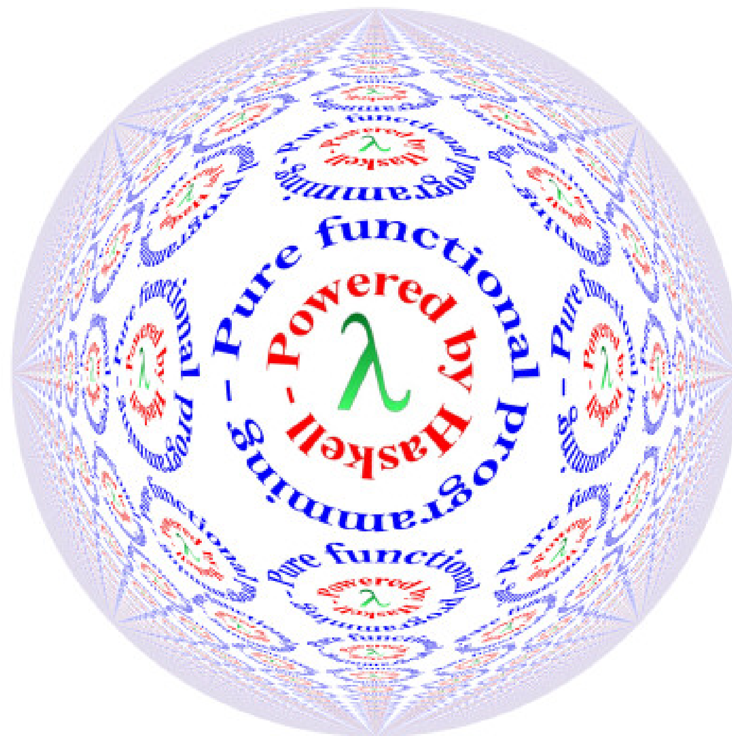
Skript zur Vorlesung

Praktische Informatik III: Deklarative Programmierung

Wintersemester 2001/02

Prof. Dr. Rita Loogen



Fachbereich Mathematik und Informatik
Philipps-Universität Marburg

Vorwort

C, Java, Pascal, Ada, und so weiter, sind allesamt imperative Programmiersprachen. „Imperativ“ bedeutet, dass Programme aus einer Folge von Kommandos bestehen, die streng eins nach dem anderen ausgeführt werden. In *deklarativen Sprachen* sind Programme hingegen Ausdrücke, die bei Ausführung eines Programms ausgewertet werden. Die Programme spezifizieren nur, *was* berechnet werden soll und *nicht, wie* die Berechnung im einzelnen durchgeführt werden soll. Von den Ausführungsdetails, die in imperativen Sprachen bestimmend sind, wird in deklarativen Sprachen abstrahiert. Demzufolge sind deklarative Programme meist kürzer und prägnanter als imperative Programme. Sie sind leicht verständlich und bieten mächtige Abstraktionsmechanismen, die die Programmierung erheblich erleichtern.

Die zweistündige Vorlesung „Deklarative Programmierung“ gehört seit dem Wintersemester 1997/98 zur Grundausbildung im Diplomstudiengang Informatik der Philipps-Universität Marburg. Sie soll grundlegende Konzepte und Methoden der deklarativen Programmierung vermitteln. Die Studierenden hören die Vorlesung im allgemeinen im dritten Semester als „Praktische Informatik III“ im Anschluss an eine fundierte Ausbildung in imperativer und objekt-orientierter Programmierung durch die Vorlesungen „Praktische Informatik I: Imperative Programmierung“ und „Praktische Informatik II: Algorithmen und Datenstrukturen“. Die deklarative Programmierung fordert von den Studierenden ein Umdenken. Während bei der imperativen Programmierung sequentielle Abläufe in Programmen beschrieben werden, geht es bei der deklarativen Programmierung um Problembeschreibungen mit mathematischen Mitteln. Diese alternative Art der Programmierung eröffnet völlig neue Perspektiven und wird sich auf den Programmierstil auswirken, sobald das essentielle Grundprinzip der Abstraktion erkannt und verinnerlicht wurde.

Zu den deklarativen Programmiersprachen zählen funktionale und Logik-Programmiersprachen. In dieser Vorlesung steht die funktionale Programmierung im Vordergrund. Anhand der Programmiersprache Haskell werden die wichtigsten Konzepte moderner funktionaler Sprachen unter dem Gesichtspunkt der Programmentwicklung und Programmiermethodik behandelt. Haskell ist eine nicht-strikte, polymorph getypte, rein funktionale Standard-Sprache. Die Sprache wurde nach dem Logiker Haskell Brooks Curry benannt, dessen Arbeiten in mathematischer Logik wichtige Grundlagen für die funktionale Programmierung bilden. Da Haskell auf dem λ-Kalkül basiert, wird das Lambda: λ als Logo verwendet. Die Graphik auf dem Deckblatt wurde von Conal Elliot und Fritz Ruehr (Microsoft Research, Cambridge UK) mit der Graphikbibliothek PAN in Haskell erstellt.

Im letzten Kapitel des Skriptes wird eine kurze Einführung in die Logik-Programmierung gegeben. Dabei wird vor allem auf Gemeinsamkeiten mit und Unterschiede zu der funktionalen Programmierung eingegangen.

Die Vorlesung wird durch zweistündige Übungen begleitet, in denen das HUGS-System, eine frei verfügbare, portable Umgebung für die Erstellung und interpretative Auswertung von Haskell-Programmen, eingesetzt wird. Hinweise zur Bedienung dieses Systems finden sich im Anhang. Eine reichhaltige Fülle von Informationen zu Haskell und anderen funktionalen Sprachen kann über die Haskell Webpage

`http://www.haskell.org`

erreicht werden. Insbesondere können von dort eine Reihe weiterer frei verfügbarer Implementierungen geladen werden, etwa der Glasgow Haskell Compiler (ghc).

Die in diesem Skript vorgestellten Fallstudien sind zum Teil den in der Literaturauswahl am Ende des Skriptes genannten Lehrbüchern entnommen. Besonders empfehlen möchte ich zum einen das Buch von Simon Thompson (University of Kent, UK), da es eine systematische Einführung in die funktionale Programmierung bietet. Zum anderen ist die anwendungsbasierte Darstellung im Buch von Paul Hudak (Yale University, USA) lesenswert, da die Vorzüge von Haskell für Multimedia-Anwendungen gezeigt werden.

Marburg, im Oktober 2001 Rita Loogen

Inhaltsverzeichnis

1 Einführung **5**

1.1 Das Programmiersprachen-Spektrum 5

1.2 Merkmale deklarativer Sprachen 6

1.2.1 Logik-Sprachen 8

1.2.2 Funktionale Sprachen 9

1.3 Historische Entwicklung funktionaler Sprachen 11

2 Grundkonzepte **15**

2.1 Funktionale Programme 15

2.2 Typen 17

2.3 Auswertung 19

2.4 Rekursive Funktionsdefinitionen 20

2.5 Lokale Definitionen 23

2.6 Operatoren 24

3 Datenstrukturen **25**

3.1 Listen 25

3.2 Pattern Matching 26

3.3 Tupel 29

3.4 Listenabstraktionen (list comprehensions) 30

3.5 Algebraische Datenstrukturen 32

3.6 Fallstudie: Implementierung einer FIFO-Queue 35

3.7 Fallstudie: Geometrische Formen 38

4 Programmeigenschaften **41**

4.1 Vollständige Induktion 42

4.2 Listeninduktion 43

4.3 Strukturelle Induktion 46

5 Interaktive Ein-/Ausgabe **49**

5.1 Monadische Ein-/Ausgabe 50

5.2 Graphik 54

5.3 Fallstudie: Sierpinski Dreiecke 56

6 Funktionen höherer Ordnung **59**
6.1 Listenverarbeitung 60
6.1.1 Transformation 60
6.1.2 Faltung 61
6.1.3 Filtern 63
6.2 Partielle Applikationen und Currying 63
6.3 λ-Abstraktionen 65
6.4 Weitere Listenfunktionale 65
6.5 Funktionale über algebraischen Datenstrukturen 66
6.6 Fallstudie: Auswertung von Polynomen 68
6.7 Fallstudie: Erstellen eines Indexes 69
6.8 Nachweis von Programmeigenschaften 74
6.9 Monadische Kompositionsfunktionen 75

7 Typen **77**
7.1 Polymorphie 77
7.2 Typinferenz 78
7.3 Typklassen 85
7.3.1 Die Klasse `Eq` 85
7.3.2 Die Klasse `Ord` 87
7.3.3 Haskells Klassenhierarchie 88

8 Auswertungsstrategien **91**
8.1 Behandlung von Pattern Matching 93
8.2 Auswertungsgrade für Redexe 95
8.3 Lazy Evaluation 97
8.3.1 Unendliche Strukturen 97
8.3.2 Verarbeitung partieller Informationen 100
8.3.3 Prozessnetze 101
8.3.4 Strombasierte Ein-/Ausgabe 102

9 Logik-Programmierung **105**
9.1 Prolog-Programme 105
9.2 Auswertungsmechanismus 107
9.3 Strukturen und logische Variablen 110
9.4 Suchbasierte Berechnungen 112

A Kurze Einführung in das HUGS System **115**
A.1 Start des Systems 115
A.2 Hugs-Kommandos 115
A.3 Syntaktische Besonderheiten in Haskell 116

Literatur **119**

Kapitel 1

Einführung

1.1 Das Programmiersprachen-Spektrum

Bei den Programmiersprachen (PS) unterscheidet man

- *imperative* Programmierspachen, in denen Programme eine Abfolge von *Befehlen* (*lat.* imperare = befehlen) beschreiben, und
- *deklarative* Programmiersprachen, in denen Programme „abstrakte" Problembeschreibungen (*lat.* declarare = deutlich machen, erklären) darstellen und keine Ausführungsanweisungen an einen Rechner enthalten.

Die imperativen Sprachen unterteilt man weiterhin in die sogenannten

– *konventionellen* Sprachen, wie FORTRAN, PASCAL, MODULA, ADA, C, OCCAM etc.

  und die moderneren

– *objekt-orientierten* Sprachen, wie SMALLTALK, EIFFEL, C++, Java etc.

Zu den deklarativen Sprachen zählen die

– *funktionalen* Sprachen, wie LISP, ML, Miranda, Haskell etc.

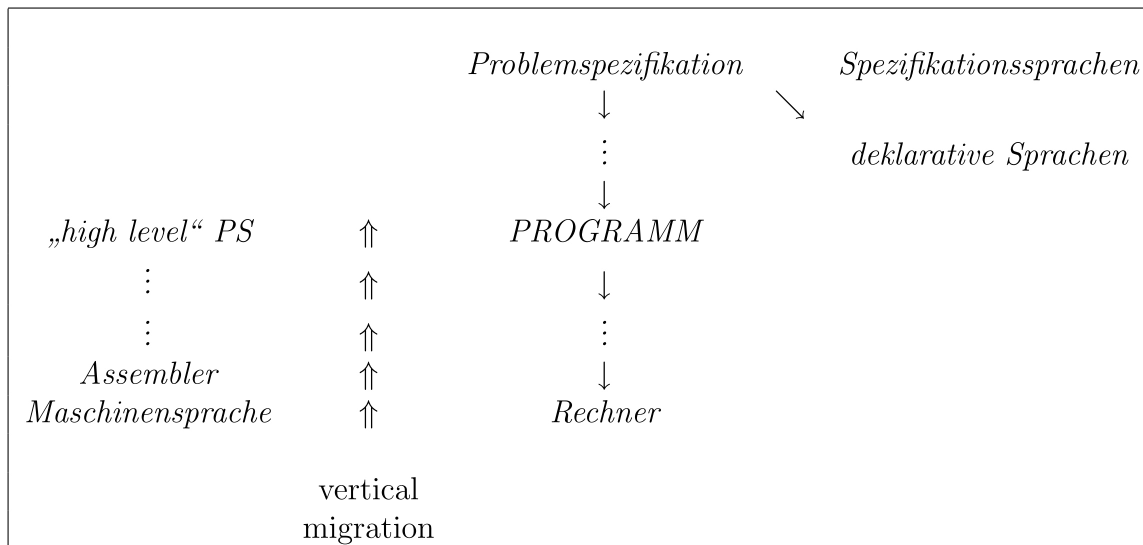
  sowie die

– *Logik*-Sprachen, deren bekanntester Vertreter PROLOG ist.

  In den letzten Jahren gab es außerdem viele Bestrebungen zur Integration dieser beiden Basisklassen deklarativer Sprachen, die unter anderem zu den sogenannten

– *funktional-logischen* Sprachen, wie IDEAL, BABEL, Curry etc., führten.

Im Zuge der Entwicklung von den ersten einfachen elektromagnetischen Rechenmaschinen zu immer leistungsfähigeren Supercomputern fand eine Evolution der Programmiersprachen statt, die man oft als *„vertical migration"* bezeichnet:



Merkmale imperativer Sprachen

Diese evolutionäre Fortentwicklung *imperativer Programmiersprachen* von maschinennahen zu „high level“ Sprachen bestimmt ihre charakteristischen Merkmale:

1. Die gemeinsame **Basis** bildet das Prinzip des *„von Neumann“ oder „stored program“ Rechners*, bei dem Programme aus Folgen von Instruktionen bestehen, die nacheinander ausgeführt werden.

2. Programme beschreiben **Berechnungen** als Abfolge von *lokalen Speichertransformationen* (Wertzuweisungen).

   Die Wertzuweisung „X := X + 3“ bewirkt die lokale Modifikation der Speicherzelle, die der Variablen X zugeordnet ist.

3. Als **Hauptkontrollstruktur** wird die *Iteration* (WHILE-Schleife) eingesetzt.

Auch Objekte in objekt-orientierten Sprachen haben einen Zustand, der durch Variablen bestimmt ist. Die Inhalte der Variablen können durch Methodenaufrufe modifiziert werden. Dabei ist die Reihenfolge der Methodenaufrufe entscheidend für die Semantik eines Programms.

1.2 Merkmale deklarativer Sprachen

Deklarative Sprachen zeichnen sich durch eine rechnerunabhängige Fundierung aus. Im Unterschied zu den imperativen Sprachen sind die Charakteristika dieser Sprachen:

1. **Basis**: mathematische Theorie

2. **Berechnung:** Manipulation von Werten

3. **Hauptkontrollstruktur:** Rekursion

Die Programmierung erfolgt auf einem vergleichsweise hohen abstrakten Niveau, nahe der Ebene von reinen Problemspezifikationen. Gemeinsam ist allen deklarativen Sprachen die *applikative Natur von Programmen*, die im wesentlichen Ausdrücke oder Formeln sind. Es gibt keine imperativen Variablen, d.h. Speicherplatzbezeichner. Berechnungen modifizieren Ausdrücke, die Werte definieren.
Aufgrund der mathematischen Fundierung existiert im allgemeinen eine *wohldefinierte formale Semantik* von Programmen, die die Basis für Korrektheitsbeweise, die Verifikation und Analyse von Programmeigenschaften und für Programmtransformationen bildet.

Deklarative Programmiersprachen zeigen besondere Vorteile auf dem Gebiet des *„Rapid Prototyping“*, da durch das hohe Abstraktionsniveau die Programmentwicklung im allgemeinen sehr schnell erfolgt und Programme zudem kürzer, prägnanter und meist auch leichter verständlich sind. Kürzere Programme sind natürlich auch weniger fehleranfällig und damit zuverlässiger und leichter zu warten. Analysen der schwedischen Telefongesellschaft Ericsson ergaben, dass durch den Einsatz der funktionalen Sprache Erlang die Programmiererproduktivität bei der Erstellung ihrer Telekommunikationssoftware um einen Faktor 9 bis 25 gesteigert werden konnte.

Im Hinblick auf die zunehmende Verfügbarkeit von Parallelrechnern erweist sich eine weitere Eigenschaft deklarativer Programme als bedeutend: *implizite Parallelität.* Diese besteht darin, dass Teilberechnungen, d. h. die Auswertung unabhängiger Programmteile, auch unabhängig, also insbesondere parallel durchgeführt werden können. Diese Eigenschaft ist eine Folge der abstrakten Form der Programmierung. Im Vordergrund steht

WAS anstelle von WIE!

Inhärent in Problemstellungen enthaltene Parallelität wird in deklarativen Programmen meist nicht künstlich in sequentielle Abläufe überführt und bleibt bei der Programmentwicklung erhalten. Dies erleichtert die automatische Parallelisierung deklarativer Programme.
Basis der impliziten Parallelität ist die sogenannte *referential transparency* von deklarativen Programmen:

> Der Wert eines Ausdrucks hängt nur von seiner Umgebung und nicht vom Zeitpunkt seiner Auswertung ab. Deshalb kann ein Ausdruck immer durch einen anderen Ausdruck mit gleichem Wert ersetzt werden (Substitutionsprinzip).

Variablen repräsentieren Werte, die unveränderbar sind, und *nicht* Speicherplätze, deren Inhalt geändert werden kann. In deklarativen Programmen bewirken lokale Berechnungen *keine Seiteneffekte.* Dies erklärt die leichte Verifizierbarkeit solcher Programme.
In Abbildung 1.1 ist ein imperatives Programm angegeben, in dem die Ausführung der Anweisung

if f(2) = f(2) **then** writeln(“ok”) **else** writeln(“nicht ok”)

wegen eines Seiteneffektes beim Aufruf der Funktion f immer die Ausgabe “nicht ok” liefert. Die formale Verifikation eines solchen Programms wirft natürlich große Probleme auf, weil ein inkrementelles Vorgehen nicht möglich ist.

```
program example;
    var flag : boolean;

    function f (n : integer) : integer;
    begin
        if flag then f:= n else f:= n+1;
        flag := not flag;
    end;

begin
    flag := true;
    if f(2) = f(2) then writeln("ok") else writeln("nicht ok");
end.
```

Abbildung 1.1: Imperatives Programm mit Seiteneffekt

1.2.1 Logik-Sprachen

Neben den funktionalen Sprachen, die in dieser Vorlesung im Zentrum stehen, zählen vor allem die **Logik-Sprachen** zu den deklarativen Sprachen.
Stichwortartig können diese Sprachen wie folgt beschrieben werden:

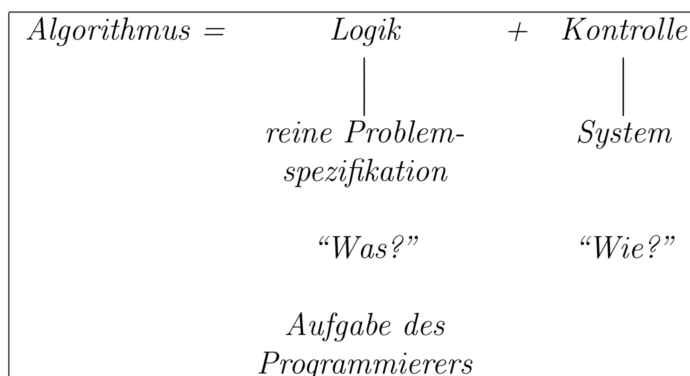
Basis: Hornklausellogik (Teillogik der Prädikatenlogik 1. Stufe)

Berechnung: "Widerlegen von Formeln" mittels Unifikation und Resolution, Widerspruchsbeweis
→ nichtdeterministische, suchbasierte Berechnungen

Programm: Menge von Formeln

Aufruf: negierte Formel

Das *Idealbild der Logik-Programmierung* liegt in der völligen Trennung von Logik und Kontrolle bei der Programmierung:



Intellektuell anspruchsvoll und damit Aufgabe des Programmierers sind vor allem die Problemlösung und -spezifikation. Die Lösungsbestimmung, d. h. die Kontrolle der

durchzuführenden Berechnung sollte dagegen vollständig vom System geleistet werden. Obwohl deklarative Sprachen einen Schritt in Richtung dieses Idealbildes darstellen, ist die vollständige Trennung von Logik und Kontrolle in den heutigen Logik-Sprachen aus Effizienzgründen noch nicht realisiert.
Die am meisten verbreitete Logik-Sprache ist **PROLOG** (**Pro**gramming in **Log**ic), das von Alain Colmerauer und seinen Kollegen Anfang der 70er Jahre an der Université Marseille im Zuge der Entwicklung eines Theorembeweisers entwickelt wurde und von R. Kowalski und seinen Kollegen an der University of Edinburgh Mitte der 70er Jahre eine saubere theoretische Fundierung erhielt.

