Paradigmen der Programmierung I

Funktionale Programmierung und Logikprogrammierung

Prof. Dr. Erich Ehses

FH Köln
Abteilung Gummersbach

Wintersemester 2011/2012

©Prof. Dr. E. Ehses, 1996-2011

Inhaltsverzeichnis

1 Einführung **5**

2 Die Programmiersprache Prolog **9**

2.1 Ein Überblick über die Verwendung von Prolog 10

2.1.1 Eine Prolog-Programmdatei 10

2.1.2 Eine interaktive Prolog-Sitzung 12

2.2 Die Prolog-Syntax 14

2.2.1 Lexikalische Grundelemente 14

2.2.2 Die syntaktischen Strukturen von Prolog 17

2.3 Unifikation und Resolution 19

2.3.1 Die Unifikation 19

2.3.2 Die Resolution 21

2.3.3 Die prozedurale Interpretation von Prolog 25

2.4 Abweichungen von der Logik 26

2.4.1 Eingebaute Prädikate 26

2.4.2 Weitere eingebaute Prädikate 28

2.4.3 Negation und Cut 28

2.5 Die logische Grundlage von Prolog 29

2.5.1 Anforderungen 30

2.5.2 Logik und Prolog-Syntax 30

2.5.3 Das negative Resolutionskalkül 32

3 Logikprogrammierung in Prolog **35**

3.1 Grundregeln 35

3.2 Endrekursion 36

3.3 Listenverarbeitung 38

3.3.1 Prolog-Notation für Listen 38

3.4 Lösungssuche 41

3.4.1 Tiefensuche in Prolog 41

3.4.2 Lösungssuche durch systematisches Ausprobieren 42

3.4.3 Kombinatorische Suche 44

4 Überblick über Scala 47

4.1 Alles ist ein Objekt 47

4.2 Aufbau eines Scala-Programms 48

4.2.1 Pakete 48

4.2.2 Variablendeklarationen und Typparameter 49

4.2.3 Scala-Singleton-Objekte 50

4.2.4 Scala-Klassen und Konstruktoren 51

4.2.5 Methodendeklaration 52

4.2.6 Kontrollstrukturen 53

4.3 Wichtige Erweiterungen 54

4.3.1 Funktionsliterale 54

4.3.2 Die Match-Case Anweisung von Scala 56

5 Funktionale Programmierung 59

5.1 Das Paradigma der funktionalen Programmierung 59

5.1.1 Funktionen in der Mathematik 59

5.1.2 Grundelemente der funktionalen Programmierung 62

5.2 Funktionale Programmierung am Beispiel Scala 64

5.2.1 Funktionsdefinition und Funktionsanwendung 64

5.2.2 Closures und Currying 66

5.2.3 Rekursion 69

5.2.4 Funktionsobjekte 72

5.2.5 Partielle Funktion 73

5.2.6 Call by Name und Kontrollabstraktion 74

5.2.7 Funktionale Datenstrukturen 77

5.2.8 Funktionen höherer Ordnung 78

6 Funktionale Klassen 83

6.1 Zustandslose Objekte 83

6.2 Unveränderliche Behälterklassen 87

6.3 Die Implementierung von Listenklassen 88

A Glossar 93

©Prof. Dr. E. Ehses, 1996-2011

Kapitel 1

Einführung

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

unbekannter Prolog Programmierer

Den meisten Studenten der Allgemeinen Informatik geht es darum, möglichst schnell und gut in die gängigen Methoden der Objektorientierten Programmierung einzusteigen. Sie wünschen sich oft weiterführende Vertiefungen in Java.

Zum Teil wird das auch noch kommen. Es gibt aber gerade in der Informatik und in der Programmierung auch die Notwendigkeit, sich mit Konzepten auseinanderzusetzen, die im Augenblick eine geringe praktische Bedeutung haben. Diese Konzepte bilden häufig das Gemeinwissen von Informatikern und liegen oft auch dem Entwurf aktueller Programmiersprachen zugrunde. Ganz abgesehen davon, muss man in der Informatik immer damit rechnen, dass „vergessene" Ansätze plötzlich aktuell werden.

In der Lehrveranstaltung *Paradigmen der Programmierung* sollen alle die Ansätze zusammengefasst werden, die bei Algorithmen und Programmierung unter den Tisch fallen.

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.

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 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

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

prozeduralen Paradigmas dar.

Die Erfahrung mit Objektorientierung hat zu der Erkenntnis geführt, dass die Effizienz einer Programmiersprache nicht das ausschlaggebende Kriterium sein darf. Heute wird anderen Softwarequalitäten das nötige Gewicht eingeräumt.

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 bestimtme Sicht auf die Welt dar. Auch wenn es gleichmaßen für alles brauchbare Paradigma gibt, so sieht es innerhalb des Denksystemens eines Paradigmas immer so aus, als gebe es kein anderes.*

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 passiven Methoden der Objekte durch den 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 gesitzt 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.

©Prof. Dr. E. Ehses, 1996-2011

Die dargestellten Programmierstile lassen sich grob in zwei Bereiche einteilen, die aber nicht ohne Übergänge sind.

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

Programmierung von Nebenläufigkeit erfordert ein Abgehen von dem rein imperativen Denken, da sich nebenläufige Aktionen nicht exakt vorher bestimmen lassen. Objektorientierung ist ein Konzept, dass 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.

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.
- 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 Unterschiedung mit allen Programmierparadigmen verbinden. Es ist aber so, dass Systeme für höhere Programmierkonzepte (Objektorientierung, Logikprogrammierung und Funktionale Programmierung) von Anfang 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.

©Prof. Dr. E. Ehses, 1996-2011

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. Eine kurze Einführung in die Konzepte der formalen Logik ist daher unerlässlich. Dafür erhalten Sie aber dann auch einen Einblick, wie man per Programm mit Prädikatenlogik umgehen kann.

Bei der Diskussion der funktionalen Programmierung werde ich die Anwendbarkeit funktionaler Techniken betonen. Sie lassen sich oft auch in andere Programmiersprachen übertragen.

Alternative Programmierparadigmata sind eine Quelle für neue Techniken und Muster auch in der imperativen Programmierung. Sie stellen den Hintergrund an Allgemeinbildung über Programmierung dar.

Die Logikprogrammierung werde ich anhand der Programmiersprache *Prolog* vorstellen. Für die Funktionale Programmierung verwende ich die Programmiersprache *Scala*.

