist ebenfalls eine vordefinierte höhere Funktion. Sie gibt für eine Liste ein Paar von zwei Listen zurück `(small,large)`. Die erste der beiden Listen enthält die Elemente für die die angegebene Bedingung zutriff, die andere den Rest.

Kapitel 6

Funktionale Datenstrukturen

Es stellt sich nun die Frage, was Objektorientierung und funktionale Programmierung gemein haben.

Um uns dem Thema anzunähern, seien zunächst zwei Definitionen gegenübergestellt:

Definition:
Objektorientierung *kapselt veränderliche Daten und die sie verändernden und abfragenden Methoden in ein Objekt. Gleichartige Objekte werden durch Klassen beschrieben.*

Definition:
Funktionale Programmierung *basiert auf seiteneffektfreien Funktionen. Funktionen ordnen den Objekten des Definitionsbereichs neue Objekte des Wertebereichs zu. Dadurch, dass Funktionen erstklassige Objekte sind, lassen sich auch höhere Funktionen definieren.*

Der große Unterschied zwischen beiden Sichtweisen besteht darin, dass Objektorientierung grundsätzlich von *veränderlichen Objekten* ausgeht. Funktionale Programmierung basiert demgegenüber auf *unveränderlichen Objekten*.

Daraus folgt, dass veränderliche Objekte nicht funktional sind. Es stellt sich aber dann immer noch die Frage, ob die Betonung der Unveränderlichkeit von Objekten Sinn macht und welche Rolle dabei die funktionale Programmierung spielt. Diese Frage soll im Folgenden etwas beleuchtet werden.

6.1 Zustandslose Objekte

Veränderliche Objekte haben in der Tat einige Probleme:

- Es ist nicht unproblematisch, Referenzen veränderlicher Objekte nach „außen“ weiterzugeben. Man weiß nie, was damit geschieht.
- Bei nebenläufigen Programmen kann es zu gravierenden Fehlern kommen, wenn mehrere Threads gleichzeitig auf gemeinsame veränderliche Objekte zugreifen.

- Da das Verständnis eines Programms von dem jeweiligen Objektzustand abhängt, wird die Verständlichkeit durch Veränderung erschwert. Es wird daher auch angeraten, die möglichen Veränderungen durch Klasseninvarianten einzuschränken.

Unveränderliche Objekte haben diese Nachteile nicht. Zwar macht es in der Objektorientierung keinen Sinn, nur auf Unveränderlichkeit zu bauen. Oft werden aber veränderliche Objekte auch da gedankenlos eingeführt, wo es sinnvoller wäre, funktional vorzugehen.

Als Beispiel soll hier eine Klasse `Bruch` dienen. Eine erste Version könnte so aussehen:

```
package mutable

class Bruch(z: Int, n: Int) extends Ordered[Bruch] {
  // Invariante zaehler und nenner sind gekuerzt
  private var zaehler = z
  private var nenner = n
  kuerzen()

  private def kuerzen() {
    require(nenner != 0)
    ...
  }

  def getZaehler = zaehler
  def getNenner = nenner

  def add(b: Bruch) =
    zaehler = zaehler * b.nenner + b.zaehler * nenner
    nenner = nenner * b.nenner
    kuerzen()
  }
}  ...
```

Die Klasse ist hier nicht ausformuliert. Die Programmierung sollte kein großes Problem sein!

Beachten Sie, dass die Klasse schon relativ „ordentlich“ programmiert ist. Es wird nämlich darauf geachtet, dass die Instanzvariablen gekapselt sind und dass bei allen Veränderungen die Klasseninvariante (gekürzt) eingehalten wird.

Trotzdem kann es bei dieser Klasse zu unerwarteten Problemen kommen wie das folgende Beispiel zeigt:

```
def methode1(b: mutable.Bruch) = {
   b.add(new mutable.Bruch(1, 2)
   val x = b.getZaehler
   ...
}

def methode2() {
   val b = new mutable.Bruch(4,7)
   val c = methode1(b)
   // welchen Wert hat b ?
   ...
}
```

Wenn wir in dem Beispiel `methode2` aufrufen, definieren wir dort eine unveränderliche Variable `b` als Bruch 4/7. Die Methode `methode1` soll für einen übergebenen Bruch ein Ergebnis berechnen. Wie Sie sehen, verwendet sie dabei ganz sorglos das übergebene Bruchobjekt. Und dabei zerstört sie den ursprünglichen Bruch. Wir haben hier prozedurale Programmierung in ihrer schlimmsten Form!

In einer Multithreading-Umgebung muss man auch darauf achten, dass ein solches Bruchobjekt nicht mehreren Threads bekannt ist. Sobald nämlich ein Thread eine Veränderung vornimmt, muss man garantieren, dass nicht gleichzeitig ein anderer Thread darauf zugreift. Sonst wäre es nämlich möglich, dass dieser das Objekt in einem Zustand antrifft, in dem die Klasseninvariante momentan nicht gilt. Diese Probleme und ihre Lösung werden später beim Thema Nebenläufigkeit besprochen.

Beim Rechnen mit Brüchen ist die Verwendung veränderlicher Objekte aber überhaupt nicht nötig und auch keinesfalls vorteilhaft.[1] Brüche stellen schließlich nichts anderes als besondere Zahlen dar. Beim „normalen" Rechnen mit Zahlen gehen Sie von einer funktionalen Sichtweise aus. Der Ausdruck $2 + 3$ ordnet den beiden Zahlen 2 und 3 die *neue* Zahl 5 zu (es wird *nicht* das „Objekt" 2 so verändert, dass es eine 5 ist).

Technisch gesehen, kann die geschickte Formulierung in Scala manchmal ein paar Probleme machen. Die Grundzüge sind jedoch ganz einfach.

```
package immutable

class Bruch private (z: Int, n: Int, g: Int) extends Ordered[
    Bruch] {
  require (n != 0)
  val zaehler = (if(n < 0) -z else z) / g
  val nenner = n.abs / g

  def this(z: Int) = this(z, 1, 1)
  def this(z: Int, n: Int) = this(z, n, gcd(z.abs, n.abs)

   @tailrec
   private def gcd(a: Int, b: Int): Int =
     if(b != 0) gcd(b, a % b) else a

   def +(b: Bruch) =
     new Bruch(zaehler*b.nenner + b.zaehler*nenner,
               nenner*b.nenner)

  ...
  override def toString =
    if (nenner == 1) zaehler.toString else zaehler + "/" +
       nenner

  override def equals(that: Any) = that match {
    case b:Bruch => this.zaehler == b.zaehler &&
                    this.nenner == b.nenner
    case _ => false
  }

  override def compare(b: Bruch) = (this - b).zaehler
}
```

[1] Vielleicht gibt es bei manchen Operationen geringfügige Laufzeitunterschiede.

Das einzige technische Problem besteht hier darin, dass die lokalen Variablen des primären Scala-Konstruktors immer als Instanzvariablen erscheinen. Das umgehe ich hier indem der primäre Konstruktor privat ist. Er erhält als dritten Parameter den größten gemeinsamen Teiler. Die öffentlichen sekundären Konstruktoren tätigen dann den richtigen Aufruf. Zugegeben, das ist etwas trickreich (ein Problem von Scala), hat aber mit dem eigentlichen Thema nichts zu tun.