1.2.2 Funktionale Sprachen

Im Zentrum *funktionaler Programmiersprachen* steht der aus der Mathematik geläufige Funktionsbegriff. Eine Funktion bildet Eingabewerte auf Ausgabewerte ab und definiert auf diese Weise eine Berechnung. Ein *funktionales Programm* ist eine Menge oder besser Folge von Funktionsdefinitionen. Der Aufruf eines Programms erfolgt etwa durch die Anwendung (Applikation) einer Funktion auf Eingabewerte. Die Ausgabe des Programms ist der Wert dieser Funktionsapplikation, der ermittelt wird, indem die Funktionsapplikation schrittweise durch Verwendung der Funktionsdefinitionen als Ersetzungsregeln ausgewertet wird.
Eine Kurzcharakterisierung kann wie folgt angegeben werden:

Basis: λ-Kalkül von Church

Berechnung: Ersetzen von Ausdrücken (Reduktion)

Programm: Menge von Funktionsdefinitionen

Aufruf: Applikation von Funktion auf Eingabewerte

Abbildung 1.2 zeigt den bekannten Quicksort-Algorithmus zur Sortierung einer Folge von Zahlen in Pascal und in der funktionalen Sprache Haskell. Das Pascal-Programm benutzt eine ausgefeilte, von Hoare erfundene Technik zur Sortierung des Arrays in-place, d.h. ohne Verwendung von zusätzlichen Speicherplätzen. Das Programm ist daher besonders zeit- und platzeffizient. Allerdings wird die grundsätzliche Methodik des Sortierverfahrens verschleiert.
Dementgegen definiert das Haskell-Programm eine rekursive Funktion *quicksort*, die Listen von Werten eines beliebigen Typs `a` auf ebensolche abbildet. Der sogenannte Kontext `Ord a =>` legt fest, dass auf dem Elementtyp `a` eine Ordnungsrelation $<$ definiert sein muss. Die Funktion kann also auf beliebige Listen mit Elementen, auf denen eine Ordnung definiert ist, angewendet werden, insbesondere auf Zahlenlisten. Die erste Gleichung besagt, dass die Sortierung der leeren Liste `[]` ebendiese ergibt. Die zweite Gleichung legt fest, dass eine nicht-leere Liste mit erstem Element `x` und Restliste `xs` sortiert werden kann, indem aus der Restliste alle Elemente kleiner `x` bzw. größer gleich `x` gefiltert werden — Ausdrücke (*filter* (< x) xs) bzw. (*filter* (>= x) xs) — und diese kürzeren Teillisten durch rekursive Aufrufe von *quicksort* sortiert werden. Die sortierte Ergebnisliste ergibt sich durch Aneinanderhängen (Infixoperator `++`)

a) Standard-Quicksortalgorithmus in PASCAL:

```
var a : array [1..n] of integer;
procedure quicksort (l,r : integer);
var x, i, j, tmp : integer;
begin
  if r > l then
    begin
      x := a[l]; i := l; j := r+1;
      repeat
        repeat i:=i+1 until a[i]≥x;
        repeat j:=j−1 until a[j]≤x;
        tmp := a[j]; a[j] := a[i]; a[i] := tmp;
      until j≤i;
      a[i] := a[j]; a[j] := a[l]; a[l] := tmp;
      quicksort (l, j−1); quicksort (j+1, r);
    end
end
```

b) Standard-Quicksortalgorithmus in HASKELL:

```
quicksort        :: Ord a => [a] -> [a]
quicksort []     =  []
quicksort (x:xs) =  quicksort (filter (< x) xs) ++
                    [x] ++ quicksort (filter (>= x) xs)
```

Abbildung 1.2: Gegenüberstellung eines imperativen und eines funktionalen Programms

der sortierten Teilliste der kleineren Elemente, der einelementigen Liste [x] und der sortierten Teilliste der größeren Elemente (siehe Abbildung 1.3).
Das funktionale Programm spiegelt direkt die Arbeitsweise des Quicksortalgorithmus wieder. Es ist kurz und schnell geschrieben, allerdings benötigt es weit mehr Speicherplatz und Zeit als das imperative Programm. Die Speicherplatzverwaltung erfolgt in funktionalen Programmiersprachen ähnlich wie in Java implizit. Ein „garbage collector“ entdeckt nicht mehr benötigten Speicherplatz und gibt diesen wieder frei. Moderne Implementierungen funktionaler Sprachen verwenden hierzu ausgefeilte Verfahren.
Zwischen imperativen und funktionalen Sprachen gibt es also einen Trade-off zwischen Programmier- und Laufzeiteffizienz der Programme. In Anwendungen, in denen Leistung um jeden Preis erzielt werden muss, oder wenn die möglichst direkte Umsetzung eines speicheroptimierten Algorithmus gefordert ist, sind imperative Sprachen wie C sinnvoller als Haskell, weil sie eine genaue Kontrolle über die Berechnung und die Speicherplatzbelegung erlauben. Allerdings fordern nur wenige Programme Leistung um jedem Preis! Schon seit langem wurden etwa Assemblerprogramme durch Programme in Hochsprachen verdrängt, da diese eine komfortablere Programmierumgebung bieten

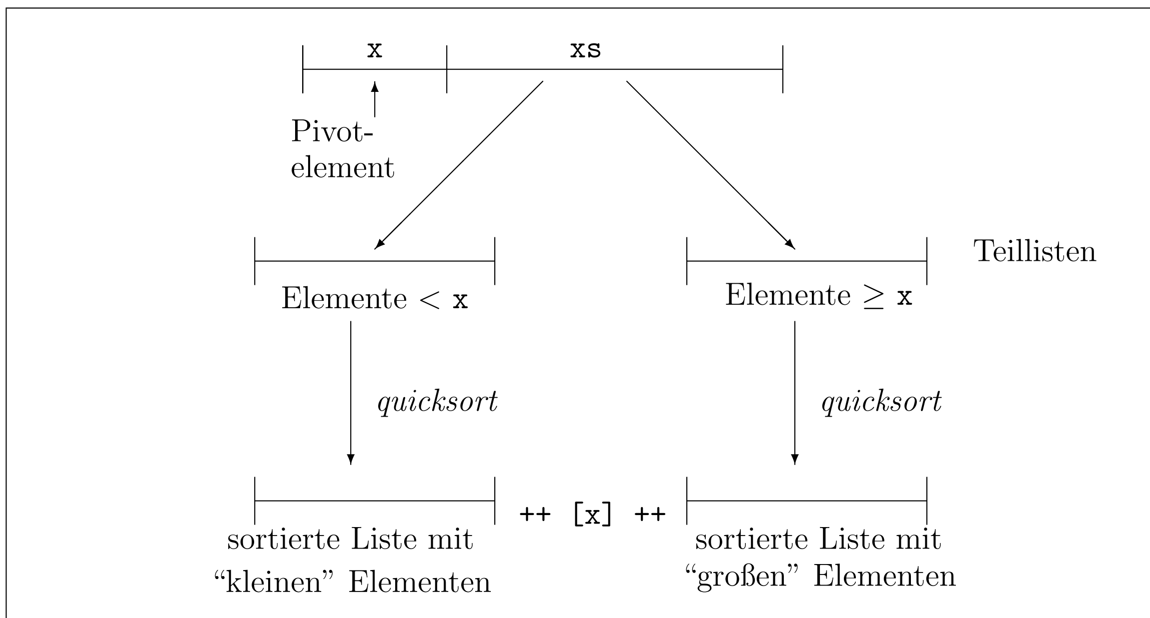


Abbildung 1.3: Arbeitsweise des funktionalen Quicksortalgorithmus

und die Compiler inzwischen so gut geworden sind, dass die Laufzeitnachteile ohne weiteres in Kauf genommen werden können. Die Fortschritte in der Compilertechnologie sind auch funktionalen Sprachen zugute gekommen. Die Laufzeiteffizienz von Haskellprogrammen liegt mittlerweile in der Größenordnung der von C-Programmen, vor allem im Bereich symbolischer Anwendungen.
Der Schlüssel zum Erfolg funktionaler Sprachen liegt in der Abstraktion. Abstraktion ist in der Programmierung generell besonders wichtig. Unter *Abstraktion* versteht man dabei im allgemeinen die Erkennung sich wiederholender Muster und die Trennung dieser allgemeinen Muster von speziellen Kontexten, in denen sie auftreten. Dies erfordert die Konzentration auf wesentliche Aspekte. Abstraktion findet in funktionalen Sprachen auf verschiedenene Ebenen statt. Die einfachste Form der Abstraktion ist die Verwendung von Namen für mehrfach verwendete Werte (Konstantendefinitionen). Weitere wichtige Abstraktionen sind funktionale Abstraktionen wie Funktionen höherer Ordnung (siehe Kapitel 6) und Datenabstraktionen wie abstrakte Datentypen (siehe Kapitel 3). Abstraktion hat zur Folge, dass Programme einfach zu entwerfen, zu schreiben und zu pflegen sind.

1.3 Historische Entwicklung funktionaler Sprachen

Der **λ-Kalkül** von Alonzo Church[1] wird oft als Ursprung und gemeinsamer Kern aller funktionalen Sprachen bezeichnet. Churchs Intention bei der Entwicklung des λ-Kalküls war eine "Formalisierung des Berechenbarkeitsbegriffs auf der Basis von Funk-

[1] A. Church: *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies N0. 6, Princeton University Press, 1941, siehe auch: H.P. Barendregt: *The Lambda Calculus: Its Syntax and Semantics*, North-Holland 1984.

tionen". Sein Ziel war vor allem die Erfassung der Grundaspekte der Berechnung von Funktionen in einem möglichst elementaren formalen Kalkül. Dies führte schließlich zu der berühmten „Churchschen These" der Berechenbarkeitstheorie, dass alle intuitiv berechenbaren Funktionen im λ-Kalkül definierbar sind. Diese These wurde durch den Nachweis, dass alternative Formalisierungen des Berechenbarkeitsbegriffes, insbesondere die Turing-Berechenbarkeit, zur λ-Definierbarkeit äquivalent sind, untermauert.

Die Sprache **LISP** (**LIS**T **P**rocessing), die Anfang der 60er Jahre von John McCarthy[2] entwickelt wurde, war die erste Sprache, die weit von den Prinzipien der damaligen, durch die Sprache FORTRAN geprägten Programmierung abwich. Die wesentlichen Neuheiten von LISP waren:

1. die Einführung von *bedingten Ausdrücken (!)* und ihre Verwendung in der Definition rekursiver Funktionen

2. die Verwendung von *Listen* als Basisstruktur und die Definition von Operationen mit funktionalen Parametern über Listen

3. die Heapverwaltung der dynamischen Listen mit *Garbage Collection* zur Freigabe nicht mehr benötigter Speicherzellen

4. der Einsatz von *S-Ausdrücken* (*engl.* symbolic expressions) zur Repräsentation von Programmen *und* Daten.

Neben diesen rein funktionalen Aspekten enthält LISP aber auch Wertzuweisungen, Hintereinanderausführung von Anweisungen und weitere seiteneffektbehaftete Konzepte.

Die effiziente Implementierung funktionaler Sprachen auf von Neumann-Rechnern war für lange Zeit ein ungelöstes Problem, das einer breiten Akzeptanz dieser Sprachen entgegenstand. Die erste abstrakte Maschine zur Auswertung von Ausdrücken des λ-Kalküls, die nach ihren Komponenten „**S**tack, **E**nvironment, **C**ontrol, **D**ump" benannte **SECD-Maschine**, wurde 1964 von Peter Landin[3] entwickelt.

In seiner Turing-Award-Lecture hielt John Backus[4] 1977 ein Plädoyer für die funktionale Programmierung und gegen den imperativen Programmierstil. In dieser Rede macht er imperative Programmierung für die Software-Krise verantwortlich und schlägt als Alternative eine auf vordefinierten Funktionalen (Funktionen, die Funktionen als Argumente oder Werte haben, wie z.B. die Funktionskomposition) basierende Sprache FP vor, deren Grundprinzip sich aber nicht durchsetzen konnte.

Weitere wichtige Sprachentwicklungen sind:

- **ML** (**M**eta **L**anguage)
  Entwickler: R. Milner, M. Gordon et al., University of Edinburgh 1975

[2] J. McCarthy: *Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I*, Communications of the ACM, Vol. 3, No. 4, 1960, 184–195.

[3] P.J. Landin: *The Mechanical Evaluation of Expressions*, Computer Journal, 6, 4, 1964.

[4] J. Backus: *Can Programming Be Liberated from the von Neumann Style? — A Functional Style and Its Algebra of Programs*, Communications of the ACM, Vol.21, No.8, August 1978.

ML wurde an der University of Edinburgh ursprünglich lediglich als Metasprache im Rahmen der Arbeiten für das automatische Beweissystem LCF zum Nachweis von Eigenschaften rekursiver Funktionen im Kontext von Programmiersprachen entworfen. Inzwischen wurde ML allerdings unabhängig von LCF weiterentwickelt und sogar standardisiert (→ Standard ML, SML). Streng genommen ist ML keine rein funktionale Sprache, da über „References“ Speicherplätze beschrieben und gelesen werden können und auch das Ein-/Ausgabesystem Seiteneffekte erzeugen kann.

Das Bedeutendste in ML ist das mächtige *polymorphe Typkonzept*, das von den meisten modernen funktionalen Sprachen adaptiert wurde.

  - Programme sind streng und statisch getypt.
  - Der Programmierer braucht jedoch keine Typdefinitionen anzugeben, denn das System inferiert die Typen automatisch.
  - Polymorphie bedeutet, dass Funktionen und Datenstrukturen unabhängig von dem Typ ihrer Argumente bzw. Komponenten definiert werden können.

- **Hope**
  Entwickler: R. Burstall, D. MacQueen, D. Sannella, University of Edinburgh, 1980

  In Hope wurden zum ersten Mal *benutzerdefinierte Datenstrukturen* und *Pattern Matching* bei der Definitionen von Funktionen über solchen Strukturen zugelassen. Diese Konzepte wurden nachträglich auch in ML eingearbeitet.

- **Miranda**
  Entwickler: D. Turner, University of Kent, 1985

  Miranda ist eine der wenigen kommerziell vertriebenen funktionalen Sprachen. Sie ist das Resultat einer Folge von Sprachentwicklungen, die von David Turner Ende der 70er und Anfang der 80er Jahre vorgenommen wurden. Turner betonte in seinen Sprachen vor allem die Verwendung von *bedingten rekursiven Gleichungen* zur Definition von Funktionen, den Einsatz benutzerdefinierter *Funktionale*, auch *Funktionen höherer Ordnung* genannt, sowie die Verwendung einer bedarfsgesteuerter Auswertungsstrategie namens *„lazy evaluation“*.

  Vorläufer von Miranda waren die Sprachen:

  - SASL (St. Andrews Static Language) und
  - KRC (Kent Recursive Calculator).

  Turner legte bei seinen Sprachentwürfen sehr viel Wert auf einfache Programmierbarkeit und Lesbarkeit. Er versuchte Funktionsdefinitionen so weit wie möglich der in der Mathematik üblichen Schreibweise von Funktionsdefinitionen anzupassen.

- **Haskell** (Vorname des Logikers H.B. Curry)
  Entwickler: Komitee unter Leitung von P. Hudak (Yale University), Ph. Wadler (Glasgow University), 1988

Haskell wurde von einem Komitee von Forschern definiert, die das Ziel verfolgten, *eine* reine funktionale Programmiersprache einzuführen, die für die funktionale Programmierung die Stellung einnehmen soll, die Prolog im Bereich von Logik-Sprachen einnimmt. Auf der FPCA (Functional Programming and Computer Architecture) Konferenz in Portland (Oregon) 1987 wurde dieses Komitee gegründet und es wurde beschlossen, der zu diesem Zeitpunkt existierenden Vielfalt von funktionalen Programmiersprachen, oft scherzhaft als „Turm von Babel“ bezeichnet, eine Sprache entgegenzustellen, die alle bewährten Konzepte inkorporiert und als gemeinsame Basis für weitere Forschungsarbeiten im Bereich funktionaler Programmiersprachen dienen soll.

Haskell unterstützt eine *enorme Vielfalt von Konzepten* und zeichnet sich insbesondere durch ein mächtiges Typsystem und eine saubere Modellierung von interaktiven Ein-/Ausgaben aus. Bemerkenswert ist, dass eine *vollständige formale Semantikdefinition*[5] existiert. Seit kurzem gibt es eine Sprachdefinition mit dem Namen Haskell98, die bei künftigen Sprachversionen und Implementierungen stabil gehalten werden soll. So kann garantiert werden, dass Haskell98-Programme auch mit künftigen Compilern und Interpretern ausgeführt werden können. Frei verfügbare Haskell-Compiler wurden in Yale, Glasgow und Göteborg entwickelt. In den Übungen zu dieser Vorlesung wird der an der University of Nottingham erstellte Haskell Interpreter hugs (Haskell User's Gopher System) eingesetzt.

Zur Implementierung funktionaler Sprachen hat sich die Technik der *Graphreduktion* durchgesetzt, die 1971 erstmals von Wadsworth[6] vorgeschlagen und 1983 erstmals von Th. Johnsson[7] und L. Augustsson[8] unter Verwendung der abstrakten *G-Maschine* in einem Compiler für eine ML Variante (Lazy ML) integriert wurde. Der Glasgow Haskell Compiler (GHC) basiert auf einer Weiterentwicklung der G-Maschine, der STG-Maschine (Spineless Tagless G-Machine)[9].
Momentane Forschungsschwerpunkte im Bereich funktionaler Sprachen sind

- die Entwicklung von Schnittstellen zu anderen Sprachen und Systemen
- die Entwicklung von Bibliotheken für spezielle Anwendungsbereiche
- die Erschließung neuer Anwendungsgebiete
- die Ausnutzung von Parallelität

und vieles mehr. Umfassende Informationen sind über die Haskell-Internetseite

`http://www.haskell.org`

zu erhalten.

[5] J. Peterson, K. Hammond (editors) : *Haskell 1.4 – A Non-strict, Purely Functional Language*, `http://haskell.org/report/index.html`

[6] C.P.Wadsworth: *Semantics and Pragmatics of the Lambda Calculus*, Ph.D. Thesis, Oxford Univ., 1971.

[7] Th.Johnsson: *Efficient Compilation of Lazy Evaluation*, SIGPLAN Notices Vol. 19, No. 6, June 1984.

[8] L.Augustsson: *A Compiler for Lazy ML*, ACM Symp. on Lisp and Functional Programming, 1984.

[9] S. L. Peyton Jones: *Implementing lazy functional languages on stock hardware: The spineless tagless G-machine*, Journal of Funct. Prog. 2(2), 1992.

Kapitel 2

Grundkonzepte

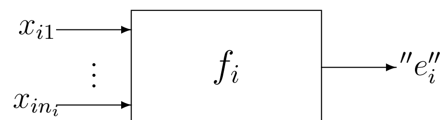
2.1 Funktionale Programme

Viele Anwendungsprobleme sind in natürlicher Weise als Funktionen definierbar. Beispiele sind neben mathematischen Funktionen wie $\sin, \cos, \sqrt{\ldots}$ etwa Textverarbeitungsprogramme wie LaTeX, das `tex`-Dateien in `dvi`-Dateien transformiert, oder Datenbanksysteme, die Anfragen in Antworten umsetzen.

Ein *funktionales Programm* ist eine Folge von Funktionsdefinitionen:

$$\begin{array}{cccc} f_1 & x_{11} \ldots x_{rn_1} & = & e_1 \\ \vdots & & & \vdots \\ \underbrace{f_r}_{\substack{\textit{Funktions-}\\ \textit{bezeichner}}} & \underbrace{x_{r_1} \ldots x_{rn_r}}_{\substack{\textit{Parameter-}\\ \textit{variablen}\\ \text{„Eingabe“}}} & = & \underbrace{e_r}_{\substack{\textit{Rumpf-}\\ \textit{ausdruck}\\ \text{„Ausgabe“}}} \end{array}$$

Eine *Funktion* transformiert Eingaben in Ausgaben:



Die Ausdrücke auf der rechten Seite von Definitionen sind geschachtelte Anwendungen (Applikationen) von Funktionen und vordefinierten Operatoren auf Werte.

Notation: Die Anwendung von Funktionen wird mit Außenklammern notiert, d.h. man schreibt etwa $(f\ 3\ 5)$ anstelle von $f(3,5)$.

Man unterscheidet *Präfix-* und *Infix*-Funktionen. Binäre arithmetische Funktionen wie $+, -, *$ sind vordefinierte Infixoperatoren, während benutzerdefinierte Funktionen Präfixfunktionen sind.

Beispiel: Folgende Definitionen bilden ein einfaches Haskell-Programm. Zeilen, die mit `--` beginnen, sind Kommentarzeilen.

```
-- einfache arithmetische Funktionen zur Addition und Quadrierung
add x1 x2    = x1 + x2
square x     = x * x
simple a b c = a * (b+c)

-- Konstantendefinitionen
newline = '\n'
eps     = 0.0000001

-- Test auf identische Argumente
allEqual n m p = (n==m) && (m==p)
```

In diesem Programm werden die vordefinierten Infixoperatoren `+` und `*` für die Addition und Multiplikation, die logische Konjunktion `&&` sowie die Vergleichsoperatoren `==` (Test auf Gleichheit, im Unterschied zum einfachen Gleichheitszeichen, das in Definitionen verwendet wird) und `>=` (größer gleich) verwendet.

◁

Haskell erlaubt Funktionsdefinitionen mit Fallunterscheidungen in folgender Schreibweise, die an die mathematische Notation erinnert:

$$
\begin{array}{lllll}
f & x_1 \ldots x_n & | & b_1 & = & e_1 \\
 & & | & b_2 & = & e_2 \\
 & & \vdots & & & \\
 & & | & b_r & = & e_r \\
 & & | & \texttt{otherwise} & = & e_{r+1}
\end{array}
$$

Nacheinander werden die Booleschen Ausdrücke $b_1, \ldots, b_r$ ausgewertet, bis der erste Ausdruck, etwa b_j, den Wahrheitswert `True` ergibt. Der entsprechende Ausdruck e_j bestimmt dann den Funktionswert. Ergeben alle Ausdrücke den Wahrheitswert `False`, so wird die mit `otherwise` eingeleitete Alternative gewählt, sofern eine solche vorhanden ist. Falls nicht, führt die Funktionsanwendung zu einem Laufzeitfehler.

Beispiel: Das obige Programm kann in einfacher Weise um eine Maximumsfunktion erweitert werden. Da die Maximumfunktion mit dem Namen `max` in Haskell vordefiniert ist, wählen wir den Namen `maxi`.

```
-- Maximumsfunktion
maxi n m | n >= m    =  n
         | otherwise =  m
```

◁

2.2 Typen

In Haskell haben alle Datenobjekte einen wohldefinierten Typ. Ein *Typ* ist eine Zusammenfassung (Menge) von Objekten gleicher Art, z.B.

> `Int` als Menge aller ganzen Zahlen (bis zu einer festen durch die Wortlänge bestimmten Größe)
> `Bool` als Menge der Wahrheitswerte `True` und `False`
> `Char` als Menge der Zeichen (Zeichenvorrat, ASCII-Zeichensatz).

Neben diesen Basistypen gibt es auch *Funktionstypen*, z.B. `Int -> Int` als Menge aller einstelligen Funktionen über den ganzen Zahlen oder `Int -> Int -> Int` als Menge aller zweistelligen Funktionen über `Int`. Aus Gründen, die später erläutert werden, verwendet man keine Produkttypen `Int x Int` für den Definitionsbereich mehrstelliger Funktionen, schreibt also nicht `Int x Int -> Int` für zweistellige Funktionen über `Int`, sondern wie oben angegeben `Int -> Int -> Int`.
In Haskell-Programmen können optional Typdeklarationen der Form `name :: type` zu Definitionen angegeben werden. Bei der Compilation von Programmen erfolgt eine automatische Typinferenz und Typüberprüfung (type checker). Nur korrekt typisierbare Programme können compiliert werden. Hierdurch werden viele Programmfehler frühzeitig erkannt.

Beispiel: Zu obigem Programm können folgende Typdeklarationen hinzugefügt werden:

```
-- einfache arithmetische Funktionen zur Addition und Quadrierung
add :: Int -> Int -> Int
add    x1     x2  =  x1 + x2

square :: Int -> Int
square    x   =  x * x

simple :: Int -> Int -> Int -> Int
simple    a      b      c   =  a * (b+c)

-- Konstantendefinitionen
newline :: Char
newline  = '\n'

eps :: Double
eps  = 0.0000001

-- Test auf identische Argumente
allEqual :: Int -> Int -> Int -> Bool
allEqual    n      m      p   =  (n==m) && (m==p)

-- Maximumsfunktion
```

```
maxi :: Int -> Int ->              Int
maxi    n      m    | n >= m     =  n
                    | otherwise  =  m
```

◁

Haskell unterscheidet sich von vielen Programmiersprachen dadurch, dass ganze und rationale Zahlen in beliebiger Genauigkeit dargestellt werden können. Der Typ `Integer` umfasst beliebige ganze Zahlen, d.h. insbesondere ganze Zahlen beliebiger Größe, genau genommen ganze Zahlen, deren Größe nicht durch die Wortlänge, sondern letztendlich nur durch die Speichergröße beschränkt ist. Da die Implementierung von ganzen Zahlen beliebiger Genauigkeit aufwändig ist, benötigen Programme, in denen der Typ `Integer` anstelle von `Int` verwendet wird, mehr Zeit und Platz. Dies ist der Preis, der für die Genauigkeit bezahlt werden muss. Welcher Typ für ganze Zahlen besser geeignet ist, hängt von der jeweiligen Anwendung ab.