Letze Bemerkung: Dieses Skript basiert zum Teil auf einer älteren Vorlesung zu Prolog. Es ist noch nicht überall auf dem aktuell wünschenswerten Stand.

©*Prof. Dr. E. Ehses, 1996-2011*

Kapitel 2

Die Programmiersprache Prolog

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 etwas in den Hintergrund getreten. Nach wie vor ist Prolog aber das Paradebeispiel für eine Programmiersprache, die auf der Idee der Logikprogrammierung aufbaut. Es ist daher 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.

Das Kunstwort Prolog steht für **Pro**grammierung in **Log**ik. Damit ist gesagt, dass Prolog eine Brücke zwischen den Konzepten von Programmierprachen und Logik schlägt. Prolog gehört damit in den allgemeineren Kontext der *Logikprogrammierung*. Auf diesen Zusammenhang werden wir später genauer eingehen. Zunächst wollen wir Prolog als eine *regelbasierte* Programmiersprache betrachten.

Das Grundprinzip der Logikprogrammierung besteht darin, dass keine Abläufe programmiert werden, sondern dass eigentlich nur das vorhandene Wissen im Programm darzustellen ist. Die Aufgabe, auf der Basis dieses Wissens die nötigen Schlussfolgerungen zu ziehen, wird dann dem Computer überlassen (zumindest im Prinzip).

Definition:

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

In diesem Kapitel werden die wichtigsten Themen zum Verständnis eines Prolog-Programms vorgestellt. Im nächsten Kapitel geht es dann um das Thema Logikprogrammierung.

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:

©Prof. Dr. E. Ehses, 1996-2011

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

heißt nichts weiter als: „Carmen ist ein Elternteil von dem Kind Karin“. Mit etwas großzüger 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

©Prof. Dr. E. Ehses, 1996-2011

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 `pl` aufgerufen. Anschließend meldet es sich mit einer kurzen Versionsmeldung und dann mit dem interaktiven Prompt:

```
    1 ?-
```

Die 1 ist einfach eine vorlaufende Nummer. Das `?-` drückt aus, das das System jetzt bereit ist, eine Anfrage entgegenzunehmen.

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 (`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.

©Prof. Dr. E. Ehses, 1996-2011

```
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 bring 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 Eingabeprompt. 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'
```

©Prof. Dr. E. Ehses, 1996-2011

```
33 yes
```

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 Prologsitzung 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 Syntaxregeln 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 *Syntaxregeln* kurz beschrieben.

2.2.1 Lexikalische Grundelemente

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

©Prof. Dr. E. Ehses, 1996-2011

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').
```

©Prof. Dr. E. Ehses, 1996-2011

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
```

In einer „normalen" Programmiersprachen kann eine Variable in zwei Rollen auftreten: Rechts vom Zuweisungszeichen steht sie für einen Wert, links von der Zuweisung aber für eine Speicherzelle.

Auch in Prolog gibt es 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.

Die mit einem Unterstrich beginnenden Variablennamen haben eine Sonderrolle. Einmal wird ihr Wert bei interaktiver Verwendung nicht ausgegeben und zum andern spielt der einzeln stehende Unterstrich die Rolle einer anonymen einmaligen Variablen (d.h. jedes Vorkommen von _ bezeichnet eine andere 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.

©Prof. Dr. E. Ehses, 1996-2011

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, da das Stringkonzept in Prolog nicht ganz einheitlich gehandhabt wird und da wir letztlich mit Atomen einen ausreichenden Ersatz haben, werde ich auf Strings nicht weiter eingehen.

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*[1] 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.

[1]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.

©Prof. Dr. E. Ehses, 1996-2011

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.

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 Logik geht es nicht primär um die Berechung 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.

2 ?- 3 + 5 = 8.
false.

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

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.

©Prof. Dr. E. Ehses, 1996-2011

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 den Mechanismen des Funktionsaufrufs und der Funktionsausführung in prozeduralen Sprachen.

2.3.1 Die Unifikation

Zunächst sollten wir uns mit dem Aufrufmechanismus von Prolog vertraut machen. Wie Sie vielleicht an den Beispielen gesehen haben, ist da sicher einiges anders als bei anderen Programmiersprachen.

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.

Definition:

*Das Ziel der **Unifikation** besteht darin zwei Terme in Übereinstimmung zu bringen. Dabei wird überprüft ob Konstanten übereinstimmen und ob sich Teilterme rekursiv unifizieren lassen. Bisher ungebundene Variablen werden an die entsprechenden Terme gebunden. Wenn die Unifikation erfolgreicht ist, können alle Variablen durch die „allgemeinster Unifikator“ genannte Variablensubstitution gebunden werden. Der allgemeinste Unifikator ist dadurch definiert, dass er nicht mehr als die unbedingt nötigen Variablenbindungen vornimmt.*

Wenn man die Unifikation in Prolog 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.

©Prof. Dr. E. Ehses, 1996-2011

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.

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.

Die Unifikation übernimmt in Prolog die Funktion der Parameterübergabe von funktionalen Programmiersprachen. Sie stellt verglichen mit der Parameterübergabe einen erheblich mächtigeren Mechanismus dar. 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).
no
```

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.

©Prof. Dr. E. Ehses, 1996-2011

2.3.2 Die Resolution

Die gerade besprochene Unifikation stellt im Vergleich zu prozeduralen Sprachen – bei allen Unterschieden! – das Gegenstück zum Funktions*aufruf* dar. Was uns jetzt noch fehlt, ist das Gegenstück zur Funktions*ausführung*! Diese Lücke wird durch das geschlossen, was in Prolog unter dem Begriff der *Resolution* bekannt ist. Um diesen Begriff zu erläutern, führe ich ein etwas komplizierteres 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)`. Die Durchführung führt zu den Bindungen $Z = X$ und $Y =$ karin. Aber wiese 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.

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)
         ------------------------------
            []
```

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

©Prof. Dr. E. Ehses, 1996-2011

dieser Denkweise nichts anderes als die Unifiktion 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.

Im Rahmen dieser kurzen Einführung will ich hier summarisch die wichtigsten Punkte der Resolution aufführen. Dabei gehe ich auch auf ein paar bisher nicht angeführte Punkte ein:

- Bei der Beantwortung einer Anfrage, die ja im allgemeinen Fall eine Liste 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 `no`.
- Wenn die Unifikation gelingt, wird zunächst das Zielliteral aus der Anfrage entfernt.
- Als nächstes werden eventuelle Literale des Regelkörpes an die Anfrage angehängt.
- 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

©Prof. Dr. E. Ehses, 1996-2011

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. Diesesmal 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.