Der eigentliche Vorteil liegt in der einfachen Verwendung und in der Unveränderlichkeit der Objekte. Beachten Sie, dass ich die Addition durch den +-Operator ausgedrückt habe. Schließlich verhält sich diese Methode genauso funktional wie die Addition von Zahlen.

Beachten Sie weiter, dass die Instanzvariablen `zaehler` und `nenner` jetzt öffentlich sind. Dies ist kein Verstoß gegen irgendwelche Stilregeln! Zunächst kann man damit den Objektzustand nicht zerstören, es sind ja schließlich unveränderliche Variablen. Zudem ist es auch so, dass Scala für den Zugriff auf Instanzvariablen ohnehin Zugriffsmethoden erzeugt. Es ist also immer möglich, später den direkten Zugriff auf den Wert einer Variablen in der Klasse `Bruch` selbst durch eine kompliziertere Funktion zu ersetzen, ohne dass dies außerhalb der Klasse bemerkt wird.

Zur Verdeutlichung sei nochmals das Anwendungsbeispiel angeführt.

```
import immutable.Bruch

def methode1(b: Bruch) = {
   val c = b + new Bruch(1, 2)
   val x = c.zaehler
   ...
}

def methode2() {
   val b = new Bruch(4,7)
   val c = methode1(b)
   // b = 4/7, egal was methode1 macht !!
   ...
}
```

Zuletzt soll noch eine Scala-Besonderheit angemerkt werden. In Scala kann man jeder Klasse ein gleichnamiges Objekt zuordnen (*assoziiertes Objekt*). In diesem Objekt lassen sich allgemeine Funktionen für die Klassenobjekte definieren (in Java wären das statische Funktionen). Eine Sonderrolle spielen dabei Funktionen namens `apply`. Diese fungieren als Fabrikmethoden. Sie erlauben eine vereinfachte Schreibweise für die Objekterzeugung und eine größere Flexibilität in der Erzeugung von Objekten.

Zum Beispiel können für häufig vorkommende Fälle fertige Objekte vorgehalten werden. Wenn man beispielsweise den Bruch 0 benötigt, wird kein neues Objekt erzeugt, sondern einfach eine Referenz auf das schon vorhandene 0-Objekt zurückgegeben. Diese Optimierung ist natürlich nur bei unveränderlichen Objekten möglich.

In Java ist die Verwendung unveränderlicher Objekte nicht unbekannt. Auch dort gibt es die Optimierung durch Wiederverwendung vorhandener Objekte. Beispiele sind die Klasse `String` (hier sorgt der Compiler dafür, dass gleich lautende Strings durch ein einziges gemeinsames Objekt gespeichert werden) und die

Verpackungsklassen für Zahlen (z.B. `Integer`). Die Verpackungsklassen haben zwar einen Konstruktor, es wird jedoch angeraten an seiner Stelle die Methode `valueOf` aufzurufen, die dann die Optimierung vornehmen kann. So wird der Aufruf `Integer.valueOf(0)` einfach eine Referenz auf das vorhandene Objekt `ZERO` zurückgeben.

```
package immutable

object Bruch { // assoziert zur Klasse Bruch
  val Zero = new Bruch(0)
  val One = new Bruch(1)
  val MinusOne = new Bruch(-1)

  def apply(zahl: Int) = zahl match {
    case 0 => Zero
    case 1 => One
    case -1 => MinusOne
    case _ => new Bruch(zahl)
  }

  def apply(zaehler: Int, nenner: Int) =
    if (nenner != 0 && zaehler % nenner == 0)
      apply(zaehler / nenner)
    else
      new Bruch(zaehler, nenner)
}

// und als Anwendung

def methode1(b: Bruch) = {
  val c = b + Bruch(1, 2) - Bruch.One
  val x = c.zaehler
  ...
}

def methode2() {
  val b = Bruch(4, 7)
  val c = methode1(b)
  ...
}
```

6.2 Unveränderliche Behälterklassen

Das Beispiel der Bruchklasse erscheint Ihnen vielleicht trivial. Warum sollte man das auch anders machen? Anders sieht das aber bei Klassen aus, die dazu gedacht sind, eine Ansammlung von Objekten zu speichern, nämlich bei den sogenannten Behälterklassen.

Zunächst einmal gibt es eine ganze Menge von Anwendungen, in denen es wirklich um unveränderliche Datenmenge geht. Wir hatten schon als einfachstes Beispiel die Klasse `String`. Stringobjekte sind ja auch nichts anderes als eine Folge von Buchstaben. Ähnlich kommen in Programmen oft andere Behälter vor. So kann ich in meinem Programm die Liste der Primzahlen bis 20 vorhalten:

```
val primesTo20 = List(2,3,5,7,11,13,17,19)
```

In diesem Fall haben wir eine Liste. Listen sind Datenbehälter in denen jedes Element eine Nummer hat. Alternativ hätten wir für unseren Zweck auch eine Menge definieren können. Mengen kennen keine Reihenfolge der Elemente, stellen aber sicher, dass kein Element doppelt vorkommt.

```
val primesTo20 = Set(2,3,5,7,11,13,17,19)
```

Egal ob Menge oder Liste, die Inhalte werden sich in diesem Fall nie ändern. Wir können aber trotzdem aus vorhandenen Mengen oder Listen neue Behälter erzeugen:

```
val primesTo20 = Set(2,3,5,7,11,13,17,19}
val primesTo10 = primesTo20 select(_ <= 10)
val primesTo30 = primesTo20 + Set(23,29)
```

Die Beispiele sehen sicher nicht schlecht aus. Sie erkennen aber unschwer, dass das auch Nachteile haben kann. Wenn ich z.B. die Menge der Primzahlen bis 20 ohne die 11 benötige, kann ich das ganz elegant so schreiben:

```
val primesTo20Without11 = primesTo20 - Set(11)
```

Allerdings haben wir hier einen erheblichen Kopieraufwand in der Größenordnung $O(N)$, wobei N für die Länge der Mengen steht. Dabei wollen wir doch nur ein einziges Element entfernen. In einer als veränderlicher `HashSet` organisierten Menge würde das in $O(1)$ also in konstanter Zeit geschehen.

Eine Antwort auf dieses Problem ist, dass sich Informatiker für einige funktionale Probleme optimierte Algorithmen haben einfallen lassen. Die andere Antwort ist, dass man in anderen Fällen am besten veränderliche Behälter verwendet! Dies kennen Sie auch schon von Java her, wo es neben der Klasse `String` auch die Klasse `StringBuilder` gibt, die Veränderungen effizient unterstützt. Ähnliches gilt auch in Scala. Da mein Thema aber die funktionale Programmierung ist, will ich hier nicht weiter darauf eingehen.

Merksatz:

Wenn es möglich ist, sollte man immer unveränderliche Objekte verwenden. Klassen für unveränderliche Objekte sollte die Unveränderlichkeit deutlich herausstellen. Durch unveränderliche Objekte wird vielen Programmierproblem von vornherein aus dem Weg gegangen.