Beispiel: Die Auswertung des Ausdrucks `2 * 1234567890` in den Typen `Int` und `Integer` mit dem Hugs-Interpreter zeigt deutlich den Unterschied zwischen diesen Typen:

```
Prelude> 2 * 1234567890 :: Int
-1825831516
Prelude> 2 * 1234567890 :: Integer
2469135780
Prelude> 2 * 1234567890
2469135780
```

Im Typ `Int` findet ein Überlauf statt. Voreingestellt, d.h. ohne Typangaben, wird in beliebiger Genauigkeit gerechnet. ◁

Rationale Zahlen beliebiger Genauigkeit werden über den in der Bibliothek `Ratio` definierten Typ `Rational` bereitgestellt. Für reelle Zahlen existiert keine exakte Darstellung. Wie in vielen anderen Sprachen werden diese in Haskell durch Gleitkommazahlen einfacher und doppelter Genauigkeit (Typen `Float` und `Double`) approximiert.

Beispiel: Beim Rechnen mit Gleitkommazahlen ist Vorsicht angesagt, da schnell Ungenauigkeiten auftreten können:

```
Prelude> 5 * (-0.1234) + 5 * 0.1235
0.000500023
Prelude> 5 * (-0.1234 + 0.1235)
0.000499971
```

◁

2.3 Auswertung

Bei der *Auswertung von Ausdrücken* werden die Funktionsdefinitionen als Ersetzungsregeln interpretiert. Hinzu kommen Festlegungen für die vordefinierten Operationen, z.B.

```
True  && y  =  y
False && y  =  False
```

In einem *Reduktionsschritt* wird eine Funktionsapplikation durch den Rumpf der entsprechenden Funktionsdefinition ersetzt, wobei eine Substitution der formalen Parameter durch die aktuellen Parameter vorgenommen wird. Ist $f\ x_1 \ldots x_n = e$ eine Funktionsdefinition (ohne Fallunterscheidung), so lautet die zugehörige Reduktionsregel:

$$f\ e_1 \ldots e_n \quad \Rightarrow \quad e \quad \underbrace{[x_1/e_1, \ldots, x_n/e_n]}_{\text{Substitution der Variablen durch Argumentausdrücke}} \quad .$$

Ein reduzierbarer Ausdruck wird im allgemeinen *Redex* (*red*ucible *ex*pression) genannt. Ein Ausdruck ohne reduzierbare Teilausdrücke heißt in *Normalform*. Die Normalform eines Ausdrucks ist also das Resultat seiner Auswertung.

Beispiel: `square 5 => 5*5 => 25`

```
simple 2 3 4 => 2 * (3+5) => 2 * 8 => 16

allEqual 2 3 5  => (2==3) && (3==5) => False && (3==5)  => False
```
◁

In geschachtelten Ausdrücken gibt es meist mehrere Redexe und entsprechend verschiedene Reihenfolgen der Auswertung. Die referentielle Transparenz garantiert allerdings, dass die berechnete Antwort immer dieselbe ist.

Beispiel: In folgender Beispielauswertung existieren in jedem Schritt verschiedene Möglichkeiten zur weiteren Auswertung. Die jeweiligen Redexe sind unterstrichen. Das Ergebnis ist von der Auswertungsreihenfolge unabhängig. Hier wird eine *leftmost outermost* Strategie angewendet, d.h. in jedem Schritt wird das am weitesten außen stehende Redex und, sofern mehrere solche Redexe existieren, unter diesen das am weitesten links stehende als nächstes ausgewertet:

```
allEqual (simple 3 9 5) 42 (square 6)
         --------------    ----------
-------------------------------------
=> ((simple 3 9 5) == 42) && (42 == (square 6))
    --------------                  ----------
=> ((3*(9+5)) == 42) && (42 == (square 6))
       -----                   ----------
=> ((3*14) == 42) && (42 == (square 6))
    ------                  ----------
```

```
=> (42 == 42) && (42 == (square 6))
   ----------           ----------
=> True && (42 == (square 6))
                  ----------
   --------------------------
=> (42 == (square 6))
          ----------
=> (42 == (6*6))
          -----
=> (42 == 36)
   ----------
=> False
```

◁

Diese Beispielauswertung macht deutlich, dass Auswertungen in funktionalen Sprachen nichts anderes als Gleichheitsumformungen sind. Einfache Aussagen über Funktionen können ebenfalls durch Gleichheitsumformungen gezeigt werden. Unter Verwendung der Kommutativität der Addition kann man etwa leicht zeigen, dass folgende Gleichung für beliebige Funktionsargumente `a, b, c` gilt:

```
simple a b c = simple a c b
```

```
simple a b c
    = a * (b+c)      -- Funktionsdefinition
    = a * (c+b)      -- Kommutativitaet der Addition
    = simple a c b   -- Funktionsdefinition (umgekehrt angewendet)
```

Funktionsdefinitionen sind Gleichungen, die in beiden Richtungen verwendet werden können. Die Anwendung von links nach rechts, wie bei der Auswertung einer Funktion, nennt man Entfalten (Unfolding). Die umgekehrte Richtung nennt man Falten (Folding). Den Nachweis von Funktionseigenschaften durch Gleichungsumformungen bezeichnet man als *equational reasoning*.

2.4 Rekursive Funktionsdefinitionen

Rekursion ist die wichtigste Kontrollstruktur in funktionalen Sprachen. Rekursive Definitionen von Funktionen führen die Werte von Argumenten auf die Werte von Basisargumenten zurück.

Beispiel: Gegeben sei eine Funktion `sales :: Int -> Int`, die zu Wochennummern die in der jeweiligen Woche erfolgten Verkäufe bestimmt. Gesucht werden Funktionen

1. `totalSales :: Int -> Int`, die zu Wochennummern die bis zu dieser Woche erfolgten Verkäufe ausgibt, und

2. `maxSales :: Int -> Int`, die die Wochennummer der Woche mit den maximalen Verkäufen bis zur eingegebenen Wochennummer liefert.

Zur Definition der Funktion `totalSales` ist folgende Fallunterscheidung hilfreich:

- Für $n = 0$ gilt: `totalSales` $0 =$ `sales` 0.
- Falls $n > 0$ gilt: `totalSales` $n = \underbrace{\texttt{sales}\ 0 + \ldots + \texttt{sales}\ (n-1)}_{\texttt{totalSales}\ (n-1)} + \texttt{sales}\ n$

Dies führt zu folgender Funktionsdefinition:

```
totalSales   :: Int -> Int
totalSales 0 =  sales 0
totalSales n =  totalSales (n-1) + sales n
```

Sei etwa `sales` $0 = 7$, `sales` $1 = 3$, `sales` $2 = 5$. Dann gilt:

```
totalSales 2 => totalSales 1 + sales 2
             => (totalSales 0 + sales 1) + sales 2
             => (sales 0 + sales 1) + sales 2
             => (7 + 3) + 5  => 15
```

Die Definition einer Funktion durch Angabe mehrerer Gleichungen, die verschiedene Fälle abdecken, nennt man Definition mit *Pattern Matching*, zu deutsch Musteranpassung[1]. Die Funktion `totalSales` wird durch zwei Gleichungen definiert. Die erste Gleichung legt den Funktionswert für den Fall fest, dass das Argument der Null entspricht. Die zweite Gleichung wird in allen anderen Fällen angewendet. Die Reihenfolge der Gleichungen ist wichtig, da die Definitionsgleichungen in der angegebenen Reihenfolge auf Anwendbarkeit getestet werden und das Parametermuster der zweiten Gleichung, die Variable `n`, allgemeiner ist als das Muster der ersten Gleichung, die Null.

Alternativ kann man die Funktion `totalSales` auch mit einer Fallunterscheidung definieren. Diese Schreibweise ist aber im allgemeinen weniger übersichtlich als die Verwendung von mehreren Gleichungen mit Mustern für die Parameter:

```
totalSales   :: Int -> Int
totalSales n | n == 0    =  sales 0
             | otherwise =  totalSales (n-1) + sales n
```

Zur Definition der Funktion `maxSales` kann man analog vorgehen:

- Für $n = 0$ gilt: `maxSales` $0 = 0$.

[1] Die Musteranpassung findet eigentlich nicht bei der Definition, sondern erst bei der Anwendung einer Funktion statt. Dennoch hat sich der Begriff Pattern Matching eingebürgert.

- Falls $n > 0$ gilt:

$$\texttt{maxSales}\ n = \begin{cases} \texttt{maxSales}\ (n-1), \\ \quad \text{falls } \texttt{sales}(\texttt{maxSales}\ (n-1)) \geq \texttt{sales}\ n \\ n, \quad \text{sonst.} \end{cases}$$

Dies führt zu folgender Funktionsdefinition:

```
maxSales :: Int -> Int
maxSales 0 =  0
maxSales n | sales (maxSales (n-1)) >= sales n
                        =  maxSales (n-1)
           | otherwise    = n
```

◁

Allgemein heißt diese Form der Definition einer Funktion f `:: Int -> Int` *primitive Rekursion*:

Basisfall: Funktionsdefinition für $n = 0$

Rekursiver Fall: Definition von $(f\ n)$ durch Zurückführung auf $(f\ (n-1))$

So definierte Funktionen sind nur für natürliche Zahlen festgelegt. Die Eingabe negativer Zahlen führt zu Nichttermination. Es ist besser, die Eingabe negativer Zahlen durch eine Fallunterscheidung abzufragen und in diesen Fällen einen festen Wert oder eine Fehlermeldung auszugeben.

Fehlermeldungen können mit der vordefinierten Funktion `error` ausgegeben werden. Die Funktion nimmt als Parameter einen String, d.h. eine in Anführungszeichen eingeschlossene Zeichenkette, und gibt diese bei Aufruf mit einer Laufzeitfehlermeldung aus. Das Programm wird abgebrochen.

Beispiel: Die folgenden Definitionen der Funktionen `totalSales` und `maxSales` terminieren auch bei Eingabe negativer Wochennummern. `totalSales` liefert den Wert 0. `maxSales` stoppt mit einer Fehlermeldung.

```
totalSales   :: Int -> Int
totalSales 0 =  sales 0
totalSales n | n > 0     = totalSales (n-1) + sales n
             | otherwise = 0  -- Wert 0 bei negativer Wochennummer


maxSales   :: Int -> Int
maxSales 0 =  0
maxSales n | n < 0      = error "maxSales: negative parameter"
           | (n > 0) && sales (maxSales (n-1)) >= sales n
                        = maxSales (n-1)
           | otherwise  = n
```

◁

2.5 Lokale Definitionen

In der Definition von `maxSales` tritt der Teilausdruck `maxSales (n-1)` mehrfach auf. Dies hat zur Folge, dass dieser Ausdruck auch mehrfach ausgewertet wird. Zur Vermeidung solcher Mehrfachauswertungen und zur besseren Strukturierung von Programmen dienen *lokale Definitionen*. In Ausdrücken können lokale Bezeichner für Teilausdrücke wie folgt eingeführt werden:

$$\begin{array}{ll} let & x_1 = e_1 \\ & x_2 = e_2 \\ & \dots \\ & x_r = e_r \\ in & e \end{array}$$

Im Ausdruck e stehen die Variablen $x_1, \dots, x_r$ für die Teilausdrücke $e_1, \dots, e_r$. In Funktionsdefinitionen können lokale Definitionen mit dem Schlüsselwort *where* nachgestellt werden:

$$\begin{array}{lllll} f & x_1 \dots x_n & = & e & \\ & & & where & x_1 = e_1 \\ & & & & x_2 = e_2 \\ & & & & \dots \\ & & & & x_r = e_r \end{array}$$

Beispiel:

1. Für die Funktion `maxSales` ist folgende Definition effizienter, da für den Teilausdruck `maxSales (n-1)` eine lokale Variable `maxOld` eingeführt wird, deren Wert mehrfach referenziert wird:

```
maxSales   :: Int -> Int
maxSales 0 =  0
maxSales n | n < 0      = error "maxSales: negative parameter"
           | (n > 0) && sales maxOld >= sales n
                        = maxOld
           | otherwise  = n
                        where maxOld = maxSales (n-1)
```

2. Die folgende Funktion berechnet das Produkt aller Zahlen in einem Zahlenintervall $[m|n]$ durch sukzessive Intervallhalbierung (divide et impera!):

```
prod               :: Int -> Int -> Int
prod m n | m == n =  m
         | m > n  =  1
         | m < n  =  let mid = (m+n) `div` 2
                     in  (prod m mid) * (prod (mid+1) n)
```

2.6 Operatoren

Binäre Funktionen können in Haskell infix oder präfix definiert werden. Binäre Infixfunktionen heißen *Operatoren*. Wenn man den Bezeichner einer Präfixfunktion in Hochkommata setzt, kann dieser als Infixoperator verwendet werden, siehe etwa `‘div‘` im obigen Beispiel. Wenn man Infixoperatoren in Klammern setzt, können diese als Präfixfunktion verwendet werden.

- `f x1 x2 = e` definiert die Präfixfunktion `f`. `‘f‘` ist Infixoperator.
- `x1 op x2 = e` definiert einen Infixoperator `op`. `(op)` ist Präfixfunktion.

Beispiel: Die folgenden Ausdrücke sind in Haskell gleichbedeutend:

Infix	Präfix
`(m + n) ‘div‘ 2`	`div ((+) m n) 2`

Natürlich können Infix- und Präfix-Funktionsanwendungen auch beliebig gemischt werden: `div (m+n) 2`.

◁

Kapitel 3

Datenstrukturen

3.1 Listen

Die einfachste Datenstruktur, die in jeder funktionalen Sprache vordefiniert ist, ist die Liste. Eine *Liste* `l :: [t]` ist eine Folge von Elementen gleichen Typs `t`.

Beispiel: Folgende Listen sind zulässige Haskell-Ausdrücke:

```
[1,2,3,4,3,2,1]    :: [Int]
[True,False,True] :: [Bool]
['a','b','c']      :: [Char]
[(+),(*),div,mod] :: [Int -> Int -> Int]
```

Für den Listentyp `[Char]` wird auch das Typsynonym `String` verwendet:
`type String = [Char].`

Es ist die Kurzschreibweise `"string"` für `['s','t','r','i','n','g']` möglich. ◁

Datenstrukturen werden in funktionalen Sprachen mit *Datenkonstruktoren* gebildet. Datenkonstruktoren sind spezielle Funktionen, die frei interpretiert werden, d.h. das Ergebnis der Applikation eines n-stelligen Konstruktors C auf Ausdrücke $e_1 \ldots e_n$ ist die Struktur $(C\ e_1 \ldots e_n)$, die häufig baumartig dargestellt wird:

```
   C
  / \
e1 ... en
```

Listen werden mit Hilfe von zwei Konstruktoren erzeugt:

- `[ ] :: [t]`, genannt Nil, ist eine Konstruktorkonstante (nullstelliges Konstruktorsymbol) zur Darstellung der leeren Liste. `[ ]` hat den Typ `[t]` für beliebige Typen `t`, `t` ist hierbei Typvariable.
- `(:) :: t -> [t] -> [t]`, genannt Cons, ist ein binärer Infix-Konstruktor, der bei Applikation auf ein Listenelement und eine Liste eine neue Liste durch Hinzufügen des Listenelementes am Anfang der Liste erzeugt. `(x:xs)` bezeichnet eine Liste mit erstem Element `x`, das auch Listenkopf (engl. head) genannt wird, und

Restliste `xs`, die in der englischen Literatur „tail“ genannt wird. `(:)` hat den Typ `t -> [t] -> [t]` für beliebige Typen `t`.

Der Infixkonstruktor ist rechtsassoziativ, d.h. ein Ausdruck $e_1 : e_2 : \ldots : e_{n-1} : e_n : [\,]$ ist gleichbedeutend mit $e_1 : (e_2 : \ldots (e_{n-1} : (e_n : [\,])) \ldots)$.

Beispiel: Die im obigen Beispiel angegebenen Listen sind Kurzschreibweisen für folgende Konstruktorterme:

```
[1,2,3,4,3,2,1]         = 1:2:3:4:3:2:1:[]    :: [Int]
[True,False,True]       = True:False:True:[] :: [Bool]
"abc" = ['a','b','c']   = 'a':'b':'c':[]      :: [Char]
[(+),(*),div,mod]       = (+):(*):div:mod:[] :: [Int -> Int -> Int]
```

Dabei wurde die Rechtsassoziativität von `(:)` ausgenutzt. ◁

Die Listenkonstruktoren `[]` und `(:)` haben einen sogenannten *polymorphen Typ*, d.h. in ihrem Typ kommt eine Typvariable vor. Typvariablen sind implizit allquantifiziert, d.h. für beliebige Typen `t` hat `[]` den Typ `t` und `(:)` den Typ `t -> [t] -> [t]`. Als Typvariablen können in Haskell beliebige Bezeichner, die mit einem Kleinbuchstaben beginnen, benutzt werden.

Für Zahlenlisten gibt es in Haskell weitere praktische Kurznotationen:

- `[n..m]` steht für `[n,n+1,...,m]`, falls $n \leq m$, und entspricht der leeren Liste `[]`, falls $n > m$.

  Beispiele: `[1..5] = [1,2,3,4,5]`
  `[3.1..6] = [3.1,4.1,5.1]`

- `[n,p..m]` ist die Liste von Zahlen zwischen `n` und `m` mit Schrittweite `p-n`.

  Beispiele: `[5,4..1] = [5,4,3,2,1]`
  `[0.0,0.3..1.0] = [0.0,0.3,0.6,0.9]`

3.2 Pattern Matching

Die Definition von Funktionen über Strukturen wie z.B. Listen kann einfach und anschaulich mittels sogenanntem *Pattern Matching* erfolgen. Als formale Parameter verwendet man dabei auf der linken Seite der Funktionsdefinitionen *Terme*, d.h. Ausdrücke, die nur aus Variablen und Konstruktoren aufgebaut sind. Für die einzelnen möglichen Parametermuster wird jeweils eine gesonderte Definitionsgleichung angegeben.
Für Funktionen über Listen werden im allgemeinen zwei Definitionsgleichungen benötigt. Eine Gleichung behandelt den Fall der leeren Liste `[]` und die andere den Fall einer nicht-leeren Liste mit Parametermuster `(x:xs)`. Die Variable `x` repräsentiert den Kopf der Liste, d.h. das erste Element der Liste, und die Variable `xs` die Restliste.

Beispiel: Typische Listenfunktionen sind beispielsweise

- die Funktion length zur Bestimmung der Länge einer Liste, d.h. zur Bestimmung der Anzahl der Listenelemente:

  ```
  length         :: [a] -> Int
  length []      =  0
  length (x:xs)  =  1 + length xs
  ```

  length ist eine polymorphe Funktion, die beliebigen Listen über einem Elementtyp a eine ganze Zahl zuordnet. Die erste Definitionsgleichung besagt, dass die Länge der leeren Liste gleich Null ist. Die zweite Gleichung legt fest, dass sich die Länge einer nicht-leeren Liste (x:xs) aus der Länge der Restliste xs durch Addition von Eins ergibt.

- eine Funktion double, die alle Elemente einer Liste von ganzen Zahlen verdoppelt:

  ```
  double         :: [Int] -> [Int]
  double []      =  []
  double (x:xs)  =  (2*x) : double xs
  ```

- die polymorphe Funktion append zur Konkatenation zweier Listen:

  ```
  append            :: [a] -> [a] -> [a]
  append []      ys =  ys
  append (x:xs) ys  =  x : append xs ys
  ```

- die polymorphe Funktion reverse, die die Reihenfolge der Listenelemente umkehrt:

  ```
  reverse         :: [a] -> [a]
  reverse []      =  []
  reverse (x:xs)  =  append (reverse xs) [x]
  ```

- Funktionen head und tail zur Selektion des ersten Elementes bzw. der Restliste einer nicht-leeren Liste

  ```
  head        :: [a] -> a
  head (x:xs) =  x

  tail        :: [a] -> [a]
  tail (x:xs) =  xs
  ```

  Die Funktionen sind für leere Listen nicht definiert.

◁

Die Funktionen length, reverse, head und tail sind in Haskell vordefiniert. Ihre Definitionen finden sich in der Datei prelude.hs, in der viele Basisdefinitionen enthalten sind und die zu allen Haskell-Programmen hinzugebunden bzw. im Hugs-Interpreter als erste Datei per Default geladen wird. Die in der prelude-Datei angegebene Definitionen für length und reverse verwenden allerdings Funktionen höherer Ordnung

(siehe Kapitel 6) und unterscheiden sich daher von den hier betrachteten Festlegungen. Semantisch sind die entsprechenden Definitionen aber äquivalent.
Für die Listenkonkatenation ist in der prelude-Datei der rechtsassoziative Operator

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

definiert. Man schreibt also einfacher `xs++ys` statt `(append xs ys)`, sollte aber sorgfältig zwischen der Listenkonstruktion und -konkatenation unterscheiden:

```
(:)  ::  t -> [t] -> [t]
(++)::  [t]-> [t] -> [t]
         !
```

Ist eine Funktion mittels Pattern Matching definiert, so werden bei einem Funktionsaufruf, d.h. bei der Auswertung einer Funktionsapplikation, die aktuellen Parameter mit den formalen Parametertermen der verschiedenen Definitionsregeln verglichen, um eine Regel zu finden, zu deren Parametertermen die aktuellen Parameter passen (*Matching*). Dazu ist ein Parameterübergabemechanismus erforderlich, der die Termparameter und die Funktionsargumente schrittweise vergleicht und zerlegt.
Wird eine passende Regel gefunden, werden die in der linken Regelseite auftretenden Variablen in den Parametertermen an die entsprechenden Teilausdrücke der aktuellen Parameter gebunden (implizite Selektion) und die rechte Regelseite bezüglich dieser Bindungen ausgewertet.