Die Suche nach einer Lösung kann konzeptionell durch einen Baum dargestellt werden. In diesem Suchbaum (siehe Abb. 2.1) entsprechen die Verzweigungen

©Prof. Dr. E. Ehses, 1996-2011

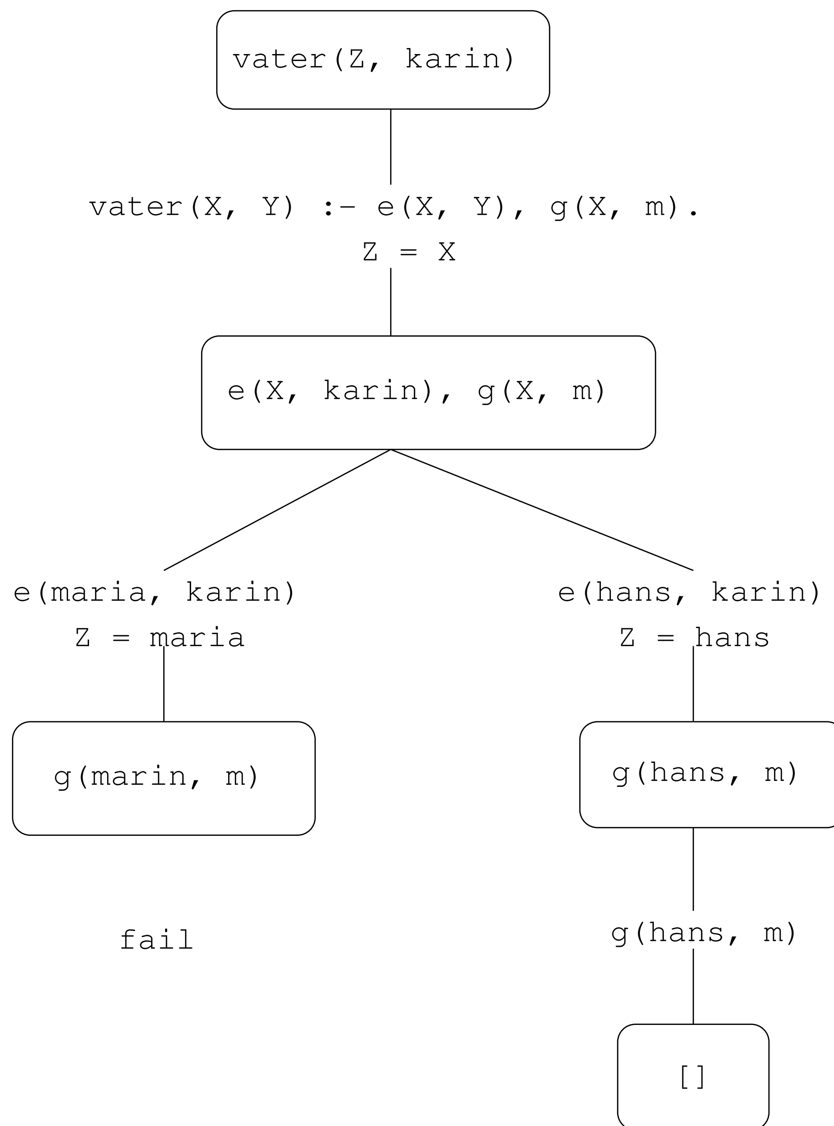

Abbildung 2.1: Prolog-Suchbaum. Die Ovale stellen die jeweiligen Resolventen dar.

©Prof. Dr. E. Ehses, 1996-2011

den jeweiligen Alternativen. Der Prolog-Ablauf erscheint als eine besondere Art der Suche in diesem Baum (es wird ein `[]`-Blatt gesucht). Der Prolog-Interpreter geht dabei nach der *Tiefensuche* vor.

Dies ist ganz grob der normale Ablauf des Prolog-Interpreters. Eine Aushnahme von diesem Ablauf bilden eingebaute Systemprädikate. Am einfachsten merken Sie sich, dass bei der Behandlung eines internen Prädikats einfach ein vordefinierter Ablauf den normalen Resolutionsmechanismus ersetzt. Allerdings sind nur ein ganz kleiner Teil der vordefinierten Prädikate wirklich eingebaut. Die allermeisten vordefinierten Prädikate sind bereits in Prolog selbst definiert.

Im nächsten Abschnitt will ich Ihnen noch eine andere Sichtweise auf Prolog-Programme erläutern, mit der die Brücke zur prozeduralen Programmierung geschlagen werden kann.

2.3.3 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. Diese Einzelheiten der Prolog-Arithmetik werden aber noch im nächsten Abschnitt besprochen. 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),
```

©Prof. Dr. E. Ehses, 1996-2011

```
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 (dann führt die Unifikation praktisch den Vergleich aus) oder aber auch als eigene Bedingung formuliert werden, wi 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 verarbeitet.
- Eine Alternative wird in Prolog durch Angabe mehrerer Klauseln zu einem Prädikat ausgedrückt.
- Die Wiederholung wird in Prolog durch rekursive Prädikatsaufrufe realisiert.[2]

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. Damit kann eine Variable letztlich zwar für einen beliebigen, jedoch nur für einen einzigen Wert stehen.

2.4 Abweichungen von der Logik

2.4.1 Eingebaute Prädikate

Arithmetik spielt also 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.
```

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

©Prof. Dr. E. Ehses, 1996-2011

```
yes
?- 3 * 4 = 12.
no
?- X is 3 * 4.
X = 12.
?- 3 * 4 =:= 12.
yes
```

so müssen Sie 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. Wenn Variable in einem Ausdruck stehen,Da Prolog zunächst streng auf Logik aufbaut, müsste Arithmetik eigentlich auch logisch begründet werden. Das heißt, ehe Prolog überhaupt in der Lage ist, numerische Berechnungen durchzuführen, müssten also zunächst alle möglichen mathematischen Definition in das Prolog System integriert werden. Ich weiß nicht, ob das schon einmal jemand versucht hat. In Prolog baut man jedoch nicht auf einer solchen formal logischen Definition der Mathematik auf. Und das hat einen ganz einfachen Grund: Wenn mathematische Operationen *effizient* ausgeführt werden sollen, dann muss die Hardware der CPU ausgenützt werden!

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
```

©Prof. Dr. E. Ehses, 1996-2011

Die Vergleichsoperationen werden so ausgeführt, dass zunächst beide Seiten arithmetisch ausgewertet werden und anschließend ein Vergleich der Ergebnisse stattfindet. Bei den nicht-arithmetischen Vergleichsoperationen findet dagegen diese arithmetische Auswertung nicht statt. Beispiel:

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 natürlich genauso vordefiniert wie die wichtigsten *Systemprädikate*
- 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.
- Viele Prädikate, wie z.B. *Listenprädikate* sind deshalb vordefiniert, weil sie sehr häufig verwendet werden.