6.3 Die Implementierung von Listenklassen

Wir haben im letzten Kapitel schon Listen kennengelernt. Listen sind in Scala (defaultmäßig) unveränderlich. Bei der Verwendung von Listen wird darauf geachtet, dass man möglichst nur Operationen verwendet, die effizient ausgeführt werden. Die Standardoperationen auf Listen `isEmpty`, `head` und `tail` werden alle in $O(1)$ ausgeführt.

Natürlich gibt es viele Möglichkeiten wie man Listen implementieren kann. Eine davon ist der Weg der Scala-Bibliothek. Sie hat ein paar Besonderheiten, die ihre

Verwendung erleichtern, für ihr Verständnis aber weitere Kenntnis von Scala voraussetzen. Eine andere lernen Sie vielleicht im Praktikum kennen. Im Kern sind die verschiedenen Wege aber gleich, sodass Sie die folgenden Darstellung für das Grundverständnis lesen können.

Das folgende Beispiel zeigt das Prinzip. Die Klassenhierarchie besteht aus drei Klassen. Die abstrakte Klasse `List` fungiert als gemeinsame Oberklasse. Sie nimmt gleichzeitig den Großteil der Listenfunktionen auf. Nur die drei elementarsten Funktionen `head`, `tail` und `isEmpty` sind bloß als abstrakte Funktionen deklariert.

Das Objekt `Nil` repräsentiert die leere Liste. Die Methode `isEmpty` gibt erwartungsgemäß `true` zurück. Leere Listen haben weder ein erstes, noch restliche Listenelemente. Die für `Nil` nicht sinnvoll definierbaren Methoden `head` und `tail` werfen eine Ausnahme.

Ignorieren Sie im Augenblick die etwas komplizierteren Ausdrücke für die Typparameter. Sie machen den Compiler glücklich (und den Programmierer manchmal unglücklich). Wichtiger ist die grundsätzliche Funktionsweise.

```
package myDefs

sealed abstract class List[+T] {
  def head: T
  def tail: List[T]
  def isEmpty: Boolean

  def length: Int =
    if (isEmpty) 0 else 1 + tail.length

  def ::[U >:T](x: U): List[U] = new Node(x, this)

  // foreach erlaubt die For-Each-Schleife
  def foreach(action: T => Unit): Unit = this match {
      case Nil       =>
      case Node(h,t) =>
        action(h)
        t.foreach(action)
  }
}

case object Nil extends List[Nothing] {
  override def isEmpty = true
  override def head: Nothing =
    throw new NoSuchElementException
  override def tail: List[Nothing] =
    throw new NoSuchElementException
}

case class Node[T] (value: T, next: List[T]) extends List[T]
    {
  override def head = value
  override def tail = next
  override def isEmpty = false
}
```

Die zentrale Klasse ist die Klasse `Node`.[2] Diese Klasse hat zwei Instanzvariablen

[2] In dem Scala-System heißt diese Klasse in Wirklichkeit `::`.

nämlich `value` und `next`. Diese Namen sind eigentlich in dem Zusammenhang etwas ungebräuchlich. Ich habe sie gewählt, um die Ähnlichkeit zur Implementierung von verketteten Listen in Java zu betonen. Mit diesen Klassen können wir Listen aufbauen und verwenden.

Die Definition der Funktion `foreach` zeigt einmal, wie dadurch die Anwendung von `for` auf unsere Liste implementiert wird. Außerdem weiche ich hier davon ab, die unterschiedlichen Methoden für `Nil` und für `Node` durch späte Bindung auszuwählen. Statt dessen wird hier diese Auswahl durch Pattern-matching vorgenommen.

Die Anwendung der Klasse sieht dann fast so, wie die der Bibliotheksklasse `List` aus.

```
import myDefs._ // verwende meine Definitionen

val list123 = 1::2::3::Nil
println(liste123.length)
for (x <- liste123) println(x)
```

Da wir in unserer Klasse über die grundlegenden Listenoperationen verfügen, lassen sich grundsätzlich alle höheren Operationen einfach definieren. Die vereinfachte Darstellung hat ein paar Einschränkungen. Wir haben hier keine Fabrikfunktion für die Definition von Listen und das Pattern-Matching sieht (wegen dem anderen Klassennamen) etwas anders aus. Das sind aber syntaktische Feinheiten, die mit dem generellen Thema nichts zu tun haben.

Es bleibt noch eine letzte Anmerkung zur Klasse `Nil`. `Nil` steht für die leere Liste. In Java hatten wir zur Verdeutlichung des Listenendes einfach eine Null-Referenz verwendet. Das erfüllt den Zweck und, da die Implementierung der verketteten Liste ohnehin verborgen ist, hat das auch keine besonderen Nachteile.

Die verbreitete Verwendung von `null` gilt aber auch in Java als problematisch. Das Problem ist, dass `null` kein Objekt ist. Wir dürfen nichts damit machen. Dies führt dazu, dass wir immer wieder damit zu kämpfen haben, dass wir an besondere Regeln für `null` zu denken haben.

Die Alternative zu `null` ist in Java, so wie hier, die Einführung besonderer Objekte für leere Datenstrukturen. In Java gibt es in der Bibliothek ein vordefiniertes leeres Listenobjekt (`java.util.Collections.emptyList()`). Eine verbreitete Stilregel bevorzugt mit der gleichen Begründung auch leere Strings und Arrays der Länge 0 vor Null-Referenzen.

Eine weitere Lösung wird im nächsten Abschnitt besprochen.

6.4 Die Scala-Schnittstelle `Option`

Prozedurale Programmierung kennt unterschiedliche Wege, mit nicht verfügbaren Resultaten einer Funktion umgegangen werden kann:

- Die Rückgabe besteht aus einem Fehlercode. Das widerspricht aber dem funktionalen Charakter. Ein Beispiel ist die C-Funktion `scanf`.
- Die Rückgabe besteht aus einem normalerweise unmöglichen Wert. So verfährt z.B. die Java-Methode `HashMap.get`.

- Es wird eine Ausnahme geworfen. Dies gilt in Java (und auch in Scala) für viele Methoden z.B. `ArrayList.get(i)`, wenn die Liste keine i-tes Element enthält. Ausnahmen sind im Kern imperativ: Sie springen zu einem anderen Befehl.

Scala nutzt in vielen Fällen das Java-Erbe. Es enthält aber auch Mechanismen, die aus der funktionalen Richtung kommen. Eine davon ist `Option`. Ein wesentlicher Vorteil von `Option` besteht darin, dass bereits im Datentyp deutlich wird, dass nicht immer ein Ergebnis vorliegt.

Das folgende Anwendungsbeispiel zeigt den Zusammenhang. Zunächst eine Skizze der Klassenhierarchie von `Option` (Sie erinnern sich? Eine `case class` ist eine vereinfachte Definition für eine Klasse, die Pattern-Matching unterstützt. Das Gleiche gilt für das Objekt `None`):

```
  sealed abstract class Option[+T] {
     def get: T
     def isEmpty: Boolean
     ...
  }

  case class Some[+T](x: T) extends Option[T] {
     def get = x
     def isEmpty = false
     ...
}

  case object None extends Option[Nothing] {
     def get = throw new NoSuchElementException
     def isEmpty = true
     ...
  }
```

Es ist nicht sinnvoll, hier alle Funktionen von `Option` anzugeben. Wie wir unten sehen werden sind Optionen nämlich Behälter. Sie enthalten entweder 0 oder 1 Element.