Beispiel:

Definition:	`length (x : xs)`	=	`1 + (length xs)`
	↑ ↑		
Pattern Matching	↑ ↑		
	↑ ↑		
Aufruf:	`length (3 : [])`	⇒	`1 + (length [])`
		⇒	`1 + 0`
		⇒	`1`

Auswertungsschritte

◁

Beispiel: (Sortieren durch Einfügen)

Die Grundidee des Sortierens durch Einfügen besteht darin, Elemente aus einem unsortierten Bereich nacheinander in eine bereits sortierte Teilfolge an der richtigen Position einzufügen. Zu Beginn ist die bereits sortierte Teilfolge leer. Damit ergibt sich folgende Funktionsdefinition:

```
isort        :: [Int] -> [Int]
isort []      =  []
isort (x:xs) =  insert x (isort xs)
```

Die Funktion `insert` fügt das erste Element einer unsortierten Liste in die durch einen rekursiven Aufruf der Funktion sortierte Restliste ein. Die Definition von `insert` erfolgt durch Fallunterscheidung über die Struktur der Liste, in die eingefügt wird:

```
insert           :: Int -> [Int] -> [Int]
insert a []      = [a]
insert a (b:bs)
        | a < b  = a : b : bs
        | a >= b = b : insert a bs
```

◁

3.3 Tupel

Tupel sind endliche Folgen von Elementen beliebigen Typs:

$$(e_1, \ldots, e_n) :: (t_1, \ldots, t_n), \text{ falls } e_i :: t_i \text{ für alle } 1 \leq i \leq n.$$

Tupel können in Funktionsdefinitionen als Parameterterme (pattern) verwendet werden. Für Paare gibt es die vordefinierten Selektionsfunktionen `fst` (first) und `snd` (second):

```
fst :: (a,b) -> a
fst    (x,y) =  x
```

```
snd :: (a,b) -> b
snd    (x,y) =  y
```

Beispiel: Punkte in einem kartesischen Koordinatensystem werden üblicherweise als Tupel dargestellt. Der Nullpunkt in einem zweidimensionalen System kann wie folgt festgelegt werden:

```
origin :: (Float,Float)
origin =  (0.0,0.0)
```

Vektoren können durch Paare von Punkten dargestellt werden:

```
vector :: ((Float,Float),(Float,Float))
```

◁

Haskell stellt *Typsynonyme* bereit, damit komplexe, mehrfach auftretende Typen mit Namen versehen werden können:

```
type <name> = <existing type>
```

Typsynonyme führen lediglich neue Namen für bereits existierende Typen ein. Es werden keine neuen Typen definiert. Das obige Beispiel kann mit Typsynonymen besser lesbar gestaltet werden:

Beispiel: Das folgende einfache Programm definiert zweidimensionale Vektoren und Linienzüge (polylines) sowie Funktionen zur Bestimmung ihrer Länge:

```
-- type synonyms
type Point2D  = (Float,Float)
type Vector   = (Point2D, Point2D)
type Polyline = [Point2D]

origin :: Point2D
origin =  (0.0,0.0)

-- length of vectors
lengthVector :: Vector -> Float
lengthVector ((x1,y1),(x2,y2))
             =  sqrt (a*a + b*b)   -- Use Pythagoras
                where a = x2 - x1  -- length in x-dimension
                      b = y2 - y1  -- length in y-dimension


-- length of polygons
lengthPolyline            :: Polyline -> Float
lengthPolyline []         =  0
lengthPolyline [p1]       =  0
lengthPolyline (p1:p2:ps) =  lengthVector (p1,p2)
                             + lengthPolyline (p2:ps)
```

Die vordefinierte Funktion `sqrt :: Float -> Float` berechnet die Quadratwurzel (engl. square root) von Gleitkommazahlen.

In den Definitionen der Längenfunktionen werden geschachtelte Muster verwendet. Das Muster `(p1:p2:ps)` passt auf Listen mit mindestens zwei Elementen. Die Fälle entarteter Linienzüge mit keinem oder nur einem einzelnen Punkt werden gesondert behandelt. ◁

3.4 Listenabstraktionen (list comprehensions)

In der Mathematik werden häufig Mengenabstraktionen (Zermelo-Fraenkel Ausdrücke) der Form

$$\{x \mid x \in M, p(x)\}$$

zur Beschreibung von Mengen eingesetzt. Dabei wird auf vorgegebene Mengen wie M als *Generatoren* und Prädikate wie p als *Selektoren* zurückgegriffen, um neue Mengen zu beschreiben. Obiger Ausdruck beschreibt beispielsweise die Menge aller $x \in M$, für die p gilt.

In Anlehnung an solche Mengenabstraktionen wurden in Haskell *Listenabstraktionen* (list comprehensions) als Kurzschreibweise für Listen eingeführt. Solche Listenabstraktionen sind vor allem nützlich, wenn Listen als Modifikation, z.B. Filterung, bestehender Listen definiert werden können. Eine Listenabstraktion hat die Form

[expr | q_1, ..., q_n]

wobei die $q_i (1 \leq i \leq n)$ von der folgenden Form sein können:

- *Generatoren* der Form `pat <- listexp`, wobei das Muster `pat` den Typ `t` hat und `listexp` eine Liste vom Typ `[t]` definiert. In den meisten Fällen ist `pat` eine Variable.
- *Bedingungen*, d.h. beliebige Ausdrücke vom Typ `Bool`
- *lokale Bindungen*, d.h. durch `let` eingeleitete Definitionen von Bezeichnern, die in nachfolgenden Generatoren und Bedingungen benutzt werden können.

Für jeden Ausdruck bzw. jede Kombination von Ausdrücken, die durch den Generator bzw. die Generatoren produziert werden, wird ein Listenelement erzeugt, sofern die Bedingungen erfüllt sind.

List-Comprehensions stellen für Haskell ausschließlich 'syntaktischen Zucker' dar, d.h. sie erhöhen nicht die Ausdrucksmächtigkeit der Sprache, sondern dienen nur dem Programmierkomfort.

Beispiel:

1. `[x*x | x <- [1..10]]` produziert die Quadrate aller Zahlen zwischen `1` und `10`.
2. ```
   [(zeile,spalte) |  zeile <- [1..10], spalte <- [1..10],
                      zeile <= spalte]
   ```
   generiert die Liste derjenigen Paare aus Zeilen- und Spaltenindex, die im oberen Dreieck einer 10×10 Matrix liegen.
3. Die Funktion `double :: [Int] -> [Int]` kann alternativ wie folgt definiert werden:
   ```
   double l = [2*a | a <- l]
   ```
4. Die Quicksort-Funktion aus Abbildung 1.2 kann mit Listenabstraktionen wie folgt umgeschrieben werden:
   ```
   qsort        :: Ord a => [a] -> [a]
   qsort []     =  []
   qsort (x:xs) =  qsort [y | y <- xs, y <  x] ++ x :
                   qsort [y | y <- xs, y >= x]
   ```
   Der Typ von `qsort` verwendet die Typklasse `Ord a` aller Typen, auf denen eine Ordnungsrelation definiert ist. Typklassen werden in Kapitel 7 behandelt.

◁

Bei der Verwendung von Listenabstraktionen ist zu beachten, dass die Variablen im Muster von Generatoren immer lokal und Funktionsparameter nur in den Bedingungen sichtbar sind.

Beispiel: Es soll eine Funktion `project2 :: [(Int,Int)] -> Int -> [Int]` definiert werden, die zu einer Liste von Paaren `l` und einem Wert `a` die Liste von Werten `b` generieren soll, für die ein Paar `(a,b)` in `l` vorkommt. Die Definition
   ```

```
project2 l a = [b | (x,b) <- l, x == a]
```

leistet das Gewünschte. Es ist *nicht* möglich, die Funktion wie folgt zu definieren:

```
p2wrong l a = [b | (a,b) <- l]
```

Der Parameter `a` ist von der lokalen Mustervariablen `a` im Generator verschieden. Die Funktion `p2wrong` ist unabhängig vom Parameter `a`. ◁

3.5 Algebraische Datenstrukturen

In gleicher Weise wie die vordefinierten Listen durch die beiden Datenkonstruktoren `[]` und `(:)` definiert sind, kann der Benutzer in funktionalen Sprachen eigene, durch endlich viele Datenkonstruktoren definierte Datenstrukturen, sogenannte *algebraische Datenstrukturen* deklarieren. Solche Datenstrukturen werden durch einen Typkonstruktor benannt, dessen Stelligkeit sich aus der Anzahl der in den Komponententypen, also den Argumenttypen der Konstruktoren, vorkommenden Typvariablen ergibt.
In Haskell beginnen Typ- und Datenkonstruktoren mit einem Großbuchstaben. Die *Deklaration einer algebraischen Datenstruktur* hat folgende allgemeine Struktur:

$$\texttt{data}\; T\ \alpha_1 \dots \alpha_n = C_1\ t_{11} \dots t_{1m_1} \mid \cdots \mid C_k\ t_{k1} \dots t_{km_k}$$

Eine solche Deklaration definiert k Konstruktorfunktionen

$$C_j : t_{j1} \rightarrow\ \cdots \rightarrow t_{jm_j} \rightarrow (T\ \alpha_1 \dots \alpha_n)$$

wobei die Typvariablen in den Komponententypen t_{ij} aus $\{\alpha_1, \dots, \alpha_n\}$ sind. Die Datenkonstruktoren sind polymorph, d.h. sie sind für beliebige Belegungen der Typvariablen mit konkreten Typen definiert.
Um Werte eines algebraischen Datentyps ausgeben zu können, fügt man der Deklaration am Ende die Worte `deriving Show` zu. Dieser Zusatz bewirkt, dass eine Ausgabefunktion `show :: t -> String` für den betreffenden Typ t bereitgestellt wird.

Beispiel: Eine explizite Deklaration von Listen hätte demnach folgendes Aussehen in Haskell:

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

bzw. allgemein:

```
data List a = Nil | Cons a (List a)
            deriving Show
```

◁

Aufzählungstypen sind spezielle algebraische Strukturen, deren Konstruktoren allesamt Konstanten (nullstellige Datenkonstruktoren) sind.

Beispiel:
```
-- Wahrheitswerte (vordefiniert)
data Bool  = True | False
     deriving Show
-- Wochentage
data Tag   = Montag | Dienstag | Mittwoch | Donnerstag
           | Freitag | Samstag | Sonntag
     deriving Show
-- Farben
data Farbe = Rot | Blau | Gruen
     deriving Show
```

◁

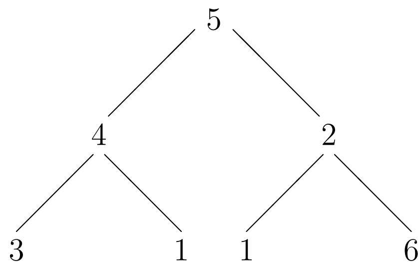
Binärbäume mit beschrifteten Knoten können wie folgt deklariert werden:

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

Der Ausdruck

`(Node 5 (Node 4 (Leaf 3) (Leaf 1)) (Node 2 (Leaf 1) (Leaf 6))) :: Tree Int`

repräsentiert den Binärbaum:



Verbundtypen (records) können durch Typen mit einem einzelnen Datenkonstruktor definiert werden.

Beispiel: In folgender Deklaration wäre derselbe Name für den Typ- und den Datenkonstruktor möglich. Zur besseren Unterscheidung werden hier unterschiedliche Bezeichner verwendet.

```
data Person = P String Int  -- Person mit Name und Alter
     deriving Show
```

◁

Pattern Matching ist für benutzerdefinierte algebraische Datenstrukturen in gleicher Weise möglich wie für die vordefinierten Listen.

Beispiel: Für Binärbäume können die folgenden Funktionen definiert werden:

- Verdopplung aller Baumeinträge:

```
doubleTree              :: Tree Int -> Tree Int
doubleTree (Leaf x)     =  Leaf (2*x)
doubleTree (Node x l r) =  Node (2*x) (doubleTree l)
                                      (doubleTree r)
```

- Summation aller Baumeinträge:

```
sumTree              :: Tree Int -> Int
sumTree (Leaf x)     =  x
sumTree (Node x l r) =  x + (sumTree l) + (sumTree r)
```

- Bestimmung der minimalen Beschriftung:

```
minTree              :: Tree Int -> Int
minTree (Leaf x)     =  x
minTree (Node x l r) =  min x (min (minTree l) (minTree r))
```

  Dabei ist `min` eine in Haskell vordefinierte Funktion:

```
min                    :: Ord a => a -> a -> a
min x y  | x < y       = x
         | otherwise   = y
```

- Bestimmung der Baumtiefe:

```
depth              :: Tree a -> Int
depth (Leaf x)     =  1
depth (Node x l r) =  1 + max (depth l) (depth r)
```

◁

Als weiteres Beispiel wird die Auswertung und Anzeige einfacher arithmetischer Ausdrücke betrachtet:

Beispiel: Zur internen Darstellung einfacher arithmetischer Ausdrücke kann ein Typ `Expr` wie folgt deklariert werden:

```
data Expr  = Lit Int
           | Add Expr Expr
           | Sub Expr Expr
```

Die folgende Funktion `eval` bestimmt den Wert eines so dargestellten Ausdrucks:

```
eval            :: Expr -> Int
eval (Lit n)    =  n
eval (Add x y) =  eval x + eval y
eval (Sub x y) =  eval x - eval y
```

Auf den Zusatz `deriving Show` wurde bei der obigen algebraischen Datenstruktur-Deklaration verzichtet, da zur Bildschirmausgabe der Ausdrücke die folgende Funktion `showExpr` definiert wird:

```
showExpr           :: Expr  -> String
showExpr (Lit n)   =  show n     -- show ist fuer Int vordefiniert
showExpr (Add x y) =  "(" ++ showExpr x ++ "+" showExpr y ++ ")"
showExpr (Sub x y) =  "(" ++ showExpr x ++ "-" showExpr y ++ ")"
```

Für vordefinierte Typen existiert in Haskell die Funktion `show :: a -> String`, die zur Bildschirmanzeige verwendet werden kann. ◁

3.6 Fallstudie: Implementierung einer FIFO-Queue

Es soll folgende als abstrakter Datentyp beschriebene FIFO-Queue (Warteschlange mit first-in-first-out Zugriff) realisiert werden. Obwohl in der Spezifikation Haskell-Syntax verwendet wird, gibt es keine direkte Unterstützung zur Spezifikation abstrakter Datentypen in Haskell:

```
abstype Queue a
  emptyQueue   :: Queue a
  enqueue      :: Queue a -> a -> Queue a
  dequeue      :: Queue a -> Queue
  firstQueue   :: Queue a -> a
  isEmptyQueue :: Queue a -> Bool
with
  dequeue (enqueue emptyQueue x) = emptyQueue
  isEmptyQueue q = False
        -> dequeue (enqueue q x) = enqueue (dequeue q) x

  firstQueue (enqueue emptyQueue x) = x
  isEmptyQueue q = False
        -> firstQueue (enqueue q x) = firstQueue q

  isEmptyQueue emptyQueue    = True
  isEmptyQueue (enqueue q x) = False
```

Abstrakte Datentypen können in Haskell mithilfe von Modulen definiert werden. Moduldeklarationen haben die Form

```
module <name> (<export list>) where <body-of-module>
```

Die Exportliste enthält die Namen von Funktionen und Datentypen, die Benutzer des Moduls verwenden können. Wird keine Exportliste angegeben, so werden alle definierten Namen des Moduls exportiert. Module werden durch die Angabe

```
import <name>
```

importiert. Diese Zeile muss immer am Anfang einer Moduldefinition oder eines Programmskripts stehen, d.h. vor allen anderen Definitionen und Deklarationen.
Im Fall der FIFO-Queue sollten die Schnittstellen-Funktionen in der Exportliste enthalten sein. Die nach dem Schlüsselwort `with` in obiger Deklaration des abstrakten

Datentyps angegebenen Gleichungen, die von den Operationen der FIFO-Queue erfüllt werden sollen, können in Haskell nicht verbindlich spezifiziert werden. Eine Angabe als Kommentar sollte auf jeden Fall vorgenommen werden. Ebenso sollte verifiziert werden, dass die im Rumpf des Moduls angegebene konkrete Implementierung die Gleichungen erfüllt. Dies kann meist mit einfachen Gleichungsumformungen erfolgen.
Als ersten Ansatz zur Implementierung einer solchen Warteschlange betrachten wir eine Listenrepräsentation:

```
module Queue
     ( Queue,
       emptyQueue,   -- Queue a
       enqueue,      -- Queue a -> a -> Queue a
       dequeue,      -- Queue a -> Queue
       firstQueue,   -- Queue a -> a
       isEmptyQueue  -- Queue a -> Bool
     ) where
          type Queue a = [a]
          ...
```

Je nachdem, ob der Anfang der Liste als Anfang oder Ende der Warteschlange betrachtet wird, ergeben sich die beiden folgenden alternativen Realisierungen:

A: Kopfelement der Liste = Schreib-Ende der Warteschlange

Damit folgt für die zu definierenden Funktionen:

```
emptyQueue     = []
enqueue q a    = a:q                          -- konstanter Aufwand
dequeue q      = reverse (tail (reverse q))  -- linearer Aufwand
firstQueue q   = head (reverse q)            -- linearer Aufwand
isEmptyQueue q = null q
```

Problematisch ist hier der lineare Aufwand für das Lesen und Löschen von Queue-Elementen.

B: Kopfelement der Liste = Lese-Ende der Warteschlange

Es folgt:

```
emptyQueue     = []
enqueue q a    = q ++ [a]    -- linearer Aufwand
dequeue q      = tail q      -- konstanter Aufwand
firstQueue q   = head q      -- konstanter Aufwand
isEmptyQueue q = null q
```

Es tritt eine Verschiebung des Aufwandes vom Schreiben zum Lesen auf. Nach wie vor hat mindestens eine der Zugriffsfunktionen einen linearen Aufwand.

Mit etwas mehr Aufwand ist eine Implementierung der Warteschlange möglich, die im Mittel sowohl für den Lese- als auch für den Schreibzugriff einen konstanten Aufwand hat. Die wesentliche Idee besteht darin, zur Darstellung der Queue zwei Listen zu verwenden. Eine Liste repräsentiert das Leseende und die andere das Schreibende der Warteschlange:

```
data Queue a = Q [a] [a]
```

Es gelten folgende Invarianten:

1. Für alle Queues `(Q fs ls)` gilt: `fs = [] -> ls = []`
2. Zwei Queues `(Q fs1 ls1)` und `(Q fs2 ls2)` sind gleich, falls

```
fs1 ++ (reverse ls1) = fs2 ++ (reverse ls2)
```

Nach diesen Vorbemerkungen ergeben sich für die Implementierung der Funktionen die folgenden Definitionen:

```
emptyQueue   =  Q [] []

enqueue (Q [] ls)      x  = Q [x] []
enqueue (Q (f:fs) ls)  x  = Q (f:fs) (x:ls)

dequeue (Q (x:(y:ys)) ls) = Q (y:ys) ls
dequeue (Q [x]        ls) = Q (reverse ls) []  -- verbleibender Fall
                                               -- mit linearem Aufwand
firstQueue (Q (x:xs) ls)  = x

isEmptyQueue (Q fs ls)    = (fs == [])
```

Nur beim Verlagern der Schreibliste in die Leseliste verbleibt der lineare Aufwand. In allen anderen Fällen haben Lese- *und* Schreibzugriffe einen konstanten Aufwand.
Mittels Gleichungsumformungen kann man einfache Anforderungen an den abstrakten Datentyp verifizieren, etwa die Gleichung:

```
dequeue (enqueue emptyQueue x) = emptyQueue
```

```
dequeue (enqueue emptyQueue x)
    = dequeue (enqueue (Q [] []) x)  -- Def. emptyQueue
    = dequeue (Q [x] [])             -- Def. enqueue
    = Q (reverse []) []              -- Def. dequeue
    = Q [] []                        -- Def. reverse
    = emptyQueue                     -- Def. emptyQueue
```

Für den Nachweis einiger Gleichungen sind aufwändigere Beweisverfahren notwendig. In Kapitel 4 wird die sogenannte Listeninduktion eingeführt.

3.7 Fallstudie: Geometrische Formen

Es soll ein Modul definiert werden, das eine Reihe von einfachen geometrischen Formen, wie Rechtecke, Kreise, Dreiecke etc., als Datenstruktur bereitstellt und Funktionen zur Flächen- und Abstandsberechnung implementiert.

```
module Shape ( Shape (...),           -- Datentyp
               Radius, Side, Vertex, -- weitere Typen
               ...
               distBetween, area     -- Funktionen
              ) where  ...
```

Als erstes muss festgelegt werden, welche Formen als Grundformen festgelegt werden und welche Formen als Spezialfälle der Grundformen behandelt werden. Quadrate sind spezielle Rechtecke, Rechtecke sind spezielle Parallelogramme und Parallelogramme sind vierseitige Polygone. Nur allgemeine Polygone als Grundformen zuzulassen, ist sicherlich zu eingeschränkt. Wir wählen daher einen Mittelweg und definieren einen algebraischen Datentyp `Shape`, in dem Rechtecke, Ellipsen, rechtwinklige Dreiecke und Polygone als Grundformen festgelegt werden:

```
data Shape = Rectangle Side Side
            | Ellipse Radius Radius
            | RtTriangle Side Side
            | Polygon [Vertex]
      deriving Show

type Radius = Float
type Side   = Float
type Vertex = (Float,Float)
```

Die Typsynonyme definieren keine neuen Typen, sondern lediglich weitere Namen für vorhandene Typen. Sie verbessern die Lesbarkeit der Datentypdeklaration, da die Typbenennung auf die Bedeutung der Daten hinweist.
Spezielle Formen können als Funktionen definiert werden:

```
square   :: Side -> Shape
square s =  Rectangle s s

circle   :: Radius -> Shape
circle r =  Ellipse r r
```

Eine Funktion `area :: Shape -> Float` zur Flächenberechnung kann mittels Pattern Matching definiert werden. Die elementaren Fälle sind direkt ersichtlich: Die Fläche eines Rechtecks ergibt sich durch die Multiplikation der Seitenlängen. Ähnlich ist die Fläche eines rechtwinkligen Dreiecks gleich der Hälfte des Produktes der Kathetenlängen. Die Fläche einer Ellipse ergibt sich als Produkt der irrationalen Zahl π mit den Radien. Im Spezialfall des Kreises sind beide Radien identisch und man erhält die bekannte Flächenformel. Es wird angenommen, dass `pi` als Konstante definiert wird.

```
area                    :: Shape -> Float
area (Rectangle  s1 s2) = s1*s2
area (RtTriangle s1 s2) = s1*s2/2
area (Ellipse r1 r2)    = pi*r1*r2

pi :: Float
pi =  3.14159
```

Komplizierter ist die Flächenberechnung eines beliebigen Polygons. Für konvexe Polygone lässt sich die Fläche zerlegen in die Fläche des Dreiecks, das aus den ersten drei Punkten gebildet wird, und die Fläche des Polygons, das durch Streichen des zweiten Punktes entsteht. Da der erste Punkt bei den sukzessiven Dreiecksberechnungen gleichbleibt, empfiehlt sich die Verwendung einer Hilfsfunktion `polyArea`, die diesen Punkt als globalen Parameter verwendet:

```
area (Polygon (v1:vs))  = polyArea vs
      where polyArea               :: [Vertex] -> Float
            polyArea (v2:v3:vs') = triArea v1 v2 v3
                                      + polyArea (v3:vs')
            polyArea _           = 0
```

Für die Berechnung der Fläche beliebiger Dreiecke mit Seitenlängen a, b und c existiert die Formel

$$\sqrt{s(s-a)(s-b)(s-c)} \text{ mit } s = \frac{1}{2}(a+b+c),$$

die in Haskell wie folgt notiert werden kann:

```
-- area of general triangle
triArea          :: Vertex -> Vertex -> Vertex -> Float
triArea v1 v2 v3 = let a = distBetween v1 v2
                       b = distBetween v2 v3
                       c = distBetween v3 v1
                       s = 0.5*(a+b+c)
                   in sqrt (s*(s-a)*(s-b)*(s-c))

distBetween :: Vertex -> Vertex -> Float
distBetween (x1,y1) (x2,y2)
  = lengthVector ((x1,y1),(x2,y2))
```

Die Berechnung der Fläche allgemeiner Polygone würde in Rahmen dieses Skriptes zu weit führen und bleibt interessierten Leserinnen und Lesern als weiterführende Übung überlassen.