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. Sie erschwert allerdings manchmal die Fehlersuche.

Anders ist es mit positiver Verneinung. Als Beispiel soll die Formulierung von gleich und ungleich dienen.

```
gleich(X, X).

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

©Prof. Dr. E. Ehses, 1996-2011

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:

Die folgenden Möglichkeiten sind weniger ein Programm als eine Erläuterung:

```
ungleich1(X, Y):- X \= Y.  % vordefiniertes \=

ungleich2(X, Y):- \+ X = Y.% vordefiniertes \+ (not)

ungleich3(X, X):- !, fail. % vordefiniertes ! (cut)
ungleich3(_, _).
```

Die ersten beiden Definitionen sind einfacher verständlich als die dritte. Daher sind sie für den Zweck der Verneinung natürlich vorzuziehen.

Das Beispiel mit dem Cut (`!`) ermöglicht sehr vielseitige Optimierung 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 bei `ungleich3` verwendete 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 zunächts 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 „Workaround", die oft nötig aber auch 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.

©Prof. Dr. E. Ehses, 1996-2011

2.5.1 Anforderungen

Als vollständige Verkörperung von der Logik sollte Prolog mehrere Mindesanforderungen 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 Widderspruch. Prolog löst diesen Konflikt indem es einige Kompromisse eingeht.

2.5.2 Logik und Prolog-Syntax

Prolog ist von vornherein auf die Prädikatenlogik 1. Stufe beschränkt. Dies bedeutet, dass logische Formeln mit den bekannten logischen Operatoren der Aussagenlogik aufgebaut sein können. Darüber hinaus kennt die Prädikatenlogik Variable und funktionale Ausdrücke. Variablen können quantisiert sein(für alle x, es gibt ein x). Allerdings können in der 1. Stufe der Logik 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.[3]

Prolog nimmt nun, wie gesagt ein paar Vereinfachungen vor. Zunächst vermeidet es die Quantisierungsoperatoren. Dies gelingt relativ leicht. Für Existensquantoren 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 Aussgabenlogik die Normalisierung von logischen

[3]Man braucht aber bereits für die formale Definition der Eigenschaften der natürlichen Zahlen höhere Logiksysteme.

©Prof. Dr. E. Ehses, 1996-2011

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 Literateln schreiben können. Die Literale sind positive oder negative atomare Ausdrücke. In der Klauselschreibweise schreibt man dann die Negation meist durch ein Minuszeichen. Die Formel sieht in Klauselform so aus:

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

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.

In der Klauselform gibt es ein paar Namensgebungen. Zunächst unter scheidet man *positive Literale* und *negative Literale*. Dann spricht man von *Einheitsklausen*, wenn die Klausel nur ein Literal enthält. letzte Aufgrund ihrer besonderer 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. Schauen wir uns die oben angegebene Klausenmenge 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.`

©Prof. Dr. E. Ehses, 1996-2011

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.[4]

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 einerseits 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. Der zweite Bestandteil ist die Anfrage. In Prolog wird eine Anfrage als eine verneinte Aussage aufgefasst. Ist nun die Anfrage selbst aus dem Programm herleitbar, dann steht die Verneinung der Anfrage sicherlich 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 das Gegenteil der Frage in einem logischen Widerspruch zu dem Programm steht. 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 Aufassung 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

[4]Bedeutet das Komman in Prolog nun „und“ ($\wedge$) oder „oder“ ($\vee$)? Das kommt auf die Lesart an. In der Klauselform (Normalform) ist es eindeutig ein $\vee$. In der Schreibweise als Regel (Implikation) ist es als $\wedge$ zu lesen.

©Prof. Dr. E. Ehses, 1996-2011

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 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“.

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.

©Prof. Dr. E. Ehses, 1996-2011

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, auch wenn dies formal möglich ist. Das sind die Kosten der Effizienz. Prolog-Programmierer müssen diesem Nachteil durch entsprechende Programmierung Rechnung tragen.

©*Prof. Dr. E. Ehses, 1996-2011*

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 symbolische Ausdrücken umzugehen. Beide Eigenschaften sind wichtig für die Anwendung von Prolog in der Künstlichen Intelligenz.

Im Zusammenhang mit Paradigmen der Programmierung ist aber wichtig, dass sich auch Algorithmen und ganze Programme durch Logik formulieren lassen.

Die Formulierung von Algorithmen in Logik hat einige Besonderheiten. Viele davon finden sich auch in der demnächst zu besprechenden funktionalen Programmierung wieder:

- 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". Soweit Prolog eine feste Reihenfolge kennt, ist diese nur für prozedurale Programme von Bedeutung.
- 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 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 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 und auch der Sprung in Funktionen hinein und nachher wieder zurück einen zusätzlichen Rechenaufwand. Rekursion gilt als weniger effizient als Iteration. Selbst wenn dieser (meist sehr geringe) Zusatzaufwand vernachlässigt werdden kann, bleibt immer noch der zusätzliche Speicheraufwand. Bei einigen Algorithmen kann dies durchaus die Rekursion verbieten.

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

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

©Prof. Dr. E. Ehses, 1996-2011

```
// 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 entsteht,

In Wirklichkeit sind die "Nachteile"der Rekursion aber hausgemacht! Die Compiler imperativer Sprachen – und damit meine ich auch Java – behandeln Rekursion absolut stiefmütterlich und übersetzen rekursive Programme einfach in 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 überhaubt 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 wegbekommt. Die oben angegebene Fakultätsfunktions lässt sich so nicht 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. In diesem Fall ist es niemals nötig, die lokalen Variablen für später aufzubewahren.

Definition:

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.*

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

Wir werden auf die Technik der Umwandling in die endrekursive Form später

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

©Prof. Dr. E. Ehses, 1996-2011

bei der funktionalen Programmierung noch eingegen. 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 in aller Regel dadurch, dass man eine (oder mehrere) Akkumulatorvariable einführt, in der das Ergebnis nach und nach aufgebaut wird. Vor dem ersten Aufruf muss diese Variable 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.