Suchfunktionen sind passende Beispiele für Funktionen, die nicht immer ein Ergebnis liefern. Als konstruiertes Beispiel können wir eine Suchfunktion nehmen.

In Java sah diese Funktion etwas wie folgt aus:

```
public static [T] int findIndex(T[] array, T x) {
  int i = 0;
  while (i < a.length && !x.equals(a[i]) i++;
  if (i == a.length)
      return -1;
  else
      return i;
}
```

Die folgende prozedurale Implementierung ist nicht viel anderes, außer, dass wir dieses Mal auch vom Ergebnistyp her deutlich machen, dass das gesuchte Element nicht immer gefunden wird.[3] Dadurch werden wir darauf gestoßen uns später auch zu vergewissern, ob wir das gesuchte Objekt gefunden haben.

[3] das Problem mit einer möglichen `NullPointerException` ignorieren wir mal.

```
def findIndex{T](a: Array[T], x: T): Option[T] = {
  var i = 0
  while (i < a.length && a(i) != x) i += 1
  if (i == a.length)
    None
  else
    Some(i)
}
```

Es gibt viele Möglichkeiten wie man dann das Ergebnis abfragt. Ein einfacher Weg besteht im Pattern-Matching:

```
val array = Array(17,29,...)
val ergebnis: Option[Int] = findIndex(array, 105)
  ...
ergebnis match {
  case None    => println("nicht gefunden")
  case Some(x) => println("gefunden bei " + x)
}
```

Sie werden einwenden, dass es bei der Weiterverarbeitung der Ergebnisse lästig sein kann, ständig die Resultate „auszupacken“. Die Scala-Bibliothek bietet bei Suchfunktionen auch zwei Varianten an. In einem Fall ist man sich des Ergebnisses sicher, im andern nicht. Man wählt einfach die passende Form.

Es gibt aber auch die Möglichkeit, mittels for-Ausdrücken mehrere optionale Resultate zu verknüpfen und zu einem Ergebnis zusammenzufassen:

```
val ergebnis: Option[Int] = for {
    index1 = findIndex(array1, 105)
    index2 = findIndex(array2, 200)
    if array3(index2) > 0
} yield funktion(array4(index1))
```

Das Ergebnis ist zwar wieder optional. Wir brauchen uns aber nicht über die Organisation der Zwischenschritte zu kümmern.

In Java würde dies vielleicht so aussehen:

```
int ergebnis = -1;
int index1 = findIndex(array1, 105);
if (index1 >= 0 {
    index2 = findIndex(array2, 200);
    if (index2 >= 0) {
        if(array3(index2) > 0)
            ergebnis = funktion(array4(index1));
    }
}
if (ergebnis == -1) ...
```

Dies ist nicht einmal soviel länger, als es unverständlicher ist. Natürlich dürfen wir in beiden Fällen nicht vergessen, am Ende zu fragen, ob die Variable `ergebnis` einen berechneten Wert hat. In Scala ist dies in der Typangabe erkennbar (und durch den Compiler überprüfbar), in Java muss der Programmierer daran denken.

Die „Magie“ des for-Ausdrucks wird mit durch die Anwendung von Funktionen höherer Ordnung verständlich. Dies wird im Abschnitt über Monaden besprochen. Dabei wird eine Technik besprochen, die für alle ähnlich strukturierten Klassen gilt: für Listen und Arrays wie für Optionen.

Es soll nicht unerwähnt bleiben, dass sich auch in Java Techniken durchsetzen, den optionalen Charakter von Ergebnissen deutlich zu machen. Dazu gehört, dass man Variable, die den Wert `null` annehmen können, entsprechend kennzeichnet:

```
@Nullable
Person p = map.get(partner);
```

6.5 Monoids

Listenklassen als wichtigste Behälterklasse haben in Scala den Typ `List[+A]`.[4] Für die folgende Darstellung ist es aber unwichtig, dass es sich um eine Liste handelt. Die Darstellung bezieht sich vielmehr auf fast jeden beliebigen Behältertyp (z.B. auch auf `Array`). Wichtiger ist es, dass der Elementtyp `A` über ein neutrales Element und eine zweistellige Verknüpfung verfügt.

Definition:

> *In der abstrakten Algebra bezeichnet ein **Monoid** eine Menge mit einer assoziativen Verknüpfung und einem neutralen Element. Ein Beispiel ist die Menge der ganzen Zahlen mit Addition und 0. Ein anderes Beispiel sind die reellen Zahlen mit Multiplikation und 1. Als Beispiel aus dem Bereich von Programmiersprachen können wir Strings mit der Konkatenierung von Zeichenketten und dem leeren String als neutralem Element nehmen.*

Die typische Anwendung des Konzepts der Monoids ist in Skala durch die Funktionen `foldLeft` und `foldRight` gegeben:

```
class M[A] {
    def foldLeft[B](neutral: B)(f: (B, A) => B): M[B]
    def foldRight[B](neutral: B)(f: (A, B) => B): M[B]
}
```

Für `M` können Sie hier eine der Behälterklassen einsetzen. Die Definition von `foldLeft` ist eine Verallgemeinerung der mathematischen Definition. Es ist nämlich auch ein anderer Ergebnistyp als `A` erlaubt.

Beispiel:

```
val a = Array(1,2,3,4)
val summeVonA = a.foldLeft(0)((x,y) => x + y)

val b = List("hello", " ", "world")
val helloWorld = b.foldleft("")(x,y) => x + y)
val anzahlChars = b.foldLeft(0)((x,y) => x + y.length)
```

[4] `+A` bezeichnet in Scala einen kovarianten Typparameter. Näheres dazu im zweiten Teil des Skripts.

Das letzte Beispiel zeigt, dass einer Menge von Strings ein anderer Datentyp, nämlich eine Zahl, zugeordnet werden kann. Bei der Angabe der Verknüpfung `f` kommt es darauf an, dass die beiden Operanden den richtigen Datentyp haben, der der Durchführung der Operation von links (beginnend mit dem neutralen Element) nach rechts entspricht.

Per Definition gilt bei Monoids das Assoziativgesetz. In den Datenstrukturen von praktischen Anwendungen muss dies nicht immer gelten. Zudem kann es auch bei echten Monoids gewünscht sein, die Operationen nicht von links nach rechts sondern umgekehrt von rechts nach links durchzuführen. Für diesen Fall gibt es die Operation `foldRight`.

Einige besonders häufig vorkommende Fälle sind in Scala bereits vereinfacht definiert, wie `sum` und `product` für die Summe oder das Produkt aller Zahlen einer Datenstruktur.

6.6 Monaden

Ebenso wie Monoids sind *Monads* (oder eingedeutscht *Monaden*) Grundstrukturen auf Behältern vom Typ `M[A]`.