Kapitel 4

Programmeigenschaften

Die referentielle Transparenz in funktionalen Programmen garantiert, dass Ausdrücke immer durch „gleichwertige“ andere Ausdrücke ersetzt werden können, ohne dass dies Auswirkungen auf die Bedeutung (Semantik) des gesamten Programms hat. Dieses Substitutionsprinzip ist die Grundlage für Programmtransformationen und den Nachweis von Programmeigenschaften. Das Ziel einer Programmtransformation ist die Ersetzung von „teuren“ durch „billige“ Ausdrücke unter Beibehaltung der Programmsemantik.

Beispiel: Der Aufwand der Standarddefinition der Listenspiegelung

```
reverse        :: [t] -> [t]
reverse []     =  []
reverse (x:xs) =  reverse xs ++ [x]
```

ist quadratisch in der Länge der Eingabeliste, denn `(++)` ist linear in der Länge seines ersten Argumentes und wird `n` mal aufgerufen, wenn `n` die Länge der Eingabeliste ist:

```
reverse [x1,...,xn]
=> reverse [x2,...,xn] ++ [x1]
=> (reverse [x3,...,xn] ++ [x2]) ++ [x1]
=> ...
=> (...((reverse [] ++ [xn]) ++ [x{n-1}]) ++...) ++ [x1]
=> (...(([] ++ [xn]) ++ [x{n-1}]) ++...) ++ [x1]      -- n+1 Schritte
=> (...([xn] ++ [x{n-1}]) ++...) ++ [x1]              -- 1 Schritt
=> (...(xn:([] ++ [x{n-1}])) ++...) ++ [x1]
=> (...[xn,x{n-1}] ++...) ++ [x1]                     -- 2 Schritte
=> ...
=> (...[xn,x{n-1},x{n-2}] ++ ...) ++ [x1]             -- 3 Schritte
=> ...
=> [xn,x{n-1},...x2] ++ [x1]
=> ...
=> [xn,x{n-1},...,x1]                                 -- n Schritte
```

Insgesamt ergibt sich ein Zeitaufwand von

$$n + 1 + \sum_{i=1}^{n} i = \sum_{i=1}^{n+1} i = \frac{(n+2)(n+1)}{2} \in O(n^2)$$

Schritten zur Bestimmung der Spiegelung einer n-elementigen Liste mit `reverse`.

Eine effizientere Version der Listenspiegelung verwendet ein zusätzliches Akkumulatorargument zur Konstruktion der gespiegelten Liste:

```
rev :: [t] -> [t]
rev    xs  =  aux xs []

aux :: [t] -> [t] -> [t]
aux    []     ys   = ys
aux    (x:xs) ys   = aux xs (x:ys)
```

Es folgt:

```
rev [x1,...,xn] => aux [x1,...,xn] []
                => aux [x2,...,xn] [x1]
                => aux [x3,...,xn] [x2,x1]
                => ...
                => aux [] [xn,...,x1]
                => [xn,...,x1]
```

Die Auswertung mit der effizienteren Version benötigt $n + 2 \in O(n)$ Schritte.

Es stellt sich nun die Frage, ob die beiden Versionen tatsächlich gleichwertig sind, d.h. gilt für alle endlichen Listen `xs`,

`reverse xs = rev xs`?

Wie kann man eine solche Aussage beweisen? ◁

Die wichtigsten Hilfsmittel zum Beweis von Eigenschaften funktionaler Programme sind die Substitutivität (Substitutionsprinzip, Gleichungsumformungen) und die *Induktion*. Als Einstieg wird zunächst kurz das Prinzip der vollständigen Induktion für natürliche Zahlen wiederholt.

4.1 Vollständige Induktion

Die Menge der natürlichen Zahlen kann wie folgt induktiv definiert werden:

1. $0 \in \mathbb{N}$
2. es gibt eine injektive Funktion *succ* mit:

   falls $n \in \mathbb{N}$ ist auch $succ(n) \in \mathbb{N}$.

3. für alle $n \in \mathbb{N}$ gilt: $succ(n) \neq 0$.

4. nur die mit 1. und 2. gebildeten Elemente liegen in $\mathbb{N}$.

Aus dieser Definition (insbesondere 4.) resultiert das Beweisprinzip der *vollständigen Induktion*:

> Zum Beweis einer Eigenschaft P, die für alle $n \in \mathbb{N}$ gelten soll ($\forall n \in \mathbb{N}.P(n)$) reicht es zu zeigen:
>
> 1. *Induktionsanfang:* Es gilt: $P(0)$.
> 2. *Induktionsschluss:* Für alle $n \in \mathbb{N}$ folgt aus $P(n)$ die Gültigkeit von $P(succ(n))$.

Außerdem muss P natürlich für alle $n \in \mathbb{N}$ definiert sein.

Beispiel: Sei $P(n) : \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Beweis mittels vollständiger Induktion:

1. *Induktionsanfang:* $\sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$.

2. *Induktionsschluss:* Sei $n \in \mathbb{N}$ mit $P(n)$ (Induktionsvoraussetzung). Dann folgt:

$$P(n+1) : \sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1) \overset{\text{IV}}{=} \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}.$$

Damit ist die Aussage bewiesen. ◁

4.2 Listeninduktion

In völliger Analogie zum Prinzip der vollständigen Induktion für natürliche Zahlen wird das Prinzip der Listeninduktion zum Beweis von Aussagen über endliche Listen entwickelt. Die Menge aller endlichen Listen über einem Grundtyp `t` wird wie folgt induktiv definiert:

1. `[]` ist Liste über `t` (`[] :: [t]`).

2. für beliebige Elemente `x :: t` und `xs :: [t]` ist auch `(x:xs)` Liste über `t` (`(x:xs) :: [t]`).

   Dabei gilt: `x:xs == y:ys` impliziert, dass `x == y` und `xs == ys` für beliebige `x`, `y` vom Typ `t` und `xs`, `ys` vom Typ `[t]` (*eindeutige Termzerlegung, Injektivität von (:)*).

3. für alle `x::t`, `xs::[t]` gilt: `(x:xs)` $\neq$ `[]`.

4. Nur die mit 1. und 2. erzeugten Elemente sind Listen über `t`.

Sei nun P eine Eigenschaft von Listen, die unabhängig von den einzelnen Listenelementen und für alle Listen definiert ist. Dann gilt P für alle endlichen Listen, falls

1. *Induktionsanfang:* $P([])$ gilt.
2. *Induktionsschluss:* für alle Elemente `x::t` und alle Listen `l::[t]` gilt: $P(\texttt{l})$ impliziert $P(\texttt{x} : \texttt{l})$.

Als erstes Beispiel zeigen wir die Assoziativität der Listenverkettung:

4.2.1 Lemma: `(++) :: [a] -> [a] -> [a]` mit

```
[]     ++   ys  = ys              (app1)
(x:xs) ++   ys  = x:(xs++ys)      (app2)
```

ist assoziativ, d.h. für beliebige endliche Listen `xs, ys, zs :: [a]` gilt:

`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.`

Beweis: Listeninduktion über den Parameter `xs`:

Fall `xs == []`: (*Induktionsanfang*):

```
[] ++ (ys ++ zs)  =        ys  ++ zs     (app1)
                  = ([] ++ ys) ++ zs     (app1R)
```

Die Angabe `app1R` bedeutet, dass die Gleichung `(app1)` von rechts nach links angewendet wurde.

Fall `xs = x:xs'` (*Induktionsschluss*):

```
(x:xs') ++ (ys ++ zs)  =  x:(xs' ++ (ys ++ zs))      (app2)
                       =  x:((xs' ++ ys) ++ zs)      (IV)
                       =  (x:(xs' ++ ys)) ++ zs      (app2R)
                       =  ((x:xs') ++ ys) ++ zs      (app2R)
```

Damit folgt die Behauptung. □

Als weiteres Beispiel wird gezeigt, dass die Funktion `reverse` zu sich selbst invers ist.

4.2.2 Lemma Für jede endliche Liste `xs :: [t]` gilt:

`reverse (reverse xs) = xs.`

Dabei sei `reverse` wie üblich definiert:

```
reverse    []     =  []                           (rev1)
reverse    (x:xs) =  reverse xs ++ [x]            (rev2)
```

Beweis: Listeninduktion über den Parameter `xs`:

Fall `xs = []`: `reverse (reverse []) = reverse [] = [].`

Fall `xs = x:xs'`: Es gilt:

```
reverse (reverse (x:xs'))  = reverse (reverse xs' ++ [x]) =  ?
```

An dieser Stelle des Beweises wird ein Hilfsresultat über den Zusammenhang zwischen der Listenkonkatenation und der Listenspiegelung benötigt:

Hilfsresultat: Für alle `y :: t` und `ys :: [t]` gilt:
`reverse (ys ++ [y]) = y: reverse ys.`

Beweis mittels Listeninduktion:

Im Fall `ys = []` gilt:

```
reverse ([] ++ [y]) = reverse [y]         (app1)
                    = reverse [] ++ [y]   (rev2)
                    = [] ++ [y]           (rev1)
                    = [y]                 (app1)
                    = y:reverse []        (rev1R)
```

Im Fall `ys = z:zs` gilt:

```
reverse ((z:zs) ++ [y]) = reverse (z:(zs ++ [y]))      (app2)
                        = reverse (zs ++ [y]) ++ [z]   (rev2)
                        = (y:reverse zs) ++ [z]        (IV)
                        = y:(reverse zs ++ [z])        (app2)
                        = y:reverse (z:zs)             (rev2R)
```

Mit diesem Hilfsresultat folgt nun leicht:

```
reverse (reverse (x:xs')) = reverse (reverse xs' ++ [x]) (rev2)
                          = x:reverse (reverse xs')      (Hilfsres.)
                          = x:xs'                        (IV)
```

Damit ist die Behauptung gezeigt. □

4.2.3 Satz Für alle endlichen Listen `xs :: [a]` gilt: `reverse xs = rev xs.`

Beweis: Wir zeigen zunächst das folgende

Hilfsresultat: für alle `xs, ys :: [t]` gilt:
`aux xs ys = reverse xs ++ ys.`

Beweis mittels Listeninduktion über `xs`:

Fall `xs = []`:

```
aux  []  ys  =  ys                   (aux1)
             =  [] ++ ys             (app1R)
             =  reverse [] ++ ys     (rev1R)
```

Fall `xs = x:xs'`:

```
aux (x:xs') ys =  aux xs' (x:ys)                  (aux2)
               =  reverse xs' ++  (x:ys)          (IV)
               =  reverse xs' ++  ([x] ++ ys)     (app1R,app2R)
               =  (reverse xs' ++ [x]) ++ ys      (Assoz. von ++)
               =  reverse (x:xs') ++ ys           (rev2R)
```

Damit folgt:

```
rev xs = aux xs [] = reverse xs ++ []  = reverse xs
```

Der Nachweis, dass für endliche Listen `xs` gilt `xs ++ [] = xs` bleibt den Lesern zur Übung. □

4.3 Strukturelle Induktion

Das Prinzip der Listeninduktion lässt sich in analoger Weise auch auf andere algebraische Datenstrukturen übertragen. Man spricht allgemein von *struktureller Induktion*. Wir begnügen uns hier mit der Betrachtung eines Beispiels.
Für Binärbäume mit folgender Deklaration

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

kann folgendes Induktionsprinzip abgeleitet werden:

1. Zeige die zu beweisende Eigenschaft für Bäume der Form `Leaf x`.

2. Zeige, dass aus der Gültigkeit der zu beweisenden Eigenschaft für Bäume `l` und `r` die Eigenschaft für den Baum `Node l r` folgt.

4.3.1 Lemma Mit den Funktionsdefinitionen

```
numLeaves :: Tree a  -> Int
numLeaves (Leaf x)   = 1
numLeaves (Node l r) = numLeaves l + numLeaves r

numNodes :: Tree a  -> Int
numNodes (Leaf x)   =  0
numNodes (Node l r) = 1 + numNodes l + numNodes r
```

folgt, dass für beliebige endliche Bäume `t :: Tree a` gilt:

`numLeaves t = numNodes t + 1.`

Beweis: Strukturelle Induktion über `t`:

Fall `t = Leaf x`:

```
numLeaves (Leaf x) = 1 = 1 + 0 = 1 + numNodes (Leaf x).
```

Fall `t = Node l r`:

```
numLeaves (Node l r) = numLeaves l + numLeaves r
                     = numNodes l + 1 + numNodes r + 1        (IV)
                     = (numNodes l + numNodes r + 1) + 1
                     = numNodes (Node l r) + 1
```

□

Kapitel 5

Interaktive Ein-/Ausgabe

Die Modellierung interaktiver Ein-/Ausgaben *unter Beibehaltung referentieller Transparenz* ist ein schwieriges Problem, denn Ein-/Ausgabeoperationen bewirken „Seiteneffekte“ auf dem Ein-/Ausgabemedium. Ein Programm, das auf Eingaben zur Laufzeit Ausgaben erzeugt, kann auf die gleiche Eingabe eventuell unterschiedlich reagieren. Zum Beispiel gibt ein Programm, das alle bisher eingegebenen Werte aufsummiert, bei doppelter Eingabe einer von Null verschiedenen Zahl unterschiedliche Summenwerte aus.
Ein solches Verhalten scheint im Widerspruch zur Seiteneffektfreiheit und zum Gleichheitsprinzip zu stehen. Allerdings ist zu beachten, dass funktionale Programme natürlich deterministisch sind, d.h. ein Programm erzeugt zu einer gleichen Folge von Eingaben die gleiche Folge von Ausgaben.

Beispiel: Ein erster Ansatz zur interaktiven Ein-/Ausgabe von Werten könnte vordefinierte Funktionen einführen. Sei etwa `Channel` ein neuer Typ mit den Ein-/Ausgabemedia `stdin, stdout, stderr` als Werten. Mit vordefinierten Funktionen

```
readInt   :: Channel -> Int
writeInt  :: Channel -> Int -> Int
```

könnten dann ganze Zahlen ein- und ausgegeben werden. Die Funktion `writeInt` liefert die ausgegebene Zahl außerdem als Ergebnis zurück.

Eine Funktion, die solange ganze Zahlen einliest, aufaddiert und die Zwischensummen ausgibt, bis Null eingegeben wird, könnte wie folgt definiert werden:

```
f :: Int -> Int
f    sum =  let val = readInt stdin
            in  if val == 0 then (writeInt stdout sum)
                else f (writeInt stdout (sum + val))
```

In dieser Funktionsdefinition geht die referentielle Transparenz verloren, denn es gilt beispielsweise nicht mehr unbedingt: `f 0 == f 0`. Das Funktionsergebnis hängt von Ein-/Ausgaben ab. Diese Abhängigkeit ist in der Funktionsanwendung nicht sichtbar.

Abhilfe kann dadurch geschaffen werden, diese impliziten Abhängigkeiten sichtbar zu machen und explizit zu verwalten. Als Typ der Ein-/Ausgabeoperationen erhält man dann:

```
readInt   :: Channel -> (Int,Channel)
writeInt  :: Channel -> Int -> (Int,Channel)
```

Die Funktionsdefinition muss wie folgt angepasst werden:

```
f :: Int -> Channel -> Channel -> Int
f    sum    inChan     outChan
  =  let (val, nextin) = readInt inChan
     in  if val == 0 then fst (writeInt outChan sum)
         else let (nextSum,nextOut) = writeInt outChan (sum+val)
              in   f nextSum nextin nextOut
```

Wie sofort ersichtlich ist, ist eine solche explizite Verwaltung von Ein-/Ausgabemedien sehr aufwendig und auch problematisch. Denn diese Verwaltung obliegt dem Betriebssystem und es ist unklar, wie mehrere Verweise auf diese Medien implementiert werden sollen. Was soll etwa das Ergebnis des folgenden Ausdrucks sein?

```
(f 0 stdin stdout) + (f 0 stdin stdout)
```

◁

Aufgrund der in diesem Beispiel gezeigten Probleme wird die Ein-/Ausgabe in Haskell mit sogenannten *Monaden* realisiert. Monaden stellen einen allgemeinen Mechanismus zur Kapselung von Berechnungen dar.

5.1 Monadische Ein-/Ausgabe

Der Begriff „Monade“ kommt aus der Kategorientheorie. Für den Einsatz von Monaden in der funktionalen Programmierung genügt es, Monaden als spezielle abstrakte Datentypen zu betrachten. In der I/O-Monade sind die Ein-/Ausgabeoperationen die abstrakten Werte. Es gibt primitive Ein-/Ausgabeaktionen und spezielle Operationen zur sequentiellen Komposition von Aktionen. Die verborgene Implementierung des abstrakten Datentyps kann als globaler Systemzustand gesehen werden, der nur mittels der speziell vordefinierten Operationen modifiziert werden kann.
Die I/O-Monade vermittelt zwischen den Werten in funktionalen Programmen und den Aktionen, die Ein-/Ausgabeoperationen charakterisieren. In rein funktionalen Programmen ist die Auswertungsreihenfolge nur durch Datenabhängigkeiten bedingt. Es kann nicht von einer festen Auswertungsreihenfolge ausgegangen werden. Eine solche ist aber für Ein-/Ausgabeoperationen wesentlich, möchte man eine deterministische Programmausführung garantieren. Die wesentliche Idee von I/O-Monaden besteht darin, dass eine sequentielle Ausführungsreihenfolge für Ein-/Ausgabeaktionen spezifiziert wird, die in jeder Implementierung eingehalten werden muss.

Grundlegende Deklarationen für die I/O-Monade sind:

```
data IO a            -- I/O-Aktionen, die einen Wert vom Typ a liefern

return :: a -> IO a  -- Aktion mit Wertrueckgabe
```

Eine Ein-/Ausgabe-Aktion, die einen Wert vom Typ `a` zurückliefert, hat den Typ `IO a`. Falls kein Wert zurückgeliefert wird, wie bei einer Ausgabeaktion, so wählt man für `a` den Einheitstyp `()` (unit), der genau ein Element hat: das leere Tupel `()`.
Die einfachste Ein-/Ausgabe-Aktion `return v :: IO t` liefert den Wert `v :: t` zurück und bewirkt sonst nichts. Die Aktion `return () :: IO ()` entspricht also einer `skip` oder `noop` Operation, bei der nichts geschieht und kein brauchbares Resultat geliefert wird. Mittels der `return`-Funktion werden Werte in Monaden eingebettet. Es ist bemerkenswert, dass es keine Funktion gibt, die den Typ `IO a -> a` hat, d.h. die zu einer Aktion, die einen Wert zurückgibt, diesen Wert liefert. Dies hängt mit der Kapselung von den seiteneffekt-behafteten Aktionen in Monaden zusammen. Der Übergang von der funktionalen zur monadischen Welt ist eine Einbahnstraße: Funktionale Werte können in Monaden eingebettet werden, aber aus Monaden können keine Werte in die funktionale Welt zurück übertragen werden. Man bleibt in einem monadischen Kontext.
Elementare vordefinierte Ein-/Ausgabeoperationen sind

```
putChar     :: Char   -> IO ()
putStr      :: String -> IO ()
getChar     :: IO Char
getLine     :: IO String
```

Die Funktionen `putChar` und `putStr` geben ein einzelnes Zeichen bzw. eine Zeichenkette auf der Standardausgabe aus und liefern das leere Tupel `()` als Resultat zurück. Die Funktionen `getChar` und `getLine` lesen ein Zeichen oder eine durch das Zeichen `'\n' :: Char` (newline) abgeschlossene Zeichenkette von der Standardeingabe und liefern das gelesene Zeichen bzw. die Zeichenkette (ohne das newline-Zeichen) zurück.

Beispiel: Das kanonische „Hello World"-Programm hat in Haskell folgende Form:

```
module Main where
main = putStr "Hello World\n"
```

◁

Wird ein Haskell-Programm compiliert, so wird bei seiner Ausführung die Variable `main` im Modul `Main` ausgewertet. Diese Variable muss den Typ `IO ()` haben. Sie bestimmt das Resultat eines Programms. Der Hugs-Interpreter erlaubt hingegen die Auswertung beliebiger Ausdrücke.

Um mehrere Aktionen nacheinander auszuführen, stellt Haskell eine `do`-*Notation* zur Verfügung. Auf das Schlüsselwort `do` folgt eine Sequenz von Aktionen, die durch die Layout-Regel getrennt werden. Dabei gelten dieselben Festlegungen wie für `let`- und `where`-Konstrukte:

```
do  <action_1>
    ...
    <action_k>
```

Die do-Notation ist nur eine spezielle Schreibweise für Funktionenkompositionen. Dies wird in Kapitel 6 näher erläutert.

Beispiel: Die vordefinierte Funktion putStr kann mit einem do-Ausdruck in einfacher Weise rekursiv definiert werden:

```
putStr        :: [Char] -> IO ()
putStr []     =  return ()
putStr (c:cs) =  do putChar c
                    putStr  cs
```

◁

Werte, die von Ein-/Ausgabeaktionen zurückgeliefert werden, können weiterverarbeitet werden, indem sie benannt werden. Dies geschieht *innerhalb eines* do*-Ausdrucks* durch das Konstrukt

```
<variable> <- <IO a-action>
```

Der Name der Variablen kann in den nachfolgenden Aktionen des do-Konstrukts verwendet werden. Es ist zu beachten, dass es sich bei diesem Konstrukt nicht um eine Wertzuweisung an eine Variable handelt. Insbesondere ist der Wert der Variablen nicht veränderlich.