3.3 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.

3.3.1 Prolog-Notation für Listen

Obwohl Prolog eine extrem einfache Syntax hat, wurde für den bequemen Umgang mit Listen eine besondere Syntax eigenführt. Prolog enthält Listenliterate und Strukturen zum Zerlegen und zum Aufbau von Listen. Natürlich enthält Prolog 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.

©Prof. Dr. E. Ehses, 1996-2011

Listenliterale sind in Prolog in eckige Klammern [und] eingeschlossen. Die Listenelemte 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 kann.

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.[2]

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

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

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

Listenliterale

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.

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

©Prof. Dr. E. Ehses, 1996-2011

Dies lässt sich auch endrekursiv formulieren:

```
length(Liste, Anzahl):-
    length(Liste, 0, Anzahl).

length([], Anzahl, Anzahl).
length([_|Rest], AnzahlBisher, Anzahl):-
    AnzahlNeu is AnzahlBisher + 1,
    length(Rest, AnzahlNeu, Anzahl).
```

Die Längenberechnung ist nun aber nicht 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 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.
% Ein Element ist in der Liste, wenn es das erste Element
   oder
% wenn es ein Element der Restliste ist.
member(Element, [Element| _]).
member(Element, [_ | Rest]):-
    member(Element, Rest).

% append(Liste1, Liste2, Liste12)
%  in Liste12 folgt Liste2 auf Liste1
%  haengt man eine Liste B an die leere Liste, erhaelt
%  man die Liste B. Eine Liste B 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 B.
append([], Bs, 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 Stärke 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). Hier sind ein paar verschiedene Formulierungen einschließlich einem allgemeinen Permutationsprädikat:

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

?- select(1, Liste, Liste1),
   select(2, Liste1, Liste2),
```

©Prof. Dr. E. Ehses, 1996-2011

```
    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 Anfangselemet 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.

3.4 Lösungssuche

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

3.4.1 Tiefensuche in Prolog

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 nachvollziebar 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 jede 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 Teilbaum unendlich lang ist, wird mittels Tiefensuche die Lösung nicht gefunden. Dagegen ist die Breitensuche ein Beispiel für eine vollständige Suchstrategie.

Als Beleg für die unvollständigkeit von Prolog nehmen Sie man bitte einmal das Beispiel von Seite 40. Dieses Mal habe ich nur die Reihenfolge der Literale vertauscht.

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

Ein ähnliches Problem taucht bei vielen 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!
weg(Ziel, Ziel).
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 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.

```
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.

3.4.2 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 ge-

©Prof. Dr. E. Ehses, 1996-2011

sucht 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),
    summe(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 - 1,
    between(1, Limit, Teiler),
    0 is Zahl mod Teiler.

% summe(Liste, Summe)
% Summe ist die Summe aller Zahlen in Liste
summe([], 0).
summe([X|Xs], S):-
    summe(Xs, S1),
    S is S1 + X.

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

Versuchen Sie wieder selbst dieses Beispiel zu verstehen. Eine Anmerkung zu `bagof`: Dieses vordefinierte Prädikat findet die Liste aller Lösungen zu einem logischen Ausdruck. Das erste Argument von `bagof` 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 efizienten Algorithmus zum Auffinden *gerader* vollkommener Zahlen entdeckt. Er konnte dabei auf den Vorarbeiten von Generationen von Mathematikern (Euklid, Ibn al-Haythan, Mersenne) aufbauen.[3]

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

©Prof. Dr. E. Ehses, 1996-2011

3.4.3 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,
    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.[4]

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 Anwendungsfällen kein Weg an kombinatorischem Ausprobieren vorbei. Dabei kommt es darauf an, soweit wie möglich die Effizienz zu erhöhen. Ein wichtiger Schritt auf diesem Weg besteht darin, Sackgassen möglichst früh zu erkennen und zu vermeiden. Dies erreicht man in dem 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
```

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

©Prof. Dr. E. Ehses, 1996-2011

```
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.

```
% selection sort
sorted([], []).
sorted(UnSorted, [Min|SortedRest]):-
    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.[5] Wenn das nicht ausreicht, führt jedoch kein Weg an der kombinatorischen Suche vorbei.

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

©Prof. Dr. E. Ehses, 1996-2011

©Prof. Dr. E. Ehses, 1996-2011

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. Zudem hat Scala als objektorientierte Sprache viele Gemeinsamkeiten mit Java. Ich hoffe, dass dadurch das Verständnis erleichtert wird. Der Kern von Scala wird hier, soweit es für diese 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`

Ein wichtiger Unterschied zu Java ist, dass in Scala vieles automatisch vom Compiler eingesetzt wird. 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 fÃ¼r 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.

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

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

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 ebebfalls 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.

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.

[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

©Prof. Dr. E. Ehses, 1996-2011

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 auch 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.

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
```

[5]Das Gegenstück zu Java-Klassenfunktionen sind die Methoden eines Singleton-Objekts. Diese erscheinen in Java als statische Methoden.

©Prof. Dr. E. Ehses, 1996-2011

```
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).

4.2.3 Scala-Singleton-Objekte

In Scala gibt es keine statischen Funktionen (diese sind ja nicht objektorientiert). Für global anzusprechende Funktionen gibt es singuläre Objekte, die mit ihrem global sichtbaren Namen angesprochen 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")
    }
}
```

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.

©Prof. Dr. E. Ehses, 1996-2011

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 Klasse definiert Personen als unveränderliche Objekte. In Java könnte die Klasse wie folgt aussehen:

```
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;
    }
```

In Scala schreibt man diese Klasse etwas kürzer. Der Konstruktor erscheint als Klassenkörper. Den Instanzvariablen werden, wenn sie nicht `private` sind, automatisch Getter- und Setter-Methoden zugeordnet.

```
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) {
```

©Prof. Dr. E. Ehses, 1996-2011

```
    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(p => p.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 Typ folg ein Gleichheitszeichen gefolgt von dem Methodenkörper.
- Der Rückgabewert einer Methode ist der Wert des zuletzt stehenden Ausdrucks.
- Geschweifte Klammern sind nur nötig, wenn die Methode aus mehreren Anweisungen besteht.