Definition:

Eine Monade *ist eine Struktur* `M[A]` *mit wenigsten den Grundfunktionen* $unit : A \to M[A]$ *und* $bind : M[A] \to (A \to B) \to M[B]$. *In Scala ist die Funktion* unit *nicht vorhanden. Sie entspricht dem Konstruktor eines Behälters, der ja ein Objekt vom Typ* `A` *in einen Behälter vom Typ* `M[A]` *packt. Die Funktion* bind *heißt in Scala* `flatMap`. *In allen Scala-Klassen kommen zusätzlich noch die Funktionen* `filter` *und* `map` *hinzu.*

Die Scala-Definitionen sehen wie folgt aus:

```
abstract class M[A] {
  def flatMap[B](f: A => M[B]): M[B]
  def map[B](f: A => B): M[B]
  def filter(p: A => Boolean): M[A]
}
```

Die Tatsache, dass der Grundtyp `A` auf einen anderen Ergebnistyp abgebildet werden kann, stellt wieder eine Verallgemeinerung dar. Man kann noch weitere monadische Funktionen definieren; diese drei sind aber die wichtigsten. Dies kommt auch dadurch zum Ausdruck, dass der For-Ausdruck von Scala eine Abkürzung dafür bereitstellt. Das folgende Beispiel zeigt eine Verwendung aller drei Funktionen und ihre Darstellung als For-Ausdruck.

```
val ergebnis1 = liste1.flatMap(x=> liste2.filter(
    z => x*z > 0).map(y => math.sqrt(x*y)))

val ergebnis2 = for {
    x <- liste1                 // flatMap
    y <- liste2 if x*y > 0      // map und filter
    } yield math.sqrt(x * y)    // teil von map
```

Für das Verständnis des For-Ausdrucks muss man sich nur das Folgende merken. Wir nehmen an, die Funktionen sind in einer Klasse `M[A]` definiert (ein konkretes Beispiel wäre `List[A]`).

- Alle Elemente werden durch Verknüpfung und Schachtelung von monadischen Funktionen ausgedrückt.
- Bedingungen führen zu dem Aufruf von `filter`.
- Die innerste Schleife, zusammen mit dem Yield-Ausdruck, wird durch `map` ausgedrückt.
- Äußere Schleifen erfordern anstelle von `map` das mächtigere `flatMap`.

Das folgende Beispiel gibt die Anschrift aller Telefonnummern von Personen einer Liste zurück, die in einem Telefonbuch enthalten sind und im Inland leben.

```
val personen = List("Hans", "Kurt", "Karin", "Lisa")
val kontakte = Map{
    "Hans" -> "01493334"
    "Karin" -> "+332334327"
    "Lisa" -> "+0221654312"
    ...
}
val wohnort = Map{
    "Karin" -> "Koeln"
    ...
}
val laender = Map {
    "Koeln" -> "Deutschland"
    ...
}

val anzurufen = for {
       person <- liste              // flatMap
       ort <- wohnort.get(person)   // flatMap
       land <- laender.get(ort)     // flatMap
       if land == "Deutschland"     // filter
       nr <- kontakte.get(person)   // map
} yield (person, nr)                // map
```

Für den Scala-Compiler ist das jetzt nichts Besonderes. So wie man `List[A]` als ein `M[A]` lesen kann, kann man dies auch für `Option[A]`.

Der äußere Behälter (`personen`) ist eine `List[String]`, das Ergebnis von `get` ist eine `Option[String]`, also entweder eine `Some(`*telefonNr*`)` oder `None`, wenn die Person nicht in den Kontakten gefunden wird. Der Vorteil der Scala-Darstellung wird deutlich, wenn man diese Anwendung in Java programmiert.

```
List<String> personen =
    Collections.asList("Hand", "Kurt", "Karin", "Lisa");
Map<String, String> kontakte = new HashMap<String,String>();
kontake.put("Hans", "+491493334");
kontakte.put("Karin", "+332334327");
...
Map<String, String> wohnort = ...
Map<String, String> laender = ...
```

```
List<String> anzurufen = new LinkedList<String>();
for (String person : personen) {
    String ort = wohnort.get(person);
    if (ort == null)
        continue;
    String land = laender.get(ort);
    if (land == null || !land.equals("Deutschland"))
        continue;
    String nr = kontakte.get(person);
    if (nr != null)
        anzurufen.add(nr);
}
```

Beachten Sie, dass Sie in der Java-Version daran denken müssen, bei jeder Operation, die möglicherweise eine `null` zurückgeben kann, zu prüfen, ob man tatsächlich eine gültige Objektreferenz erhalten hat. Andernfalls riskieren Sie, dass eine `NullPointerException` geworfen wird. Wenn Sie das Beispiel mit realen Anwendungen vergleichen, werden Sie schnell feststellen, dass es noch ziemlich harmlos ist. Auch wenn man alles korrekt macht, entsteht bereits das unschöne Ergebnis, dass der endgültige Code die eigentliche Programmlogik hinter Fehlerabfragen verschwinden lässt.

Dieser lästige und fehleranfällige Umgang mit der `null` hat den „Erfinder“ der `null`, nämlich den Informatikpionier *Anthony Hoare* veranlasst, von dem größten Fehler seines Berufslebens und dem „billion dollar bug“ zu sprechen. Vergleichen Sie nur, dass zu diesem Zweck C# den Referenzierungsoperator `.?` eingeführt hat, oder das die IOS-Variante der Programmiersprache Objective-C bei einem Methodenaufruf mit einer `nil`-Referenz anstelle eines Fehlers einfach `nil` zurückgibt (`nil` entspricht der `null`).

Monaden (in .Net gibt es sie auch) erlauben es, eine Kette von Operationen zu definieren, die sich immer richtig verhält. Nur am Ende muss man evtl. entsprechend dem Resultat entscheiden, was zu tun ist.

In funktionalen Sprachen werden monadische Verknüpfungen von Operationen auch eingesetzt um andere komplexe Operationen zu verbergen. Zum Beispiel kann man auf diese Art Parser schreiben (Syntaxanalyse), die sozusagen unter der Hand alle erforderliche Information weitergeben, einschließlich der Steuerung der Fehlerbehandlung. An der Oberfläche des Programms erscheint nur die eigentliche Verknüpfungslogik, die im Fall des Parses einfach den Regeln der Grammatik entspricht.

Ein anderes Beispiel mit möglicherweise großer Bedeutung für die Zukunft ist die Formulierung datenparalleler Anwendungen durch Monaden. Hier werden dann die technischen Details der Parallelisierung verborgen.

6.7 Implementierung von Monaden

Abschließend sollen als Beispiel `flatMap` und `map` für Listen und für Optionen definiert werden. Die anderen Funktionen ergeben sich ähnlich einfach.

```
abstract class List[+A] {
    ...
```

```
    def flatMap[B](f: A => List[B]): List[B] =
      if (this.isEmpty)
          Nil
      else
          f(this.head) ::: this.tail.flatMap(f)

    def map[B](f: A => B): List[B] =
      if (this.isEmpty)
          Nil
      else
          f(this.head) :: this.tail.map(f)


}

abstract class Option[+A] {
    ...
    def flatMap[B](f: A => Option[B]): Option[B] =
      if (this.isEmpty
        None
      else
        f(this.get)

    def map[B](f: A => B): Option[B] =
      if (this.isEmpty
        None
      else
        Some(f(this.get))

}
```

Ich habe hier eine Form der Implementierung gewählt, die einerseits die Ähnlichkeit unterstreicht und auch grundsätzlich so für die Bibliotheksklassen wie für die hier besprochenen Beispiele gültig ist.