Beispiel: Die folgende Aktion readWrite liest eine Zeile von der Standardeingabe und schreibt die gelesene Zeile auf die Standardausgabe:

```
readWrite :: IO ()
readWrite =  do line <- getLine
                putStr line
```

Die Variable line hat den Typ String. ◁

In do-Ausdrücken können Fallunterscheidungen mit if_then_else-Ausdrücken angegeben werden. Ist b ein Ausdruck vom Typ Bool und sind e1 und e2 Ausdrücke von einem Typ a, so hat der Ausdruck if b then e1 else e2 den Typ a. Innerhalb von do-Ausdrücken müssen e1 und e2 einen Typ IO a haben.

Beispiel: Die Funktion getLine ist in der prelude-Datei wie folgt definiert:

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

Zunächst wird ein Zeichen mittels `getChar` von der Standardeingabe gelesen und `c` genannt. Die nächste Aktion ist ein Verzweigungsausdruck, in dem getestet wird, ob das gelesene Zeichen `c` dem Zeichen `'\n'` entspricht. Ist dies der Fall, so wird die leere Zeichenkette als Resultat zurückgegeben. Ansonsten wird `getLine` rekursiv aufgerufen, um die restliche Zeile zu lesen. Das Ergebnis wird mit `cs` benannt. In diesem Fall wird als Ergebnis die Liste `(c:cs)` zurückgegeben. ◁

Beispiel: Das zu Beginn des Abschnitts angegebene Programm zum Aufsummieren eingelesener ganzer Zahlen kann nun wie folgt mithilfe von monadischer Ein-/Ausgabe geschrieben werden:

```
-- Einlesen und Aufsummieren ganzer Zahlen
f     :: Int -> IO Int
f sum =  do val <- getInt
            if val == 0 then return sum
               else let newSum = val + sum
                    in  do putStr ("Summe: " ++ show newSum ++ "\n"
                                   ++ "Naechste Zahl?")
                           f newSum

-- Einlesen einer ganzen Zahl
getInt :: IO Int
getInt =  do line <- getLine
             return (toNumber line 0)

-- Umwandlung einer Zeichenkette in eine ganze Zahl
toNumber          :: String -> Int -> Int
toNumber []     s =  s
toNumber (c:cs) s =  toNumber cs (s*10 +
                                  (fromEnum c - fromEnum '0'))

-- Hauptprogramm
main :: IO ()
main =  do putStr ("Erste Zahl?")
           erg <- f 0
           putStr ("\nEndergebnis:" ++ show erg)
```

Die Funktion `toNumber` wandelt ganze Zahlen, die als Zeichenketten gegeben sind, in Objekte vom Typ `Int` um. Sie verwendet hierzu die Funktion `fromEnum :: Enum a => a -> Int`, die aufzählbare Typen aus der Klasse `Enum` (siehe Kapitel 7), zu denen `Char` gehört, aufzählt. Zur Umwandlung von Zeichenketten in ganze Zahlen kann auch die vordefinierte Funktion

```
read :: Read a => String -> a
```

verwendet werden. Der Aufruf `(read line)` bewirkt im obigen Kontext dasselbe wie der Ausdruck `(toNumber line 0)`. ◁

Beachten Sie, dass es nicht möglich ist, Tests auf Gleichheit mit monadischen Ergebnissen auszuführen. Im obigen Beispiel liefert der Ausdruck `f 0 == f 0` die Fehlermeldung „IO Int is not an instance of class `EQ`“, deren Bedeutung in Kapitel 7 geklärt wird.
Für die Ein-/Ausgabe in Dateien stehen die vordefinierten Funktionen

```
writeFile   :: FilePath -> String -> IO ()
appendFile  :: FilePath -> String -> IO ()
readFile    :: FilePath -> IO String
```

zur Verfügung. Dabei ist `FilePath` ein Typsynonym für `String`.

Beispiel: Zum Kopieren von Dateien kann folgende Funktion verwendet werden:

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile from to = do contents <- readFile from
                      writeFile to contents
```

◁

5.2 Graphik

Es gibt eine Reihe von Standard-Bibliotheken für Haskell, aus denen Programmierer Module im Bedarfsfall importieren können. Bisher existiert allerdings keine Standard-Graphik-Bibliothek. Auf der Internetseite
`http://www.haskell.org/libraries/#guigs`
gibt es eine Übersicht über existierende Bibliotheken. Im folgenden verwenden wir die „Hugs Graphics Library“ [1], die einfache zweidimensionale Graphikoperationen sowie Tastatur- und Mauseingaben erlaubt. Die Basisfunktionalität wird über den Import des Moduls `SOEGraphics` bereitgestellt:

```
import SOEGraphics
```

Dieses Modul stellt Funktionen zum Anzeigen und Öffnen von Graphikfenstern sowie zum Anzeigen von Text und einfachen geometrischen Formen in Graphikfenstern bereit, unter anderem die folgenden:

```
type Title   =  String
-- Fenstergroesse in Pixeln
type Size    =  (Int,Int)

-- Durchfuehrung einer Graphikausgabe
runGraphics  :: IO () -> IO ()

-- Oeffnen und Schliessen von Fenstern
```

[1] `http://www.haskell.org/graphics/`

```
openWindow   :: Title -> Size -> IO Window
closeWindow  :: Window -> IO ()

-- Ausgabe einer Graphik in einem Fenster
drawInWindow :: Window -> Graphic -> IO ()

-- Tastatureingabe in einem Fenster
getKey       :: Window -> IO Char
```

Vordefinierte Typen sind `Window` und `Graphic`. Bei der Erzeugung eines Fensters müssen ein Titel und die Fenstergröße in Pixeln angegeben werden. Ein Pixel ist ein Punkt in einer Graphik. Er hat etwa das Ausmass von einem Hunderstel Inch, d.h. etwa 100 Pixel können auf 1 inch, ca. 2,54 cm, aneinandergereiht werden. Ein Fenster der Größe (300,300) hat also etwa das Ausmaß $7,6$ cm$*7,6$ cm.

Beispiel: Ein einfaches Graphikprogramm, das ein Fenster öffnet und bei Betätigung einer Taste wieder schließt, kann wie folgt geschrieben werden:

```
openCloseWindow :: IO ()
openCloseWindow =  runGraphics (
                    do  w <- openWindow "My Window" (300,300)
                        k <- getKey w  -- warte auf Tastatureingabe
                        closeWindow w
                   )
```

◁

Zur Ausgabe einer Graphik in einem Fenster steht die Funktion `drawInWindow` zur Verfügung. Einfache Graphiken können mit den folgenden Funktionen definiert werden:

```
-- Pixelkoordinaten
type Point   =  (Int,Int)

-- Definition einfacher Graphiken
text         :: Point -> String -> Graphic
line         :: Point -> Point  -> Graphic
polyline     :: [Point]         -> Graphic
polygon      :: [Point]         -> Graphic
ellipse      :: Point -> Point  -> Graphic

-- Faerben von Graphiken
data Color   =  Black  | Blue    | Green  | Cyan
              | Red    | Magenta | Yellow | White

withColor    :: Color -> Graphic -> Graphic
```

Graphiken werden in einem Graphikfenster über Pixelkoordinaten platziert. Der Punkt mit den Koordinaten (0,0) befindet sich in der oberen linken Ecke eines Graphikfensters. Bei Zunahme der ersten Koordinate, der x-Koordinate, bewegt sich die Position nach rechts, bei Zunahme der zweiten Koordinate, der y-Koordinate, bewegt sich die Position nach unten.
Eine Zeichenkette kann mit der Funktion `text` in eine Graphik umgewandelt werden. Die Pixelkoordinaten geben die linke obere Ecke des Textes an. Eine Linie `line` wird durch Anfang und Endpunkt definiert, ein Linienzug `polyline` durch eine Liste von Punkten. Ein Polygon `polygon` ist ein geschlossener Linienzug, d.h. Anfangs- und Endpunkt werden miteinander verbunden. Die Funktion `ellipse` erzeugt eine Ellipse, die genau in das durch die beiden Punkte bestimmte Rechteck passt. Der erste Punkt definiert die obere linke Ecke des Rechtecks, der zweite Punkt die untere rechte Ecke. Über die Funktion `withColor` können Graphiken gefärbt werden.

Beispiel: Die folgende Funktion erzeugt ein Fenster, in dem roter Kreis in einem gelben Rechteck gezeichnet und mit der roten Inschrift „Kreis im Rechteck“ versehen wird.

```
bild :: IO ()
bild =  runGraphics (
         do  w <- openWindow "Bild" (400,400)
             drawInWindow w (withColor Yellow
              (polygon [(100,100),(100,300),(300,300),(300,100)]))
             drawInWindow w (withColor Red
              (ellipse (150,150) (250,250)))
             drawInWindow w (withColor Red
              (text (140,190) "Kreis im Rechteck"))
             spaceClose w
             closeWindow w
                    )
```

Damit das Fenster nur bei Betätigung der Leertaste geschlossen wird, wird die Funktion `spaceClose` verwendet:

```
spaceClose   :: Window -> IO ()
spaceClose w =  do  k <- getKey w
                    if k == ' ' then closeWindow w
                                else spaceClose w
```

◁

5.3 Fallstudie: Sierpinski Dreiecke

Mit diesen einfachen Mitteln können bereits anspruchsvollere Graphiken erzeugt werden, etwa einfache Fraktale. Ein Fraktal ist eine mathematische Struktur, die sich

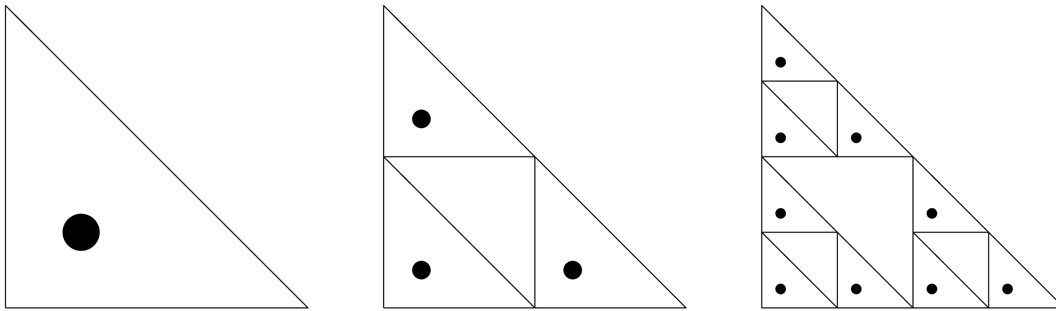


Abbildung 5.1: Konstruktion von Sierpinski Dreiecken

unendlich oft in immer kleineren Dimensionen wiederholt. Das berühmteste Beispiel für Fraktale sind Mandelbrotbäume. Wir betrachten hier eine einfachere Struktur: Sierpinski Dreiecke.
Sierpinski Dreiecke entstehen ausgehend von einem einfachen rechtwinkligen Dreieck mit gleichlangen Katheten durch sukzessives Zerlegen des Dreiecks in drei Dreiecke mit halben Kathetenlängen, also jeweils mit einem Viertel der Fläche (siehe Abbildung 5.1).
Zur graphischen Anzeige von Dreiecken verwenden wir die Funktion `triangle`, die in einem Fenster zu einem Punkt in Pixelkoordinaten und einer Kathetenlänge in Pixelanzahl ein gefülltes Dreieck einer gegebenen Farbe zeichnet:

```
triangle :: Window -> Point -> Int -> Color -> IO ()
triangle w (x,y) size color
    = drawInWindow w (withColor color
            (polygon [(x,y),(x+size,y),(x,y-size)]))
```

Mithilfe dieser Funktion können Sierpinski Dreiecke einer gegebenen Größe und Rekursionstiefe erzeugt werden. Als Größe (Kathetenlänge) empfiehlt sich die Wahl einer Zweierpotenz, damit keine Verzerrungen durch Rundungen auftreten. Als Abbruchgröße, kleinste Dreieckkathetenlänge, werden 8 Pixel gewählt.

```
minSize :: Int
minSize =  8   -- minimale Dreieckgroesse

-- Definition eines Sierpinski Dreiecks
sierpinski :: Window -> Point -> Int -> Color -> IO ()
sierpinski w p size c
    = if size <= minSize then triangle w p size c
      else let size2 = size `div` 2
               (x,y) = p
           in  do   sierpinski w p size2 c
                    sierpinski w (x,y-size2) size2 c
                    sierpinski w (x+size2,y) size2 c

-- Bildschirmausgabe
sierpinskiTriangle = runGraphics (
```

```
do  w <- openWindow "Sierpinski Dreieck" (600,600)
    sierpinski w (50,550) 512 Red
    spaceClose w
)
```

Die Definition der Funktion `sierpinski` ist sehr prägnant. Durch drei rekursive Aufrufe werden die Dreiecke halber Größe an den entsprechend verschobenen Positionen erzeugt.

Kapitel 6

Funktionen höherer Ordnung

Ein Kernkonzept funktionaler Sprachen ist die Gleichstellung von Funktionen und Werten, d.h. Funktionen sind sogenannte *first class citizens* und werden wie andere Basiswerte oder Datenstrukturen behandelt. Insbesondere ist es möglich, dass Funktionen als Argumente oder als Ergebnisse anderer Funktionen auftreten.
Funktionen höherer Ordnung oder *Funktionale* sind Funktionen, die Funktionen als Argumente haben können oder Funktionen als Ergebnis liefern. *Funktionen erster Ordnung* haben dementsprechend nur Basiswerte oder Datenstrukturen als Argumente bzw. Ergebnis.

Beispiel:

1. Ein aus der Mathematik bekanntes Funktional ist die Funktionskomposition, die zwei Funktionen als Argumente nimmt und als Ergebnis die Komposition der beiden Funktionen liefert:

   $$\mathit{compose}\ f\ g \quad = \quad f \circ g \quad \text{mit} \quad (f \circ g\ x) = (f\ (g\ x))$$
   (Haskell-Schreibweise: f.g)

   Das Funktional *compose* bzw. der Haskell-Operator `(.)` haben dementsprechend den polymorphen Typ:

   ```
   (b -> c) -> (a -> b) -> (a -> c)
   ```

   Hier treten als Argumenttypen erstmals Funktionstypen der Form `a->b` auf. Die Klammerung ist wichtig, da der Funktionstypoperator `->` rechtsassoziativ ist.

2. Die Funktion `twice` nimmt zwei Argumente und wendet das erste Argument zweimal auf das zweite Element an:

   ```
   twice :: (a -> a) -> a -> a
   twice f x = f (f x)
   ```

◁

Funktionen höherer Ordnung eröffnen neue Möglichkeiten zur Strukturierung von Problemen und Programmen. Sie erlauben die *Definition allgemeiner Berechnungsschemata*, die durch Funktionsparameter an spezielle Kontexte angepaßt werden können. Auf diese Weise wird ein modularer Programmierstil unterstützt.

Beispiel: Die Definition der `prod`-Funktion (siehe Abschnitt 2.5) folgt einem allgemeinen Rekursionsschema, das als *Divide & Conquer-Schema* bezeichnet werden kann: ein Intervall von Zahlen wird schrittweise halbiert, bis alle Teilintervalle nur noch einzelne Elemente enthalten; dann erfolgt die multiplikative Verknüpfung aller Intervallwerte.

Abstrahiert man von der konkret auf dem Zahlenintervall ausgeführten Verknüpfungsoperation, so erhält man folgende Funktion höherer Ordnung `dc`, die eine binäre Verknüpfungsfunktion als Argument nimmt.

```
dc ::  (Int -> Int -> Int) -> Int -> Int -> Int -> Int
dc g e l h  | l == h  =  h
            | l > h   =  e
            | l < h   =  let m = (l+h) `div` 2
                         in  g (dc g e l m) (dc g e (m+1) h)
```

Der Ausdruck `(g (dc g e l m) (dc g e (m+1) h))` ist eine sogenannte *Applikation höherer Ordnung*, d.h. eine Applikation eines Funktionsparameters, hier `g`, auf Argumentausdrücke. Zur Übersetzungszeit ist nicht bekannt, welche Funktion appliziert wird. Diese Information wird erst dynamisch zur Laufzeit bereitgestellt.

Es gilt

- `prod l h = dc (*) 1 l h`
- `(dc (+) 0 l h)` berechnet $\sum_{i=l}^{h} i$.

◁

6.1 Listenverarbeitung

Insbesondere über Listen und anderen strukturierten Datenobjekten sind Funktionen höherer Ordnung zur Definition allgemeiner Schemata der Verarbeitung einsetzbar. Die im folgenden beschriebenen Listenfunktionen höherer Ordnung gehören zu den in Haskell vordefinierten Standardfunktionen. Man unterscheidet drei Grundtypen der Listenverarbeitung:

6.1.1 Transformation

Häufig wird eine Transformationsfunktion auf alle Listenelemente angewendet, z.B. werden alle Listenelemente verdoppelt, inkrementiert, quadriert, codiert ...

Beispiel: Folgende Funktion quadriert alle Elemente einer Integer-Liste:

```
squareList    :: [Int] -> [Int]
squareList xs =  [ square x | x <- xs ]

square :: Int -> Int
square x = x * x
```

Die Verschlüsselung von Strings nach der Cäsarmethode erfolgt durch eine zyklische Alphabetverschiebung um 13 Zeichen:

```
codeList    :: String -> String
codeList xs =  [ code x | x <- xs ]

code :: Char -> Char
code  x | x >= 'A' && x <= 'Z'  = xox
        | otherwise             = x
   where  ox  = ord x + 13
          xox | ox > ord 'Z' = chr (ox - ord 'Z' + ord 'A' -1)
              | otherwise    = chr ox
```

Dabei bestimmen die vordefinierten Funktionen

`ord :: Char -> Int` und `chr :: Int -> Char`

zu Zeichen den ASCII Code als Integer und umgekehrt.

Vergleicht man die beiden Funktionen `squareList` und `codeList`, so fällt auf, dass beide dasselbe Berechnungsschema beschreiben und zwar einen Durchlauf durch eine Liste. Der einzige Unterschied besteht darin, dass einmal die Funktion `square` und einmal `code` auf alle Elemente der durchlaufenen Liste angewendet wird. Abstrahiert man von der jeweils angewendeten Funktion, so erhält man die im folgenden beschriebene Funktion höherer Ordnung `map`. ◁

Die Funktion `map` nimmt als Argumente eine Funktion und eine Liste und liefert als Ergebnis die Liste, die durch Anwenden der Funktion auf alle Elemente der Argumentliste entsteht:

```
map      :: (a -> b) -> [a] -> [b]
map f xs =  [f x | x <- xs]
```

Es gilt: `squareList l = map square l` und `codeList l = map code l`.

6.1.2 Faltung

Unter der Faltung einer Listen versteht man die Kombination aller Listenelemente mittels einer geeigneten Operation zu einem einzelnen Wert. Typische Listenfaltungen sind das Aufaddieren aller Elemente, die Maximums- oder Längenbestimmung.

Beispiel: Die Funktionen `sumList` und `length` falten eine Liste zu einer ganzen Zahl:

```
sumList        :: [Int] -> Int
sumList []     =  0
sumList (x:xs) =  x + sumList xs

length         :: [a] -> Int
length  []     =  0
length  (x:xs) =  1 + length xs
```

◁

Das diesen Funktionen zugrundeliegende allgemeine Schema wird durch die Funktion `foldr` definiert, die eine Liste unter Verwendung einer binären Argumentfunktion faltet.

```
foldr :: (a -> b -> b) -> b -> [a]    -> b
foldr     f                e    []     =  e
foldr     f                e    (x:xs) =  f x (foldr f e xs)
```

Es gilt:

$$\texttt{foldr}\ op\ a\ [a_1, \ldots, a_n] \overset{*}{\Longrightarrow} a_1\ op\ (a_2\ op\ (\ldots (a_n\ op\ a) \ldots)).$$

Die Funktionen `sumList`, `length` als auch `prod` können mithilfe von `foldr` ohne Rekursion definiert werden:

```
sumList xs = foldr (+) 0 xs
length  xs = foldr plus1 0 xs
             where plus1     :: a -> Int -> Int
                   plus1 x n =  1 + n
prod l h   = foldr (*) 1 [l..h]
```

Faltungen wie etwa die Maximumsbestimmung sind für leere Listen nicht definiert. Es gilt:

```
maxList        :: [Int] -> Int
maxList [x]    =  x
maxList (x:xs) =  max x (maxList xs)
```

Auch dieses Berechnungsmuster tritt häufig auf. Die entsprechende Funktion höherer Ordnung heißt `foldr1`:

```
foldr1       :: (a -> a -> a) -> [a] -> a
foldr1 f xs  =  foldr f x xs
```

Außerdem gibt es eine Variante der Faltungsfunktion, die eine Linksklammerung vornimmt:

```
foldl :: (a -> b -> a) -> a -> [b]    -> a
foldl     f                e    []     =  e
foldl     f                e    (x:xs) =  foldl f (f e x) xs
```

Es gilt:

$$\texttt{foldl}\ op\ a\ [a_1, \ldots, a_n] \overset{*}{\Longrightarrow} (\ldots ((a\ op\ a_1)\ op\ a_2)\ \ldots op\ a_n \ldots).$$

Für assoziative Faltungsoperationen macht es semantisch keinen Unterschied, ob eine Faltung von links oder rechts vorgenommen wird. Es gibt aber Beispiele, bei denen sich Effizienzunterschiede zeigen.

Beispiel: In Abschnitt 4.2 wurde gezeigt, dass die Listenkonkatenation `(++)` assoziativ ist. Für die in der Prelude-Datei wie folgt definierte Funktion `concat` zum Flachklopfen einer Liste von Listen

```
concat     :: [[a]] -> [a]
concat xss =  foldr (++) [] xxs
```

gilt daher, dass die Rechtsfaltung wie auch eine Linksfaltung dasselbe Ergebnis liefern:

```
foldr (++) [] xss = foldl (++) [] xss
```

Allerdings ist der Aufwand der Rechtsfaltung linear in der Länge der Komponentenlisten, während die Linksfaltung zu einem quadratischen Aufwand führt. ◁

Umgekehrt können auch Beispiele gefunden werden, bei denen eine Linksfaltung effizienter ist als eine Rechtsfaltung.

6.1.3 Filtern

Mittels der zweistelligen Funktion `filter` kann man zu einer Liste die Teilliste der Elemente bestimmen, die einem als Parameter übergebenen Prädikat genügen.