©Prof. Dr. E. Ehses, 1996-2011

Wie schon angedeutet, kann man das auch etwas kürzer schreiben. Bei nichtrekursiven Methoden braucht der Rückgabetyp nicht angegeben zu werden. [6]

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) = {
    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 hat die gleiche Syntax wie das Java-if. Allerdings hat es die Semantik des bedingten Ausdrucks.

while entspricht exakt dem Java Gegenstück.

do while ist ebenso vorhanden.

switch ist (zum Glück) gestrichen. An seiner Stelle wird das erheblich mächtigere Match-Case verwendet.

for entspricht der foreach-Schleife von Java. Entsprechende Konstrukte machen auch Zählschleifen möglich. Aber es gibt kein exaktes Gegenstück zu der elementaren For-Schleife von Java.

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

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

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

©Prof. Dr. E. Ehses, 1996-2011

```
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 maximum(a: Array[Int]) = a reduceLeft(_ 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 drei 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 besonders effizienten Code. Wir werden später noch darauf eingehen.

4.3 Wichtige Erweiterungen

4.3.1 Funktionsliterale

Als funktionale Sprache unterstützt Scala auch Funktionsobjekte. In der einfachsten Form speichert man eine Methode in einer Variablen. Man nennt dies auch *partiell ausgewertete Funktion*. Die Syntax verlangt den Methodenaufruf hinzuschreiben. Dabei genügt es aber anstelle der Parameterliste den Wildcard _ anzugeben.

©Prof. Dr. E. Ehses, 1996-2011

```
object A {
    def m() {
        println("hello"‘)
    }
}

object X {
    def main(args: Array[String]) {
        val hello = A.m _
        val hallo = hello
        hallo()
    }
}
```

Es versteht sich von selbst, dass der Inhalt einer Funktionsvariablen an andere Variablen und auch an Methoden weitergereicht werden kann.

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*.

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 hervorgeben, 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.reduceLeft((x:Int, y:Int) => x + y)
val summeKuerzer = a.reduceLeft((x,y) => x + y)
val summeNochKuerzer = a reduceLeft(_ + _)
val summeGanzKurz = a sum // sum ist halt vordefiniert
```

In diesem Beispiel wird die Summe aller Zahlen eines Arrays mittels der Funktion `reduceLeft` berechnet. Diese Funktion kann eine Liste oder ein Array auf einen Wert reduzieren, indem von links nach rechts die Elemente mittels der angegebenen Funktion verknüpft werden.

©Prof. Dr. E. Ehses, 1996-2011

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.

Funktionsobjekte mit Umgebung freier Variabler heißen *closure*.

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 einen auch Variable enthaltenden Ausdruck darstellen oder sie können 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 "****";
   }
}
```

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.

©Prof. Dr. E. Ehses, 1996-2011

```
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 Variable.

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 eine Funktionsdefinition die eventuell auch als partielle Definition vorliegen kann. In Scala ist die Case-Folge eine Instanz von `PartialFunction`.

©Prof. Dr. E. Ehses, 1996-2011

©Prof. Dr. E. Ehses, 1996-2011

Kapitel 5

Funktionale Programmierung

5.1 Das Paradigma der funktionalen Programmierung

Der Funktionsbegriff ist viel älter als das Nachdenken über Programmierung. Und man hat man auch schon seit Beginn des 20. Jahrhunderts begonnen darüber nachzudenken, wie sich Berechnungen mathematisch streng beschreiben lassen. Hierfür stehen Namen wie *Gödel*, *Turing* und *Church*. Insbesondere Church hat durch das von ihm entwickelte *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.

5.1.1 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, für alle möglichen Ausgangswerte die Funktionsresultate aufzulisten. Wegen der Vielzahl der Zuordnungen ist dieses Vorgehen meist aber nicht praktikabel.

In der Regel 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 Operation (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 den Definitions- und den Wertebereich der Funktion an. Dieser Angabe entspricht in der Programmierung die Festlegung der Signatur, d.h. die Angabe des Funktionsnamens und der Parameter- sowie Rückgabetypen. Es folgt dann die Definition der Funktionsgleichung. Sie baut auf auf den elementaren Funktionen der Multiplikation und Subtraktion und auf der rekursiven Anwendung der Fakultätsfunktion selbst. Gleichzeitig ist hier eine Fallunterscheidung nötig, da für die 0 eine besondere Festlegung getroffen werden muss.

Programmiersprachen legen meist wenig Wert auf die genaue Festlegung von Definitions- und Wertebereich. Bei dynamisch getypten Sprachen fehlt sogar jede Festlegung im Programm. Bei anderen Sprachen, wie Java oder Scala, ist eine ungefähre Typangabe möglich. Scala und Java kennen jedoch nicht die Möglichkeit den Zahlen erlaubten Zahlenbereich durch eine Typangabe einzugrenzen. So ist es nicht möglich, durch eine Typangabe negative Argumente zu verbieten. Solche Vorbedingungen können erst zur Laufzeit geprüft werden.

Gebundene und freie Variable

Betrachten wir einmal 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

©Prof. Dr. E. Ehses, 1996-2011

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 Funktion) selbst schon als Funktion aufgefasst werden kann.

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

Operationen auf Datenstrukturen

Von der Mathematik her 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 Java-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.[1] Man kann genau erkennen, was der Computer tut. Anders die Mathematik;

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

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

©Prof. Dr. E. Ehses, 1996-2011

In der funktionalen Programmierung werden uns ähnliche „Formeln“ begegnen, z.B. wie:

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

Man kann die Fakultätsfunktion doch definieren als „das Produkt der Zahlen von 1 bis n“:

In Java:

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

Die mathematische Definition $n! = \Pi_{\nu=1}^{n} \nu$ lautet in Scala:

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

Natürlich ist auch die rekursive Formulierung der Fakultät ist deklarativ.

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 Variablen zu verschiedenen Zeiten verschieden ist, bedeutet, dass ich nie genau sagen kann, was eine Programmweisung bedeutet und bewirkt. Es ist diese Tatsache, die prozedurale Programme so schwer zu verstehen und zu testen lässt.

Mathematische Aussagen sind dagegen allgemeingültig. Daraus ergibt sich die Beweisbarkeit. Dies hat dann aber auch zur Konsequenz, 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 wieder andere Aspekte von Interesse.

5.1.2 Grundelemente der funktionalen Programmierung

Merkmale

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

©Prof. Dr. E. Ehses, 1996-2011

- 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.

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).
- funktionale Programmierung ermöglicht weitere Optimierungen, weil der Ablauf nicht im Detail festgelegt ist. Ein Beispiel hierfür ist die einfachere automatische Parallelisierung.
- 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.
- Funktionale Programme können (relativ) problemlos für Multithreading programmiert werden. Bei prozeduraler Programmierung ist dies extrem fehleranfällig (Wettlaufbedingungen).