Zur Übung können Sie auch versuchen `flatMap` anzugeben. Hat man `flatMap`, kann man damit auch `map` und `filter` definieren.

Literaturverzeichnis

[AS96] Abelson, Sussman, *Structure and Interpretation of Computer Programs*
MIT Press 1996
Dies ist das grundlegende Buch über Programmierparadigmen. Dabei wird insbesondere auf die funktionale Programmierung eingegangen. Als Programmiersprache dient Scheme. Das Buch ist auch in Deutsch erhältlich.

[BACK] J. Backus, *Can Programming be Liberated from the Von Neuman Style?*
ACM, Rede zur Verleihung des Turing Awards
In dieser Rede stellt Backus, einer der Informatikpioniere, die streng funktionale Sprache FP vor.

[CHBJ14] P. Chiusano, R. Bjarnason, *Functional Programming in Scala*
Manning, 2014
Sehr gute Einführung in die modernen Konzepte der funktionalen Programmierung am Beispiel von Scala.

[CM89] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*
Springer-Verlag, 12010
Dies ist *das* Standardwerk zu Prolog. Neben der Beschreibung der wichtigsten Spracheigenschaften gibt es auch eine Einführung in einen guten Programmierstil und in den Zusammenhang von Prolog und Logik. Wenn jemand vorhat, sich intensiver mit Prolog zu befassen, ist es unbedingt zu empfehlen.

[MC62] J. McCarthy et al., *LISP 1,5 Programmer's Manual*
MIT Press 1962
Eine der ersten LISP-Veröffentlichungen.

[ORF09] M. Odersky, *The Scala Reference*
Draft, EPFL, 2009
Sehr formal und daher schwer zu lesende Sprachreferenz.

[ODY10] M. Odersky, *Scala by Example*
Draft, EPFL, 2010
Eine sehr gute Übersicht über das Programmieren in Scala. Momentan frei erhältlich!

[OSV08] Odersky, Spoon, Venners, *Programming in Scala*
Artima Press, 2011
Das ist momentan vielleicht die beste Referenz zu Scala.

[SCH89] U. Schöning, *Logik für Informatiker*
BI Wissenschaft 1989
Eine sehr gute und verständliche Einführung in Aussagenlogik, Prädikatenlogik und in Beweisverfahren.

[YA95] R. Yasdi, *Logik und Programmieren in Logik*
Prentice Hall 1995
In dem Buch wird parallel in Logik und in Prolog eingeführt.

Anhang A

Glossar

Äquivalenz: Zwei Formeln sind logisch äquivalent, wenn sie bei jeder Interpretation die gleichen Wahrheitswerte annehmen.

Anfrage: auch *Zielklausel*. Konjunktion von Literalen, die zu beweisen ist. Prolog stellt Anfragen durch negative Klauseln dar.

Atom: elementarste logische Aussage. In der Prädikatenlogik ist ein Atom durch einen Prädikatsnamen und eine Anzahl von Argumenttermen gegeben. In Prolog kommt der Begriff *Atom* auch mit der Bedeutung „nicht-numerische Konstante“ vor.

Aussage: Formel, die die Werte wahr oder falsch annehmen kann. Die Bedeutung einer Aussage ergibt sich aus ihrer *Interpretation*.

Backtracking: Das Backtracking stellt eine Variante der *Tiefensuche* dar. Dabei wird ein Ableitungsweg soweit verfolgt bis entweder das Ziel (die leere Klausel) hergeleitet wurde, oder bis keine weitere Ableitungen mehr möglich sind. Im letzten Fall wird dann der letzte Ableitungsschritt rückgängig gemacht und erneut eine andere Ableitung versucht.

Bedeutung: Die Bedeutung eines Logikprogramms, ist die Menge der ableitbaren Atome.

Beweis: Ableitung eines Satzes in einem *Kalkül*. In einem *negativen Testkalkül*, wie Prolog, besteht ein Beweis in der Ableitung der leeren Klausel.

Beweisbaum: Der Beweisbaum stellt einen *einzigen* Beweis graphisch dar. Die Knoten des Beweisbaums sind Literale, die Kanten stellen *Resolutionen* dar, mit denen die Literale aufgelöst werden.

Breitensuche: Vollständiges Suchverfahren, in dem der Suchbaum ebenenweise abgearbeitet wird.

call by name: Das übergebenen Argument wird nicht beim Aufruf, sondern bei seiner Verwendung (jedes mal neu) evaluiert. Der Funktionsparameter stellt damit eine Funktion zur Berechnung des Wertes dar.

call by value: Das Argument wird vor dem Aufruf evaluiert und nur der Ergebniswert wird in den Funktionsparameter kopiert.

closed world assumption: Annahme, dass alle relevanten Fakten eines Sachverhalts durch logische Formeln modelliert wurden, mit der Konsequenz, dass alles, was nicht explizit als wahr festgestellt wurde, als falsch gilt.

Closure: In der funktionalen Programmierung versteht darunter ein Funktionsobjekt, das auch die freien Parameter der Definitionsumgebung enthält.

Currying: Diese Technik wurde in den λ-Kalkül eingeführt um mehrparametrige Funktionen durch einparametrige Funktionen auszudrücken. Die Technik beruht darauf, dass in der funktionalen Programmierung Funktionen als Funktionsresultat auftreten können. In Scala wird die Technik häufig angewendet um Formulierungen zu finden, die der Benutzererwartung entsprechen (DSL).

Cut: Der Cut, ausgedrückt durch „!“, ist ein Metaprädikat mit dem die Suchstrategie des Prolog-Interpreters beeinflusst wird. Seine Wirkung besteht ausschließlich darin, dass er beim Backtracking weitere Ableitungsversuche für das Prädikat, in dem er enthalten ist, unterbindet. Die *logische* Bedeutung des Cut ist wahr.

cut-fail: Die cut-fail-Kombination wird verwendet um Verneinung in Prolog-Programmen auszudrücken. Eine andere Möglichkeit dazu ist `not`.

Deterministisches Programm: Ein deterministisches Logikprogramm ist ein Programm, bei dem der Suchbaum zu einer linearen Liste entartet ist. In einem leicht verallgemeinerten Sinn bezeichnet man oft auch Programme, die höchstens eine Lösung liefern, als deterministisch.

Einheitsklausel: (auch *unäre Klausel*) ist eine Klausel, die genau ein *Literal* enthält.

Endrekursion: Eine Funktion oder ein Prädikat heißt dann endrekursiv, wenn der rekursive Aufruf die letzte Aktion bei der Durchführung der Funktion oder des Prädikats darstellt. Bei der Endrekursion kann eine Optimierung durchgeführt werden, die die Rekursion praktisch in eine Iteration umwandelt. Dadurch wird der normalerweise mit der Rekursion verbundene Zeit- und Speicherverbrauch vermieden. In der Logikprogrammierung ist die Optimierung nur dann möglich, wenn gleichzeitig ein Backtracking ausgeschlossen ist (deterministisches Prädikat).