```
filter :: (a -> Bool) -> [a]    -> [a]
filter    p              []     =  []
filter    p              (x:xs) =  if (p x) then x: filter p xs
                                   else filter p xs
```

oder mit einer Listenabstraktion:

```
filter p xs = [x | x <- xs, p x]
```

Beispiel: Mit dem ebenfalls in Haskell vordefinierten Prädikat `even`:

```
even n =  n `mod` 2 == 0
```

gilt: `(filter even [1,2,3,4,5,6,7])` $\stackrel{*}{\Longrightarrow}$ `[2,4,6]`. ◁

6.2 Partielle Applikationen und Currying

In funktionalen Sprachen wird zugelassen, dass eine Funktion auf weniger Argumente angewendet wird als zur Auswertung notwendig. Solche sogenannten *partiellen Applikationen* repräsentieren Funktionen, die bei Anwendung auf die restlichen Argumente dasselbe Resultat wie die ursprüngliche Funktion liefern.

Beispiel: Die Additionsfunktion `add` mit `add x y = x+y` ist eine zweistellige Funktion über Zahlen.

In Haskell beschreibt der Ausdruck `(add 5)` eine einstellige Funktion über Zahlen, die zu ihrem Argument 5 hinzuaddiert. Die partielle Applikation `(add 1)` definiert die Nachfolgerfunktion. ◁

Um die Definition von Funktionen durch partielle Applikationen zu ermöglichen, wählt man als Typ einer n-stelligen Funktion

$$\textbf{nicht}\ (t_1, t_2, \ldots, t_n) \to t,\ \textbf{sondern}\ t_1 \to (t_2 \to \ldots (t_n \to t) \ldots).$$

Dies verdeutlicht, dass die n-stellige Funktion als einstellige Funktion mit einem funktionalen Wertebereich betrachtet wird. Diese Vorgehensweise ist unproblematisch, da beide Typen isomorph sind.

Beispiel: Die Funktion `map` hat den Typ `(a->b) -> [a] -> [b]` für alle Typen `a` und `b`, also insbesondere den Typ `(Int->Int) -> [Int] -> [Int]`. Da die Funktion `square` den Typ `Int -> Int` hat, gilt `map square :: [Int] -> [Int]`. Es ist zulässig, die Funktion `squareList` durch die Gleichung `squareList = map square` zu definieren, ohne Angabe des Listenparameters. Analog gilt:
`sumList = foldr (+) 0`. ◁

Auch für Operatoren sind partielle Applikationen erlaubt. Es ist sogar möglich, auf ein beliebiges der beiden Argumente zu applizieren. Man nennt dies *Operatorsektionen.* So ist z.B. `(< 5):: Int -> Bool` ein Funktionsausdruck, der Argumente daraufhin testet, ob sie kleiner als `5` sind, und `(5 <):: Int -> Bool` testet, ob Argumente größer als `5` sind.

Beispiel: Die in Abbildung 1.3 angegebene Definition von Quicksort verwendet die `filter`-Funktion und Operatorsektionen:

```
quicksort        :: Ord a => [a] -> [a]
quicksort []     =  []
quicksort (x:xs) =  quicksort (filter (< x) xs) ++
                    [x] ++ quicksort (filter (x <=) xs)
```

◁

Die Identifikation mehrstelliger Funktionen mit einstelligen Funktionen höherer Ordnung wird nach dem Logiker H.B. Curry *Currying* genannt. In deutschsprachiger Literatur findet man auch die Bezeichnung *Schönfinkeln* nach dem Logiker Schönfinkel, der mit dieser Identifikation bereits vor Curry gearbeitet hat.
In Haskell sind Funktionen vordefiniert, die eine Funktion über Paaren in eine einstellige Funktion mit funktionalem Wertebereich umwandelt und umgekehrt:

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

uncurry     :: (a -> b -> c) -> (a,b) -> c
uncurry f p =  f (fst p) (snd p)
```

Beispiel: Die Addition zweier Listen kann mithilfe der Funktion `map` und folgender Funktion `zip`, die aus zwei Listen eine Liste von Paaren macht, definiert werden:

```
zip              :: [a] -> [b] -> [(a,b)]
zip []    []     =  []
zip (x:xs)(y:ys) =  (x,y) : zip xs ys

addList :: [Int] -> [Int] -> [Int]
addList    l1       l2    =  map (uncurry (+)) (zip l1 l2)
```

Hier wird `uncurry` eingesetzt, um den Operator `(+)` auf Paare anzuwenden. ◁

6.3 λ-Abstraktionen

Oft werden Funktionsargumente nur in einem ganz bestimmten Kontext verwendet, so dass eine separate Funktionsdefinition wenig Sinn macht. λ-Abstraktionen sind Ausdrücke, die „namenlose“ Funktionen definieren. Anstatt eine Funktion durch eine Funktionsdefinition

`f x1 ... xn = e`

mit „Namen“ `f` festzulegen, kann dieselbe Funktion durch eine λ-Abstraktion der Form

`λ x1 ... xn -> e`

angegeben werden. Anstelle des λ wird in Haskell \ geschrieben. Auch bei λ-Abstraktionen gilt Currying, d.h. `λ x1 ... xn -> e` ist eine Kurzschreibweise für

`λ x1 -> ( λ x2 -> ... ( λ xn -> e)...).`

Beispiel: Einfache Listentransformationen können ohne separate Definitionen der Funktionsparameter definiert werden:

```
squareList = map (\ x -> x*x)
incList    = map (\ x -> x+1)
evenList   = filter (\ x -> ((x `mod` 2) == 0))
length     = foldr (\ x n -> n+1) 0
```

◁

In λ-Abstraktionen sind auch Terme (pattern) anstelle von Variablen zugelassen. Dies kann z.B. für Funktionen über Tupeln hilfreich sein, bei denen nur ein Muster für die Argumente in Frage kommt und somit direkt Variablen für die Komponenten in der λ-Abstraktion verwendet werden können. Beispielsweise definiert `\ (x,y) -> (y,x)` eine Funktion, die die Komponenten von Paaren vertauscht.

6.4 Weitere Listenfunktionale

Einige weitere nützliche Funktionale zur Listenverarbeitung sind die folgenden:
`zipWith` kombiniert zwei Listen elementweise mit einem binären Funktionsparameter:

```
zipWith                 :: (a -> b -> c) -> [a]  -> [b]   -> [c]
zipWith f []     []     =  []
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
```

Beispiel: Die Addition zweier Listen kann mit `zipWith` einfacher definiert werden:
`addList = zipWith (+).` ◁

Oft ist es notwendig, Präfixe bzw. Suffixe von Listen zu bestimmen. Ist die erwünschte Länge der Teilliste bekannt, können die ebenfalls vordefinierten Funktionen `take` und `drop` verwendet werden:

```
take          :: Int -> [a] -> [a]
take n []     =  []
take n (x:xs) =  x : take (n-1) xs

drop                       :: Int -> [a] -> [a]
drop n []                  =  []
drop n (x:xs) | n==0       =  xs
              | otherwise =  drop (n-1) xs
```

Beispiel: `fst_half:: [a]->[a]; fst_half l = take ((length l) ‘div‘ 2) l`
◁

Folgende Funktionen höherer Ordnung bestimmen Präfix-/Suffixlisten mithilfe eines Prädikatparameters:

```
takeWhile                        :: (a -> Bool) -> [a] -> [a]
takeWhile p []                   =  []
takeWhile p (x:xs) | p x         =  x : takeWhile p xs
                   | otherwise   =  []

dropWhile                        :: (a -> Bool) -> [a] -> [a]
dropWhile p []                   =  []
dropWhile p (x:xs) | p x         =  dropWhile p xs
                   | otherwise   =  x:xs
```

Solange ein Prädikat `p` erfüllt ist, werden Elemente in die Ergebnisliste aufgenommen bzw. weggelassen.

6.5 Funktionale über algebraischen Datenstrukturen

In gleicher Weise wie auf Listen können auch über benutzerdefinierten Datenstrukturen Funktionen höherer Ordnung definiert werden. Als Beispiel betrachten wir allgemeine Bäume, die durch folgende Deklaration definiert werden können:

```
data Tree a = Node a [Tree a]
```

Die Teilbäume werden als Liste angegeben, da der Verzweigungsgrad beliebig ist.
Baumversionen von `map` und `fold` können wie folgt definiert werden:

```
mapTree                :: (a->b) -> Tree a -> Tree b
mapTree  f (Node x ts) =  Node (f x) (map (mapTree f) ts)

foldTree               :: (a -> [b] -> b) -> Tree a -> b
foldTree f (Node x ts) =  f x (map (foldTree f) ts)
```

Beispiel: Mögliche Anwendungen dieser Funktionale sind folgende Funktionen.
`substitute` ersetzt in einem Baum alle Vorkommen des ersten Argumentes durch das zweite Argument:

```
substitute     :: a -> a -> Tree a -> Tree a
substitute x y =  mapTree (\ z -> if (z==x) then y else z)
```

`sumTree` summiert alle Einträge in einem allgemeinen Baum:

```
sumTree :: Tree Int -> Int
sumTree  = foldTree (flip ((+).sum))
```

Dabei werden folgende in Haskell vordefinierten Funktionen `flip` zum Vertauschen der Argumentreihenfolge und `sum` zur Summation aller Listenelemente verwendet:

```
flip       :: (a -> b -> c) -> b -> a -> c
flip f x y =  f y x

sum :: [Int] -> Int
sum =  foldl (+) 0
```

Es gilt: `sum = sumList`.

Die folgende Beispielrechnung soll die Funktionsweise von `sumTree` verdeutlichen:

```
sumTree (Node 3 [Node 4 [], Node 2 []])
=> foldTree (flip ((+).sum)) (Node 3 [Node 4 [], Node 2 []])
=> (flip ((+).sum)) 3
      (map (foldTree (flip ((+).sum))) [Node 4 [], Node 2 []])
=> ((+).sum)
       (map (foldTree (flip ((+).sum))) [Node 4 [], Node 2 []]) 3
=> sum (map (foldTree (flip ((+).sum))) [Node 4 [], Node 2 []]) + 3
=> sum [foldTree (flip ((+).sum)) (Node 4 []),
        foldTree (flip ((+).sum)) (Node 2 [])] + 3
=> sum [flip ((+).sum) 4 (map (foldTree ((+).sum)) []),
        flip ((+).sum) 2 (map (foldTree ((+).sum)) [])] + 3
=> sum [(+).sum (map (foldTree ((+).sum)) []) 4,
        (+).sum (map (foldTree ((+).sum)) []) 2] + 3
=> sum [sum [] + 4, sum [] + 2] + 3
=> sum [0 + 4, 0 + 2] + 3
=> sum [4,2] + 3
=> 6 + 3
=> 9
```

◁

6.6 Fallstudie: Auswertung von Polynomen

Polynome $p(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$ können nach dem Horner-Schema

$$p(x) = (\ldots((c_n x + c_{n-1})x + c_{n-2})x + \cdots + c_1)x + c_0$$

mit n Additionen und n Multiplikationen ausgewertet werden. Mit $c_n = 0 * x + c_n$ kann die Berechnung von Polynomen als Pipeline von Funktionsanwendungen $\lambda z \rightarrow z * x + c_i$ mit $i = n, \ldots, 0$ beschrieben werden. Das Polynom kann als Liste von Koeffizienten dargestellt werden:

```
type Polynom = [Float]
```

Damit ergibt sich folgende Definition:

```
eval :: Float -> Polynom -> Float
-- eval  x  p  => p(x)

fctList :: Float -> Polynom -> [Float -> Float]
fctList    x      = map (\ c z -> z * x + c)

apply :: (a->b) -> a -> b
apply    f         x  = f x

eval = curry (foldl (flip apply) 0.0 . (uncurry fctlist))
```

Auch hier verdeutlicht eine Beispielauswertung die Arbeitsweise der Funktion `eval`.
$p(x) = 2x + 1$ wird durch die Liste `[2,1]` dargestellt. Es gilt:

```
eval 5 [2,1]  => foldl (flip apply) 0.0 . fctList 5 [2,1]
=> foldl (flip apply) 0.0 (map (\ c z -> z * 5 + c) [2,1])
=> foldl (flip apply) 0.0 [\ z -> z * 5 + 2, \ z -> z * 5 + 1]
=> foldl (flip apply) ((flip apply) 0.0 (\ z -> z * 5 + 2))
                      [\ z -> z * 5 + 1]
=> foldl (flip apply)
    ((flip apply) (\ z -> z * 5 + 2  0.0)  (\ z -> z * 5 + 1))  []
=> flip apply (\ z -> z * 5 + 2  0.0)  (\ z -> z * 5 + 1)
=> (\ z -> z * 5 + 1) (\ z -> z * 5 + 2  0.0)
=> (\ z -> z * 5 + 1) (0.0 * 5 + 2)
=> (\ z -> z * 5 + 1) 2
=> 2 * 5 + 1
=> 11
```

Eine effizientere Version der Funktion `eval` arbeitet ohne explizite Zwischenliste von Funktionen:

```
eval x = foldl (\ z c -> z * x + c) 0.0
```

6.7 Fallstudie: Erstellen eines Indexes

Ein Index gibt zu Schlüsselworten eines Textes die Seiten- oder Zeilennummern ihres Auftretens an. Im folgenden nehmen wir zur Vereinfachung an, dass der Eingabetext eine Zeichenkette vom Typ `String` ist, in dem Zeilen durch das Zeichen `'\n'` (newline) getrennt sind. Schlüsselwörter sind alle Wörter mit mindestens 5 Buchstaben. Es soll ein Index erstellt werden, der alle Schlüsselwörter alphabetisch auflistet und zu jedem Wort alle Nummern der Zeilen angibt, in denen das Wort auftritt. Dabei darf keine Zeilennummer zu einem Wort mehrfach genannt werden.
Der Index kann als Liste dargestellt werden, deren Einträge Paare aus Worten und Listen von Zeilennummern sind. Gesucht ist also eine Funktion

```
makeIndex :: Doc -> [(Word,[Int])]
```

wobei die folgenden Typsynonyme verwendet werden, um zu verdeutlichen, welcher Art die verschiedenen Argumente vom Typ `String` sind:

```
type Doc  = String
type Word = String
type Line = String
```

Beispiel: Der Ausdruck

```
makeIndex "Sonne Mond Sterne\n Sterne am Himmel \n \n klarer Himmel"
```

soll das Ergebnis

```
[(Himmel,[2,4]),(klarer,[4]),(Sonne,[1]),(Sterne,[1,2])]
```

liefern. ◁

Die Funktion `makeIndex` kann als Komposition von Funktionen entwickelt werden, die Teilaufgaben lösen. Eine Analyse der Problemstellung führt beispielsweise zu folgenden Teilproblemen:

1. Zerlegung des Eingabedokumentes in Zeilen:

   ```
   splitup :: Doc -> [Line]
   ```

2. Nummerierung der Zeilen:

   ```
   numLines :: [Line] -> [(Int,Line)]
   ```

3. Zerlegung der Zeilen in Worte, zu denen die Zeilennummer gemerkt wird:

   ```
   allNumWords :: [(Int,Line)] -> [(Int,Word)]
   ```

4. Sortieren der Liste nach alphabetischer Reihenfolge der Wörter:

```
sortList :: [(Int,Word)] -> [(Word,Int)]
```

5. Zusammenfassung der Zeilennummern zu gleichen Wörtern (mit Eliminierung von Duplikaten):

```
amalgamate :: [(Word,Int)] -> [(Word,[Int])]
```

6. Einschränkung der Liste auf Schlüsselwörter:

```
shorten :: [(Word,[Int])] -> [(Word,[Int])]
```

Dieser Entwurf führt unmittelbar zu folgender Festlegung für die Funktion `makeIndex`:

```
makeIndex :: Doc -> [(Word,[Int])]
makeIndex = shorten.amalgamate.sortList.allNumWords.numLines.splitup
```

Oft kann die Angabe der komponierten Funktionen in umgekehrter Reihenfolge verwirrend sein. Es ist einfach, in Haskell selbst einen Kompositionsoperator zu definieren, der die Funktionen in der angegebenen Reihenfolge ausführt:

```
infixl 9 >.>  -- Deklaration eines linksassoziativen Operators >.>
              -- mit Bindungsst"arke 9

(>.>) :: (a -> b) -> (b -> c) -> (a -> c)

g >.> f = f . g
```

Mit diesem Operator kann `makeIndex` wie folgt definiert werden, wobei zur Dokumentation die Typen der Komponentenfunktionen als Kommentar angegeben sind:

```
makeIndex ::   Doc -> [(Word,[Int])]
makeIndex =    splitup       --  Doc             -> [Line]
          >.>  numLines      --  [Line]          -> [(Int,Line)]
          >.>  allNumWords   --  [(Int,Line)]    -> [(Int,Word)]
          >.>  sortList      --  [(Int,Word)]    -> [(Word,Int)]
          >.>  amalgamate    --  [(Word,Int)]    -> [(Word,[Int])]
          >.>  shorten       --  [(Word,[Int])] -> [(Word,[Int])]
```

Nun kann jede einzelne Teilaufgabe für sich betrachtet und gelöst werden:

1. `splitup :: Doc -> [Line]`

Die Eingabezeichenfolge muss in die Teilstrings zwischen newline-Zeichen aufgesplittet werden. Dazu können beispielsweise die Funktionen `takeWhile` und `dropWhile` verwendet werden:

```
splitup [] = []
splitup l  = takeWhile (/='\n') l :
             splitup (dropFirst (dropWhile (/='\n') l))

-- wie tail, aber auf leerer Eingabeliste definiert
dropFirst        :: [a]   -> [a]
dropFirst []     =  []
dropFirst (x:xs) =  xs
```

Hier wird die Eingabeliste allerdings zweimal durchlaufen, einmal mit `takeWhile` und einmal mit `dropWhile`. Effizienter ist die Verwendung der folgenden (ebenfalls in Haskell vordefinierten) Funktion `span`, die eine Liste mithilfe eines Prädikatparameters zerlegt in einen Anfangsteil, der das Prädikat erfüllt, und den Rest ab dem ersten Listenelement, das das Prädikat nicht erfüllt:

```
span                      :: (a -> Bool) -> [a] -> ([a],[a])
                          -- in Haskell vordefiniert
span p []                 =  ([], [])
span p (x:xs) | p x       =  let (xs1,xs2) = span p xs
                             in (x:xs1,xs2)
              | otherwise =  ([],x:xs)

splitup [] = []
splitup l  = l1 : splitup (dropFirst l2)
             where (l1,l2) = span (/= '\n') l
```

2. `numLines :: [Line] -> [(Int,Line)]`

Die Liste von Zeilen muss um Zeilennummern ergänzt werden. Wenn die Zeilenliste `listls` ist, so ist die Liste der Zeilennummern `[1 .. length listls]`. Aus der Liste von Zeilen und der Liste der Zeilennummern kann nun mittels der vordefinierten Funktion `zip` (siehe Abschnitt 5.3) eine Liste von Paaren (Zeilennummer, Zeile) erstellt werden:

```
numLines listls = zip [1 .. length listls] listls
```

3. `allNumWords :: [(Int,Line)] -> [(Int,Word)]`

Die nummerierten Zeilen müssen in eine Liste nummerierter Wörter zerlegt werden. Zunächst wird dieses Problem für eine einzelne Zeile betrachtet:

```
numWords :: (Int,Line) -> [(Int,Word)]
```

Die Zerlegung einer Zeile in Wörter kann durch eine Funktion `splitWords :: Line -> [Word]` vorgenommen werden. Als Trennzeichen von Wörtern werden folgende Zeichen zugelassen:

```
whiteSpace = ['\n','\t',' '] -- newline, tabulator, blank
```

Die Funktion `splitWords` verwendet als Hilfsfunktionen eine Funktion `separateWord` zur Bestimmung des ersten Wortes einer Zeile und der Restzeile mittels `span` und eine Funktion `dropSpace` zur Eliminierung von Trennzeichen am Anfang einer Zeichenkette:

```
separateWord    :: Line  -> (Word, Line)
separateWord ls =  span (not.(‘member‘ whiteSpace)) ls

dropSpace       :: Line  -> Line
dropSpace    ls =  dropWhile (‘member‘ whiteSpace) ls

splitWords      :: Line   -> [Word]
splitWords   cs =  split (dropSpace cs)
         where split      :: Line   -> [Word]
               split []   =  []
               split ls   =  let (w,rs) = separateWord ls
                             in  w : split (dropSpace rs)

member          :: Char -> String -> Bool
member c []     =  False
member c (x:xs  =  (x == c) || member c xs
```

Da die Testfunktion `member` auf das zweite Argument partiell appliziert werden muss, um sie als Testprädikat einzusetzen, wird eine Operatorsektion mit dem Infixoperator `‘member‘` vorgenommen.
Das Hinzufügen der Zeilennummer zu den Wörtern einer Zeile ist mit der Funktion `map` kein Problem:

```
numWords            :: (Int,Line) -> [(Int,Word)]
numWords (nr, line) =  map (\ wd -> (nr,wd)) (splitWords line)
```

Nun muss nur noch die Funktion `numWords` auf alle Zeilen angewendet werden. Dies kann wieder mithilfe von `map` erfolgen. Ergebnis von `map numWords` auf eine Liste von Zeilen ist allerdings eine Liste von Listen von Paaren aus Wörtern und Zeilennummern. Diese Liste von Listen kann mittels `foldr` zu einer Liste flachgeklopft werden:

```
allNumWords :: [(Int,Line)] -> [(Int, Word)]
allNumWords =  foldr (++) [] . map numWords
```

4. `sortList :: [(Int,Word)] -> [(Word,Int)]`

Die Liste der mit Zeilennummern versehenen Wörter muss nach den Wörtern alphabetisch sortiert werden. Mittels `map (\ (x,y) -> (y,x))` können die Wörter in den Paaren anschließend vorangestellt werden. Zum Sortieren kann ein beliebiges Sortierverfahren verwendet werden. Es empfiehlt sich, zunächst eine typunabhängige Version des Sortierverfahrens zu entwickeln, die die Ordnung auf den zu sortierenden Listenelementen als Parameter erhält. Auf dem Typ `(Int,Word)` kann folgende Ordnungsfunktion definiert werden:

```
comparePair                :: (Int,Word) -> (Int,Word) -> Bool
comparePair (n1,w1)(n2,w2) =  (w1 < w2) || (w1 == w2 && n1 <= n2)
```

Auf Zeichenketten ist die lexikographische Ordnung vordefiniert.
Eine Version der Quicksortfunktion, die eine solche Ordnungsfunktion als Parameter bekommt, kann wie folgt definiert werden:

```
genQSort           :: (a -> a -> Bool) -> [a] -> [a]
genQSort cp []     =  []
genQSort cp (x:xs) =  genQSort cp [y | y <- xs, cp y x] ++
                        x: genQSort cp [y | y <- xs, not(cp y x)]
```

Nach diesen Vorbemerkungen kann die gesuchte Funktion `sortList` wie folgt realisiert werden:

```
sortList = map (\ (x,y) -> (y,x)) . genQSort comparePair
```

5. `amalgamate :: [(Word,Int)] -> [(Word,[Int])]`

Die Liste von Wörtern mit Zeilennummern muss in eine Liste von Wörtern mit jeweils einer Liste von aufsteigend sortierten Zeilennummern ohne Duplikate umgewandelt werden.
Dazu ist es sinnvoll zunächst, aus der Liste mit Paaren aus Wörtern und Zahlen eine Liste mit Paaren von Wörtern und einelementigen Zahllisten zu machen:

```
makeLists :: [(Word,Int)] -> [(Word,[Int])]
makeLists =  map (\ (w,n) -> (w,[n]))
```

Nun müssen aufeinanderfolgende Einträge mit gleicher Wortkomponente zusammengefasst werden, indem die Listenanteile konkateniert werden:

```
concatLists                :: [(Word,[Int])] -> [(Word,[Int])]
concatLists []             =  []
concatLists [pair]         =  [pair]
concatLists ((w1,l1):(w2,l2):ps)
                | w1 == w2  =  concatLists ((w1,l1++l2):ps)
                | otherwise =  (w1,l1) : concatLists ((w2,l2):ps)
```

Abschließend sollten noch Duplikate in den sortierten Zeilennummerlisten entfernt werden:

```
removeDups :: [(Word,[Int])] -> [(Word,[Int])]
removeDups =  map (\ (w,l) -> (w, rmDups l))
              where rmDups          :: [Int] -> [Int]
                    rmDups []       =  []
                    rmDups [x]      =  [x]
                    rmDups (x:y:ys)
                        | x==y      = rmDups (y:ys)
                        | otherwise = x : rmDups (y:ys)
```

Damit folgt:

```
amalgamate = makeLists >.> concatLists >.> removeDups
```

6. `shorten :: [(Word,[Int])] -> [(Word,[Int])]`

Es müssen zum Abschluss alle Wörter mit mindestens 5 Buchstaben selektiert werden. Dazu kann die Funktion `filter` eingesetzt werden:

```
shorten = filter (\ (w,l) -> length w >= 5)
```

6.8 Nachweis von Programmeigenschaften

Beweise von Eigenschaften Funktionen höherer Ordnung erfolgen analog zu denen für Funktionen erster Ordnung. Für Funktionen über Listen wird häufig Listeninduktion eingesetzt. Eigenschaften wie die folgenden sind insbesondere im Kontext von Programmtransformationen hilfreich:

1. `map (f.g) = (map f) . (map g)`
2. `filter p . map f = (map f) . (filter (p.f))`
3. `foldr f e (xs++ys) = f (foldr f e xs) (foldr f e ys)`, falls `f` assoziativ ist, d.h. `f (f x y) z = f x (f y z)` für beliebige Elemente `x, y` und `z`, und `e` Einselement von f ist, d.h. `f x e = x = f e x` für beliebige Elemente `x`.