©Prof. Dr. E. Ehses, 1996-2011

- Auf Funktionen höherer Ordnung aufgebaute Programme sind in der Regel 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, laßst 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 nur 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.

5.2 Funktionale Programmierung am Beispiel Scala

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

5.2.1 Funktionsdefinition und Funktionsanwendung

Die Syntax der Funktionsdefinition wurde bereits 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!

©Prof. Dr. E. Ehses, 1996-2011

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" wobei man zunächst die Parameterausdrücke vereinfacht und an einzelnen Funktionsparameter gebunden hat.

```
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 kein 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!

5.2.2 Closures und Currying

Closures

Definition:

Ein in einer äußeren Umgebung definiertes Funktionsobjekt wird ***Closure*** *genannt. Darin kommt zum Ausdruck, dass die Funktion über* freie Variable *verfügt, die nicht in der Parameterliste gebunden sind, sondern in der Umgebung der Funktionsdefinition festgelegt sind. Funktionsobjekte behalten über ihren gesamten Lebenszyklus den Bezug zu dieser Umgebung von Variablen. In Java werden Closures annähernd durch anonyme Klassen nachgebildet.*

In funktionalen Sprachen sind alle Funktionsobjekte letztlich Closures. Das Konzept ist aber 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
```

Anonyme Klasse als Closure

Zur Abgrenzung soll das letzte Beispiel in Java formuliert werden.[2]. 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 auch 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.

Currying

Der Begriff Currying geht auf die Mathematiker Moses Schönfinkel und Haskell Curry zurück. Vereinfacht geht es darum, eine Paramerliste mit mehreren Para-

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

©Prof. Dr. E. Ehses, 1996-2011

meter als mehrere Listen mit jeweils einem Parametern darzustellen. Eingeführt wurde diese Technik für theoretische Untersuchungen über berechenbare Funktionen. In funktionalen Programmiersprachen, wie in Scala, wird er 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. In der Praxis bewirkt man damit oft einfach nur eine andere Schreibweise für Funktionsaufrufe.

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)
```

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)
```

©Prof. Dr. E. Ehses, 1996-2011

```
// 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 plus2 = summe(2) _    // Syntax mit _ !
println(plus2(7))         // Ausgabe = 9
```

5.2.3 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) = 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
```

©Prof. Dr. E. Ehses, 1996-2011

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)
```

©Prof. Dr. E. Ehses, 1996-2011

```
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 $f(n,p) = n! \cdot p$. Wenn man das weiß, lässt sich die Korrektheit der Gleichungsumformungen 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. Der Scala For-Ausdruck `for (i <- 1 to n)` ist auch mit dem `Range`-Objekt `1 to n` definiert, so dass sich damit keine neue Variante ergibt.

```
// endrekursiv (funktional)
def fac(n: Int) = {
  // f(i, p) = p * (n - i + 1)!
  @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 $\text{fac}(n) = f(1,1)$. Klingt kompliziert? Ist letztlich aber einfacher, als das Nachvollziehen der Schleifeninvariante in der iterativen Fassung.

©Prof. Dr. E. Ehses, 1996-2011

Beachten Sie das Auftreten der `var`-Deklarationen in der iterativen Fassung. Veränderliche Variable sind (fast) immer ein Zeichen eines imperativen Vorgehens[3]

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
}
```

Da bei der Fibonacci-Funktion sehr große Ergebnisse vorkommen können, habe ich den Datentyp `BigInt` verwendet.[4]

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.

5.2.4 Funktionsobjekte

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 ($\text{compose}(f, g)(x) = f(g(x))$).

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

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

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

©Prof. Dr. E. Ehses, 1996-2011

```
  (x: R) => f2(f1(x))

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

5.2.5 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 Definiton 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:

```
def fac(n: 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)
}
```

©Prof. Dr. E. Ehses, 1996-2011

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.[5]

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")
}
```

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!

5.2.6 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 Mechanismen 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.

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

©Prof. Dr. E. Ehses, 1996-2011

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, dass der gleiche Werte wiederholt ermittelt werden muß.[6]

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

[6] In einer rein funktionalen Anwendung ist call by name *nur* eine mögliche Optimierung. Sobald Seitenffekte, d.h. die Veränderung von Variableninhalten auftreten können, eröffnet call by name neue Möglichkeiten.

©Prof. Dr. E. Ehses, 1996-2011

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. Sie stellt 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.

©Prof. Dr. E. Ehses, 1996-2011

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.

5.2.7 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).*

```
val liste = List(1,2,3)
val ersteElement = liste.head
val restlicheElemente = liste.tail
val listeMitNull = 0::liste
```

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[Any]) = liste match {
  case Nil => true
  case _ => false
}

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 exists[T](liste; List[T], x: T): Boolean = liste match {
```

```
  case Nil => false
  case x::_ => true
  case _::tail => exists(tail, x)
}

def exists[T](liste: List[T], x: T): Boolean = liste match {
  case head::tail => (head == x) || exists(tail, x)
  case _ => false
}

def exists[T](liste: List[T], x: T): Boolean =
  if (liste == Nil) false
  else if (liste.head == x) true
  else exists(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 teilweise etwas anders aus. Die Append-Funktion wird durch den Operator `:::` ausgedrückt. Bei der Anwendung der Append-Funktion muss eine Kopie der Liste erstellt werde. Die Operation ist damit, genauso wie `length` oder `exists`, in $O(n)$. Sie können aber davon ausgehen, dass die internen Funktionen effizient durch Iteration oder durch Endrekursion implementiert sind.

```
Nil.isEmpty                  // ergibt true
List(1,2).isEmpty            // ergibt false
List(1,2).length             // ergibt 2
List(1,2):::List(3,4)        // ergibt (1,2,3,4)
List(1,2,3).exists(_ == 3)   // ergibt true
List(1,2,3).exists(_ == 5)   // ergibt false
```

Die Funktion `exists` weicht nochmals von den anderen Beispielen ab. Der als Argument erscheinende Ausdruck ist in Wirklickeit die abgekürzte Schreibweise einer anonymen Funktion. In dieser Form lässt sich `exists` besonders flexibel verwenden. Wir müssen halt nur eine Bedingung angeben, die für das gesuchte Objekt erfüllt ist.

5.2.8 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*

©Prof. Dr. E. Ehses, 1996-2011

> *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 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 Methdoe 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)
```

In Java sind solche funktionalen Anwendungen nicht so selten, wie man denkt. Denken Sie doch z.B. an die Aktionen, die man den GUI-Elementen zuordnet. Allerdings bleibt die Verwendung höherer Funktionen doch eine etwas kompliziert wirkend Struktur, die dann doch viel seltener verwendet wird, als dies bei der funktionalen Programmierung der Fall ist. Das folgende Beispiel zeigt einige typische Konstrukte.