Erfüllbarkeit: Eine logische Formel ist erfüllbar, wenn es eine Interpretation gibt, bei der sie wahr ist.

Falsifizierbarkeit: Eine logische Formel ist falsifizierbar, wenn es eine Interpretation gibt, bei der sie falsch ist.

freie Variable: Eine Variable, die nicht in einer Funktion selbst definiert ist, sondern aus der Definitionsumgebung übernommen wird.

funktionales Programm: Ein funktionales Programm fasst ein Programm als eine Abbildung auf, bei der einer Liste von Eingabeelementen eine Liste von Ausgabeelementen zugeordnet ist. Dieser funktionale Zusammenhang wird durch die Komposition elementarer Funktionen beschrieben. Ähnlich wie ein Logikprogramm ist ein funktionales Programm zunächst eine deklarative Aussage. Allerdings gewinnt es bei der Ausführung auf einem Computer das übliche dynamische Verhalten, auf dem letztlich auch die Möglichkeit der Formulierung prozeduraler Elemente beruht.

Funktion höherer Ordnung: Eine Funktion höherer Ordnung ist eine Funktion, die Funktionen als Parameter enthält. Häufig definiert man mit Funktionen höherer Ordnung Operationen auf gesamten Datenstrukturen.

Funktionsliteral: Eine Funktion, die ohne Namensangabe an Ort und Stelle definiet wird (*Lambda-Ausdruck*).

gebundene Variable: Eine Variable, die in der Funktion selbst (lokal) definiert ist.

generate and test: ist eine Strategie zur Lösung kombinatorischer Probleme. Bei diesem Ansatz benutzt man einerseits ein nichtdeterministischen Prädikat, das eine Vielzahl potentieller Lösungen *generiert*, und andererseits ein deterministisches Prädikat, das die Zulässigkeit des Lösungsvorschlags *testet*.

grüner Cut: ist ein *Cut* der die Bedeutung eines Programms nicht verändert.

Grund-: Der Vorsatz Grund- vor Begriffen, wie Instanz, Atom, usw. drückt aus, dass die entsprechende Formel keine Variablen enthält.

Hornklausel: Eine Hornklausel ist eine *Klausel*, die höchstens ein positives *Literal* enthält. In Prolog stellen negative Klauseln Anfragen dar, positive Einheitsklauseln stehen für Fakten und Hornklauseln die neben einem positiven Literal ein oder mehrere negative Literale enthalten bilden Regeln. Das positive Literal einer Hornklausel heißt auch *Kopf* der Klausel; die negativen Literale bilden den *Körper* der Klausel.

imperative Programmierung; Ein Programmierstil, der ein Programm als eine Folge von Befehlen betrachtet.

Instanz: Aus einer Formel, die Variablen enthält, können durch die *Substitution* weitere gültige Formeln – die Instanzen der Formel – abgeleitet werden.

Iteration: Die Durchführung einer Wiederholung durch eine Programmschleife heißt normalerweise Iteration. Bei der funktionalen und der Logikprogrammierung wird Wiederholung durch Rekursion ausgedrückt. Compiler und Interpreter sehen jedoch vor, Rekursion, wenn möglich, intern in Iteration umzuwandeln. Diese Optimierung ist grundsätzlich bei (deterministischen) *endrekursiven* Programmen möglich. Endrekursive Funktionen und Prädikate werden daher oft auch als iterativ bezeichnet.

Kalkül: Ein Kalkül ist ein formales System, bestehend aus einer Menge von Axiomen und einer Menge von Schlussregeln. Ein logischer Kalkül, der auf allgemeingültigen Axiomen beruht, heißt *positiver* Kalkül. Daneben gibt es auf unerfüllbaren Axiomen beruhende *negative* Kalküle. Ein Kalkül, der bei den Ableitungen von den Axiomen ausgeht, heißt *Deduktionskalkül*; ein umgekehrt vorgehender Kalkül heißt *Testkalkül*. Der auf der unerfüllbaren leeren Klausel beruhende *Resolutionskalkül* von Prolog ist ein negativer Testkalkül.

Klausel: Eine Klausel ist die abgekürzte Schreibweise für eine Disjunktion von Literalen.

Klauselnormalform: In der Klauselnormalform wird eine logische Formel durch eine Konjunktion von *Klauseln* ausgedrückt. Alle Variablen sind *universell quantifiziert*.

Korrektheit: Ein Logikprogramm ist korrekt, wenn die Menge der ableitbaren Aussagen $\mathcal{M}(\mathcal{P})$ in der *intendierten Bedeutung* $\mathcal{M}$ enthalten ist.

Lambda-Ausdruck: Historisch motivierte Bezeichnung für Funktionsliterale. In vielen Programmiersprachen wird die Definition von Funktionsliteralen durch das Schlüsselwort `lambda` oder eine davon abgeleitete Form eingeleitet.

Literal: Ein Literal ist ein *Atom*, das unter Umständen negiert sein kann. Ein negiertes Literal heiß *negatives Literal*; die anderen Literale heißen *positive Literale*.

Logikprogramm: Ein Logikprogramm ist eine Menge von Regeln. Die Art und Weise wie diese Regeln abgearbeitet werden, ist nicht Bestandteil des Programms. Prolog stellt die bekannteste Annäherung an die Idee eines Logikprogramms dar.

Monad: Genaugenommen sind Monaden Gegenstand der abstrakten Algebra. In der funktionalen Programmierung bezeichnet man damit eine Verallgemeinerung des Konzepts der Behälterklassen zusammen mit den darauf wirkenden Funktionen höherer Ordnung. Beispiele für solche Behälter sind in Scala `List[A]`, `Stream[A]` und `Option[A]`. In Scala ist das mondadische Verhalten durch die Funktionen `map`, `flatMap` und `filter` ausgedrückt. Die *for-comprehension* ist eine vereinfachte Syntax für die Anwendung dieser Funktionen.

Monoid: Monoids bezeichnen in der Mathematik eine Menge von Elementen mit einer assoziativen Verknüpfung und einem neutralen Element. In der funktionalen Programmierung wird dieses Konzept etwas erweitert. Die Verknüpfung kann auch auf einen anderen Datentyp abbilden und die Verknüpfung muss nicht assoziativ sein. Die wichtigsten Funktionen sind in Scala `foldLeft` und `foldRight`.

partiell angewendete Funktion: Bei einer partiell angewendeten Funktion wird zunächst nur ein Teil der Parameter ausgewertet. Es ergibt sich dabei eine Funktion der restlichen Parameter.

partiell definierte Funktion: Eine partiell definierte Funktion ist nicht für alle Elemente des Definitionsbereichs definiert.