©Prof. Dr. E. Ehses, 1996-2011

```
// 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 enthaeltUngeradeZahl = List(...).exists(_ % 2 == 1)

def dotProduct(v1: List[Double], v2: List[Double]) = {
  require (v1.length == v2.length)
  (v1, v2).zipped.map(_ * _).sum
}

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`: Erzeuge eine neue Liste mit den Ergebnissen der Funktionsanwendung auf die einzelnen Listenelemente.
- `exists`: gibt es ein Element, das die boole'sche Funktion erfüllt?
- `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.

Sicher wird man sich nicht immer für die hier verwendete Form entscheiden. Sie sieht manchmal etwas komplizierter als die gewohnte Schreibweise als For-Schleife aus. In Scala ist diese Wahl eine Geschmacksache. For-Schleifen sind letztlich eine Umschreibung für `foreach` oder für `map`.

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
```

©Prof. Dr. E. Ehses, 1996-2011

```
    for (x <- liste) s += x
    x
  }

   // furchtbar:
  var quadrate = List[Double]()
  for (x <- List(1, 2, 3, 4)) quadrate += x*x
```

Die so elegant aussehende letzte Zeile ist besonders schlecht? Warum? In jedem Schritt wird die komplette Liste kopiert. Die Laufzeit ist $O(n^2)$!

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.

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 parts = rest.partition(_ <= pivot)
    sort(parts._1):::pivot::sort(parts._2)
}
```

`partition` ist ebenfalls eine vordefinierte höhere Funktion. Sie gibt für eine Liste ein Paar von zwei Listen zurück. Die erste der beiden Listen (`parts._1` enthält die Elemente für die die angegebene Bedingung zutriff, die andere den Rest.

©Prof. Dr. E. Ehses, 1996-2011

©Prof. Dr. E. Ehses, 1996-2011

Kapitel 6

Funktionale Klassen

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 DefinitiNilonsbereichs neue Objekte des Wertebereichs zu. Dadurch, dass Funktionen erstklassige Objekte sind, lassen sich auch höhrere 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 ?
   ...
}
```

©Prof. Dr. E. Ehses, 1996-2011

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 Multithreadingumgebung 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,[2]

```
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 => zaehler == b.zaehler && nenner == b.
        nenner
    case _ => false
  }

  override def compare(b: Bruch) = (this - b).zaehler
```

[1] Vielleicht gibt es geringfügige Laufzeitunterschiede.

[2] In der entsprechenden Praktikumsaufgabe im 2. Semester haben Sie in Java eine Klasse für unveränderliche Brüche geschrieben.

©Prof. Dr. E. Ehses, 1996-2011

```
}
```

Das technische Problem besteht hier darin, dass die lokalen Variablen des prim"-aren 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 Addtion 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 Variable 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. Bei-

©Prof. Dr. E. Ehses, 1996-2011

spiele sind die Klasse `String` (hier sorgt der Compiler dafür, dass gleichlautende 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 == 1)
      apply(zaehler)
    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 einal 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:

©Prof. Dr. E. Ehses, 1996-2011

```
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 vorherein 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

©Prof. Dr. E. Ehses, 1996-2011

werden. Die Standardoperationen auf Listen `isEmpty`, `head` und `tail` werden alle in $O(1)$ ausgeführt.

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 Objelkt `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

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)
}

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`.[3] Diese Klasse hat zwei Instanzvariablen 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.

Mit diesen Funktionen sind die grundlegendsten Funktionen bereits vorhanden (ich verwende hier den Paketnamen `myDefs` um diese Klassen von der Scala-Bibliothek abzuheben).

[3] In dem Scala-System heißt diese Klasse in Wirklichkeit `::`.

©Prof. Dr. E. Ehses, 1996-2011

```
import myDefs._ // verwende meine Definitionen

val list123 = 1::2::3::Nil
println(liste123.length
var aktuell = list123
while (! aktuell.isEmpty) {
  println(aktuell.head)
  aktuell = aktuell.tail
}

list123 match {
  case Nil => // ein Ergebnis
  case Node(h,t) => // eine Operatio mit h und t
}
```

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 wir können auf die hier definierten Listen nicht direkt die vertrauten Patternmatching-Operationen anwenden. 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. In der Praxis ist das eine sehr häufige Fehlerquelle. Der „Erfinder" der Null-Referenz, Anthony Hoare, hat seine Entdeckung als den größten beruflichen Fehler seines Lebens bezeichnet!

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 bevorzug mit der gleichen Begründung auch leere Strings und Arrays der Länge 0 vor Null-Referenzen.

©Prof. Dr. E. Ehses, 1996-2011

Literaturverzeichnis

[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.

[CM89] W.F. Clocksin and C.S. Mellish, *Programmieren in Prolog*
Springer-Verlag, 1989
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, 2008
Das ist momentan 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.

[SS89] L. Sterling, E. Shapiro, *Prolog*
Springer-Verlag 1989
Ein ausgezeichnetes Prolog-Buch, das auf die Besonderheiten der Logikprogrammierung eingeht. Leider ist die deutsche Auflage des Buches vergriffen. Die 2. englische Ausgabe ist 1994 bei MIT Press erschienen.

[YA95] R. Yasdi, *Logik und Programmieren in Logik*
Prentice Hall 1995
In dem Buch wird parallel in Logik und in Prolog eingeführt.

©*Prof. Dr. E. Ehses, 1996-2011*

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.

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 Prologinterpreters 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.

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.

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*.

©*Prof. Dr. E. Ehses, 1996-2011*

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 Einheitsklausen 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.

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.

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.

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 definerte Funktion ist nicht für alle Elemente des Definitionsbereichs definiert.

©*Prof. Dr. E. Ehses, 1996-2011*

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.

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* verbundeAllerdings müssten wir dafür auch nur die Klasse `Node` in `::` umbenennen.n 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.

©Prof. Dr. E. Ehses, 1996-2011

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 gleichaussehende 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 implmentiert 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.

©*Prof. Dr. E. Ehses, 1996-2011*