Prädikat: Die Menge von Hornklauseln mit einem Kopfliteral gleichem Namen und gleicher Stelligkeit. Prädikate stellen in Logikprogrammen komplexe Sachverhalte oder Regen dar. Sie bilden in Logikprogrammiersprachen das Ausdrucksmittel für Funktionen und Prozeduren.

referentielle Transparenz: Der Aufruf einer referentiell transparenten Funktion kann an jeder Stelle durch ihr Ergebnis ersetzt werden (und umgekehrt). Die Funktion hat keinen von außen erkennbaren Seiteneffekt, obwohl sie evtl. selbst imperativ programmiert ist.

reines Lisp: ein Lisp-Programm, das ausschließlich funktionale Elemente enthält. Reine Lisp-Programme haben keine Seiteneffekte.

reines Prolog: ein Prolog-Programm, das ausschließlich auf der Prädikatenlogik erster Stufe beruht. Es enthält insbesondere keine Veränderung der Datenbasis durch `assert`, keine Seiteneffekte, keine Metaprädikate und keine eingebaute Arithmetik.

Rekursion: Eine Funktion, ein Prädikat oder eine Datenstruktur, die innerhalb ihrer Beschreibung auf sich selbst Bezug nehmen heißen rekursiv. Beachten Sie, dass mit der Rekursion in erster Linie eine *Aussage* verbunde ist. In funktionalen und in logikorientierten Programmiersprachen stellt Rekursion das wichtigste Mittel zur Formulierung von Wiederholungen dar. Die Auswertung einer Rekursion kann im Computer durch einen iterativen oder durch einen rekursiven Ablauf erfolgen.

Resolution: Die (binäre) Resolution ist eine Schlussregel, bei der aus zwei Klauseln eine neue gebildet wird, vorausgesetzt die beiden Klauseln enthalten zwei unifizierbare Literale unterschiedlichen Vorzeichens. Die entstehende Klausel heißt *Resolvente*. Bis auf die Unifikationsliterale enthält die Resolvente alle Literale der beiden Ausgangsklauseln.

Resolutionskalkül: Der Resolutionskalkül ist ein negativer Testkalkül, der die binäre Resolution als einzige Schlussregel enthält. Wegen der einfachen Struktur bildet er die Grundlage der Logikprogrammierung.

roter Cut: Ein roter Cut ist ein *Cut* der die Bedeutung eines Programms verändert. Enthält ein Programm einen roten Cut, so differiert die logische Bedeutung von seiner prozeduralen Bedeutung (im allgemeinen sind diese Programme dann logisch falsch).

Scala: Scala ist eine objektorientierte Programmiersprache, die weitgehend auch die funktionale Programmierung unterstützt. Die verbreitetste Implementierung ist in die Java-Umgebung eingebettet. Der Compiler erzeugt in diesem Fall Java-Bytecode und es können Java Klassen verwendet werden.

Schlussregel: Formale Vorschrift mit der in einem *Kalkül* aus einer Menge von Formeln andere Formeln abgeleitet werden können.

Seiteneffekt: Ein Effekt, der sich nicht aus der funktionalen oder der logischen Bedeutung eines Programms ergibt. Seiteneffekte beruhen auf der prozeduralen Ausführung des Programms. Seiteneffekte machen Programme schwer verständlich, sind jedoch oft auch unvermeidbar (Ein-/Ausgabe).

Semantik: Die Semantik bezeichnet die Bedeutung einer Formel. Diese Bedeutung entsteht durch eine *Interpretation* in der den einzelnen Formelzeichen (reale) Sachverhalte zugeordnet werden. In einem Logikprogramm versteht man unter der Semantik auch die Menge der Konsequenzen dieses Programms.

singuläres Objekt: Ein singuläres Objekt ist das einzige Objekt seiner Klasse. Häufig ist damit auch ein global sichtbarer Name verbunden. In Scala können singuläre Objekte durch `object` definiert und erzeugt werden. In Java durch `enum` oder durch besondere Erzeugungsmuster.

Stelligkeit: Die Stelligkeit (auch *arity*) eines *Atoms* oder eines *terms* ist die Anzahl seiner Argumente.

Substitution: Durch eine Substitution wird einer *Variablen* ein Ausdruck zugeordnet. In Prolog ist dies eine der grundlegenden *Schlussregeln*, die darauf beruht, dass in Prolog alle Variablen universell quantifiziert sind.

Suchbaum: Der Suchbaum stellt *alle möglichen* Ableitungen einer Anfrage dar. Die Wurzel des Suchbaums ist die Zielanfrage, die Knoten stellen die jeweiligen *Resolventen* in der Ableitung der. Die Kanten entsprechen möglichen *Resolutionen*. Häufig werden die Kanten mit denen bei der Resolution gefundenen *Substitutionen* dekoriert.

Struktur: In Prolog ein Begriff, der die syntaktisch gleich aussehende Struktur von Atomen und Termen beschreibt.

tail recursion: siehe *Endrekursion*.

Term: Ein Term ist in der Logik ein funktionaler Ausdruck. Er besteht aus einem symbolischen Namen (*Funktor*) und einer festen Anzahl von Parametern. Die Parameter eines Terms sind wiederum Terme. Die Parameterzahl eines Terms heißt *Stelligkeit*. Ein Term mit der Stelligkeit 0 ist eine Konstante.

Tiefensuche: Eines der wichtigsten Suchverfahren in *Graphen*. Bei der Tiefensuche wird bei jedem Knoten eine beliebige Kante weiterverfolgt bis das Ziel gefunden ist oder bis eine „Sackgasse“ erreicht ist und durch *Backtracking* andere Verzweigungen versucht werden müssen. Ein Problem bei der Tiefensuche stellt die Vermeidung von *Kreisen* im Suchablauf dar. Die Tiefensuche kann sehr speichereffizient und häufig auch laufzeiteffizient implementiert werden. Sie stellt allerdings kein *vollständiges* Suchverfahren dar.

Tautologie: logische Formel, die bei jeder Interpretation wahr ist.

Unifikation: Mit dem Unifikationsalgorithmus wird für zwei Atome (oder Terme) eine Substitution gesucht (*allgemeinster Unifikator*) mit der die beiden Atome (oder Terme) eine gemeinsame Instanz erhalten.

Variable: In der funktionalen Programmieren tauchen Variable nur als symbolische Namen für Funktionsparameter auf. In der Logikprogrammierung kann für eine Variable eine beliebige Konstante stehen, d.h. die Variablen sind universell quantifiziert. Bei der Beweissuche durch Backtracking lässt sich feststellen, ob an einem gegebenen Punkt des Programms bereits eine Festlegung des Variablenwerts durch eine Substitution stattgefunden hat oder nicht. Eine Variable, für die es noch keine Substitution gab (oder nur eine Substitution mit einer freien Variablen), heißt *frei*. Im andern Fall heißt die Variable *gebunden*.

Vollständigkeit: Ein Logikprogramm ist vollständig, wenn die intendierte Bedeutung $\mathcal{M}$ in der Bedeutung des Programms $\mathcal{M}(\mathcal{P})$ enthalten ist.

Widerspruch: logische Formel, die bei jeder Interpretation falsch ist.

Widerspruchsbeweis : ein *indirekter Beweis*, bei dem eine Aussage bewiesen wird, indem gezeigt wird, dass aus der Negation der Aussage ein Widerspruch abgeleitet werden kann.