Wir begnügen uns hier mit dem Nachweis von 2. für beliebige endliche Listenargumente: Für die leere Liste gilt:

```
(filter p . map f) [] = filter p (map f []) = filter p [] = []
(map f . filter (p.f)) [] = map f (filter (p.f) []) = map f [] = []
```

Damit folgt die Behauptung im Induktionsanfang.
Für `xs = x : xs'` folgt:

```
(filter p . map f) (x:xs') = filter p (map f (x:xs'))
                           = filter p ((f x) : map f xs')
```

An dieser Stelle ist eine Fallunterscheidung notwendig. Falls `p (f x)` den Wert `True` hat, so gilt:

```
filter p ((f x) : map f xs') = (f x) : filter p (map f xs')
                             = (f x) : (map f (filter (p.f) xs')) -- IV
                             = map f (x:(filter (p.f) xs'))
                             = map f (filter (p.f) (x:xs'))
                             = (map f . filter (p.f)) (x:xs')
```

Für den Fall, dass `p (f x)` den Wert `False` hat, erfolgt der Nachweis analog. Auch falls `p (f x)` nicht definiert sein sollte, gilt die Gleichheit, denn in diesem Fall wären beide Ausdrücke auch nicht definiert.

6.9 Monadische Kompositionsfunktionen

In Abschnitt 5.1 wurde die `do`-Notation verwendet, um mehrere IO-Aktionen nacheinander auszuführen. Hinter der `do`-Notation verbergen sich spezielle Kompositionsoperationen für IO-Aktionen:

```
(>>=) :: IO a -> (a -> IO b) -> IO b  -- Komposition mit  Wertuebergabe
(>>)  :: IO a -> IO b        -> IO b  -- Komposition ohne Wertuebergabe
```

Mittels der Infix-Kompositionsoperationen `>>=` und `>>` können zwei Aktionen hintereinanderausgeführt werden: `return 5 >> return 10` ist eine zusammengesetzte Aktion, die als Ergebnis `10` liefert. Sie entspricht der Aktion

```
do  return 5
    return 10
```

Der von der ersten Aktion gelieferte Wert wird nicht weiterverarbeitet. Möchte man Werte zwischen Aktionen übergeben, muss man den Kompositionsoperator `>>=` verwenden. Dieser erwartet als zweites Argument eine Funktion, die zu dem von der ersten Aktion gelieferten Wert eine Aktion bestimmt, die dann ausgeführt wird. Die Funktion wird meistens durch eine λ-Abstraktion der Form `\ <var> -> <expr>` angegeben. Somit bezeichnet `return 5 >>= \ x -> return (x+10)` eine zusammengesetzte Aktion, die als Ergebnis `15` liefert. Sie entspricht dem `do`-Ausdruck

```
do  x <- return 5
    return (x+10)
```

`\ x -> return (x+10)` ist eine Funktion vom Typ `Int -> IO Int`.
Die `do`-Notation ist somit nichts anderes als eine besser lesbare Schreibweise für Funktionskompositionen.

Kapitel 7

Typen

Den meisten funktionalen Sprachen liegt das sogenannte *Hindley-Milner Typsystem* zugrunde. Das Typsystem wurde unabhängig von Hindley und Milner entwickelt.[1] Es unterscheidet sich von Typsystemen konventioneller Sprachen dadurch,

- dass *(parametrische) Polymorphie* von Funktionen und Datenstrukturen zugelassen wird und
- *Typinferenz entscheidbar* ist, so dass die Typen automatisch abgeleitet werden können.

7.1 Polymorphie

Funktionen sind polymorph, wenn sie für verschiedenartige Objekte, also Objekte mit unterschiedlichen Typen, definiert sind. Man unterscheidet zwischen *parametrischer* und *Ad-hoc-Polymorphie*. Parametrische Polymorphie liegt vor, wenn Funktionen gleichartig auf einer ganzen Klasse von Datenobjekten wirken. So sind die Listenfunktionen `length` zur Bestimmung der Länge einer Liste oder `append` zur Listenverkettung völlig unabhängig vom Typ der Listenelemente und entsprechend haben wir bei der Angabe ihres Typs Typvariablen verwendet. Typisch für die Ad-hoc-Polymorphie sind überladene Operatoren wie z.B. die arithmetischen Funktionen, die für ganze Zahlen und Gleitkommazahlen definiert sind, oder die Gleichheitsfunktion, die für unterschiedliche Typen unterschiedlich definiert werden muss, aber dennoch eine feste Semantik hat. Hier steht dasselbe Symbol für eine Reihe von unterschiedlichen Funktionen, von denen je nach Kontext die passende ausgewählt wird. In objekt-orientierten Sprachen ist das Überladen von Bezeichnern ein fester Bestandteil des Programmierstils. Das Hindley/Milner-Typsystem unterstützt nur die parametrische Polymorphie. In Haskell werden zusätzlich überladene Operationen mithilfe von Typklassen bereitgestellt.

Die einfachsten (parametrisch) polymorphen Beispielfunktionen sind etwa die

- Identitätsfunktion `id` mit `id x = x` oder die

[1]siehe: R. Milner: *A Theory of Type Polymorphism in Program*, Journal of Computer and System Sciences (JCSS) (17(3), 1978, 348–375.
R. Hindley: *The principal type scheme of an object in combinatory logic*, Trans. Amer. Math. Soc. 146, 29–60.

– Projektionsfunktionen `proj2_i` mit `proj2_1 x y = x` und `proj2_2 x y = y`.

Ihre Definition ist unabhängig vom Typ der Argumente. Dies wird in den Typen solcher Funktionen dadurch zum Ausdruck gebracht, dass anstelle konkreter Argumenttypen Typvariablen angegeben werden, die implizit allquantifiziert sind.
Typvariablen werden oft durch kleine griechische Buchstaben, also α, β, γ etc. bezeichnet. In Haskell werden aus typographischen und Lesbarkeitsgründen Bezeichner, die mit einem Kleinbuchstaben beginnen, als Typvariablen verwendet.
Als Typ der Identitätsfunktion ergibt sich $\alpha \to \alpha$ (genau genommen $\forall \alpha . \alpha \to \alpha$), die Projektionsfunktionen haben den Typ $\alpha \to \beta \to \alpha$ bzw. $\alpha \to \beta \to \beta$.

In gleicher Weise sind Datenstrukturen meist unabhängig vom Typ ihrer Komponenten. Der Aufbau von Listen ist unabhängig vom Typ der Listeneinträge. Die Listenkonstruktoren sind polymorph:

– $[\,]$ hat den Typ $[\alpha]$.

– $(\,:\,)$ hat den Typ $\alpha \to [\alpha] \to [\alpha]$

Dabei ist `[.]` ein einstelliger *Typkonstruktor*. Listen mit Wahrheitswerten wird der Typ $[Bool]$ zugeordnet. Listen von Listen von ganzen Zahlen der Typ $[[Int]]$.
Formal kann die Menge aller Typen oder Typausdrücke wie folgt festgelegt werden:

7.1.1 Definition: Sei $TVar = \{\alpha, \beta, \ldots\}$ eine abzählbare Menge von Typvariablen und $\Theta = \bigcup_{n \in \mathbb{N}} \Theta^n$ ein Rangalpabet von Typkonstruktoren. Ein Typkonstruktor $\vartheta \in \Theta^n$ habe die Stelligkeit n. Die Menge Θ^0 der nullstelligen Typkonstruktoren enthalte Menge der Basistypen $T_0 := \{$`Int, Bool, Char,`$\ldots\}$.

Die Menge $Typ_\Theta(TVar)$ der *polymorphen Typen über* Θ *und* $TVar$ ist die kleinste Menge Typ, für die gilt:

1. $TVar \subseteq Typ$ (Typvariablen)
2. mit $t_1, t_2 \in Typ$ ist auch $(t_1 \to t_2) \in Typ$ (Funktionstypen)
3. falls $\vartheta \in \Theta^n$ und $t_1, \ldots, t_n \in Typ$, so gilt auch $(\vartheta\ t_1 \ldots t_n) \in Typ$ (Strukturtypen)

7.2 Typinferenz

Unter Typinferenz versteht man die Bestimmung von Typen zu ungetypten Ausdrücken. Nicht alle Ausdrücke sind typisierbar.

Beispiel: Unter einer Selbstapplikation versteht man die Anwendung eines funktionalen Objektes auf sich selbst. Für die Identitätsfunktion ist die Selbstapplikation zum Beispiel möglich. Im Hindley/Milner-Typsystem ist die Selbstapplikation allerdings nicht typisierbar und damit ausgeschlossen:

```
f x = (x x).
```

Diese Selbstapplikation kann nicht typisiert werden, denn $\alpha \to \beta = \alpha$ definiert keinen endlichen Typ.

◁

Im Hindley-Milner-System ist Typinferenz entscheidbar. Jeder typisierbare Ausdruck besitzt einen (bis auf Typvariablenumbenennung eindeutigen) allgemeinsten Typ, der zur Compilezeit bestimmt werden kann.
Zu den wesentlichen Vorteilen dieses Ansatzes zählen die

- Vermeidung von Laufzeitfehlern und das
- frühzeitige Erkennen von Programmierfehlern, die sich durch Typfehler äußern.

Dies wird auch durch den Slogan:

"Well typed programs do not go wrong" (due to type violations)

zum Ausdruck gebracht.
Die prinzipielle Vorgehensweise bei der Deduktion von Typen in einem funktionalen Programm ist die folgende:

1. Analyse der linken Seiten der Funktionsdefinitionen; Erstellung von Annahmen über die Typen der Funktionen, der Argumente und der Resultate.
2. Analyse der rechten Seiten der Funktionsdefinitionen unter Verwendung der bereits gemachten Annahmen, Erstellung von Kompatibilitätsgleichungen der Art:
   - Der Resultattyp der Funktion muss mit dem Typ des Ausdrucks auf der rechten Seite der Funktionsdefinition übereinstimmen.
   - In jeder Applikation müssen die Funktionen gemäß ihrem Typ verwendet werden.
3. Lösen der Kompatibilitätsgleichungen unter Verwendung des Unifikationsalgorithmus von Robinson[2].

1. Analyse der linken Seiten von Funktionsdefinitionen

Im allgemeinen Fall hat eine Funktionsdefinition folgende Struktur:

$$f \;\; t_1 \;\; t_2 \ldots t_n \quad = \quad e.$$

Der Typ von f hat dementsprechend die allgemeine Form

$$\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \alpha_{n+1}.$$

Die Parameterterme t_i können Variablen oder Konstruktorapplikationen sein. Im Fall einer Konstruktorapplikation kann der Typ der Funktion f präzisiert werden. Denn der Typ von Konstruktoren ist in einer Typdeklaration vordefiniert und kann verwendet werden, um den Typ der entsprechenden Argumentposition der Funktion f genauer anzugeben.

[2] J. A. Robinson: *A Machine Oriented Logic Based on the Resolution Principle*, Journal of the ACM, 12(1), 1965.

Beispiel: Tritt `[]` oder `(x:xs)` als Parameterterm auf, so folgt, dass das entsprechende Argument von f vom Typ $[\alpha]$ sein muss.

Der Parameterterm `[[]]` lässt auf den Typ $[[\alpha]]$ schließen. ◁

Natürlich sind in den verschiedenen definierenden Regeln für ein Funktionssymbol f in den einzelnen Parameterpositionen nur Terme desselben Typs zulässig.

Beispiel: Die Typanalyse der linken Seiten der Funktionsdefinitionen von map

```
map f []     =  []
map f (x:xs) =  (f x) : (map f xs)
```

ergibt den Typ

$$\alpha_1 \rightarrow [\alpha_2] \rightarrow \alpha_3$$

Es werden folgende Annahmen über die Typen der in den Gleichungen auftretenden Bezeichner festgehalten:

Bezeichner	f	x	xs	Rumpf
Typ	α_1	α_2	$[\alpha_2]$	α_3

◁

2. Analyse der rechten Seiten von Funktionsdefinitionen

Die Analyse der Rumpfausdrücke von Funktionsdefinitionen benutzt die bisher erstellten Typannahmen. Für jeden Applikationsausdruck $(e_1\ e_2)$[3] wird eine Gleichung der Form

$$typ(e_1) = typ(e_2) \rightarrow typ((e_1 e_2))$$

aufgestellt. Bereits bekannte Funktionstypen werden dabei sofort berücksichtigt, wobei für jedes Vorkommen von polymorphen Funktionen der Typ mit „neuen“, d.h. bisher noch nicht verwendeten Typvariablen, eingesetzt wird. Für nicht bekannte Typen werden neue Typannahmen gemacht, ebenfalls unter Verwendung neuer Typvariablen. Die konstruierten Gleichungen halten alle Bedingungen fest, die erfüllt sein müssen, damit jede vorkommende Applikation typkonsistent ist.

Beispiel: Die Analyse der rechten Seite der ersten definierenden Gleichung für map ergibt, dass der Resultattyp von map, α_3 laut Annahme bei der Typanalyse der linken Seiten, ein Listentyp sein muss, also

$$\alpha_3 = [\alpha_4].$$

Die Analyse des Rumpfes der zweiten Gleichung ergibt folgende Konsistenzgleichungen:

[3] Currying erlaubt hier die Beschränkung auf einstellige Applikationen. Mehrstellige Applikationen werden als Folge von einstelligen Applikationen aufgefasst.

Ausdruck	Typgleichung
`(f x)`	$\alpha_1 = \alpha_2 \to \alpha_5$
`(map f)`	$\alpha_1 \to [\alpha_2] \to \alpha_3 = \alpha_1 \to \alpha_6$
`((map f) xs)`	$\alpha_6 = [\alpha_2] \to \alpha_7$
`(f x):((map f) xs))`	$\underbrace{\alpha_8 \to [\alpha_8] \to [\alpha_8]}_{\text{Typ von (:) mit neuen Typvariablen}} = \alpha_5 \to \alpha_7 \to \alpha_3$

◁

3. Lösen von Typgleichungen

Die Lösung einer Menge von Typgleichungen:

$$\langle t_i = \tilde{t}_i \mid t_i, \tilde{t}_i \in \mathit{Typ}_\Theta(\mathit{TVar}), 1 \leq i \leq n \rangle$$

ist eine Substitution von Typvariablen durch Typen oder andere Typvariablen, also eine Abbildung

$$\sigma : \mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar}),$$

die alle Gleichungen erfüllt, d.h.

$$\hat{\sigma}(t_i) = \hat{\sigma}(\tilde{t}_i) \text{ für alle } 1 \leq i \leq n,$$

wobei $\hat{\sigma} : \mathit{Typ}_\Theta(\mathit{TVar}) \to \mathit{Typ}_\Theta(\mathit{TVar})$ die homomorphe Fortsetzung von σ auf die Menge $\mathit{Typ}_\Theta(\mathit{TVar})$ bezeichnet:

$$\begin{array}{lcll} \hat{\sigma}(\alpha) & = & \sigma(\alpha) & \text{für } \alpha \in \mathit{TVar} \\ \hat{\sigma}(t_1 \to t_2) & = & \hat{\sigma}(t_1) \to \hat{\sigma}(t_2) & \text{für Funktionstypen} \\ \hat{\sigma}((\vartheta\ t_1 \ldots t_n)) & = & (\vartheta\ \hat{\sigma}(t_1) \ldots \hat{\sigma}(t_n)) & \text{für Strukturtypen.} \end{array}$$

Statt $\hat{\sigma}(t)$ schreibt man meist $t\sigma$. Die Hintereinanderausführung von Substitutionen $\hat{\sigma}_2 \circ \hat{\sigma}_1$ wird durch $\sigma_1\sigma_2$ bezeichnet.

Im allgemeinen existieren für eine Menge von Typgleichungen mehrere Lösungen, sogenannte Unifikatoren, unter denen es allerdings immer eine (bis auf Typvariablenumbenennungen) *allgemeinste Lösung* gibt. Allgemeinste Lösung bedeutet in diesem Zusammenhang, dass sich alle Lösungen durch Komposition der allgemeinsten Lösung und einer weiteren Typsubstitution beschreiben lassen.

7.2.1 Definition Eine Substitution σ heißt *Unifikator* einer Menge von Typgleichungen $\langle t_i = \tilde{t}_i \mid t_i, \tilde{t}_i \in \mathit{Typ}_\Theta(\mathit{TVar}), 1 \leq i \leq n \rangle$, falls

$$t_i\sigma = \tilde{t}_i\sigma \text{ für alle } 1 \leq i \leq n.$$

Ein Unifikator σ heißt allgemeinster Unifikator, falls für jeden weiteren Unifikator $\tilde{\sigma}$ gilt, dass eine Substitution $\rho : \mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})$ existiert mit:

$$\tilde{\sigma} = \sigma\rho = \hat{\rho} \circ \sigma.$$

7.2.2 Satz von Robinson: Für jede unifizierbare Menge von Typgleichungen

$$\langle t_i = \tilde{t}_i \mid t_i, \tilde{t}_i \in \mathit{Typ}_\Theta(\mathit{TVar}), 1 \leq i \leq n \rangle$$

existiert ein (bis auf Umbenennung von Typvariablen) eindeutiger allgemeinster Unifikator.

Die Bestimmung der allgemeinsten Lösung eines Systems von Typgleichungen kann mittels des *Unifikationsalgorithmus von Robinson* erfolgen, der in Logik-Sprachen zur Parameterübergabe verwendet wird. Der Kern des Algorithmus besteht in der Unifikation einer Typgleichung mittels der in Abbildung 7.1 angegebenen Funktion *unify*. Der

$\mathit{unify} : \mathit{Typ}_\Theta(\mathit{TVar}) \times \mathit{Typ}_\Theta(\mathit{TVar}) \to [\mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})] \cup \{\text{fail}\}$
$\mathit{unify}\ (\tau_1, \tau_2)$
$\quad = \textbf{if } \tau_1 \in \mathit{TVar} \wedge \tau_1$ kommt nicht in τ_2 vor (sog. „occur check“) **then** $\{\tau_1 \mapsto \tau_2\}$
$\qquad \textbf{else if } \tau_2 \in \mathit{TVar} \wedge \tau_2$ kommt nicht in τ_1 vor **then** $\{\tau_2 \mapsto \tau_1\}$
$\qquad \textbf{else if } (\tau_1 \in T_0 \vee \tau_2 \in T_0) \wedge \tau_1 = \tau_2 \textbf{ then } \{\}$
$\qquad \textbf{else if } \tau_1 = \tau_{11} \to \tau_{12} \wedge \tau_2 = \tau_{21} \to \tau_{22} \textbf{ then } \mathit{unifyList}([\tau_{11}, \tau_{12}], [\tau_{21}, \tau_{22}], \{\,\})$
$\qquad \textbf{else if } \tau_1 = (\vartheta\ \tau_{11} \ldots \tau_{1n}) \wedge \tau_2 = (\vartheta\ \tau_{21} \ldots \tau_{2n})$
$\qquad\qquad \textbf{then } \mathit{unifyList}([\tau_{11}, \ldots, \tau_{1n}], [\tau_{21}, \ldots, \tau_{2n}], \{\,\})$
$\qquad \textbf{else}$ fail

$\mathit{unifyList} : [\mathit{Typ}_\Theta(\mathit{TVar})] \times [\mathit{Typ}_\Theta(\mathit{TVar})] \times ([\mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})] \cup \{\text{fail}\})$
$\qquad\qquad \to [\mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})] \cup \{\text{fail}\}$
$\mathit{unifyList}\ ([\,], [\,], \sigma) = \sigma$
$\mathit{unifyList}\ ((\tau_1 : ts1), (\tau_2 : ts2), \sigma)$
$\quad = \textbf{let } \sigma_1 = \mathit{unify}(\tau_1\sigma, \tau_2\sigma) \textbf{ in}$
$\qquad \textbf{if } \sigma_1 = \text{fail} \textbf{ then } \text{fail} \textbf{ else } \mathit{unifyList}\ (ts1, ts2, \sigma\sigma_1)$

Abbildung 7.1: Zentrale Funktion des Unifikationsalgorithmus

sogenannte „Occur Check“ garantiert, dass der Unifikationsalgorithmus immer terminiert.

Beispiel: Die Lösung der oben angegebenen Gleichungen unter Verwendung des Unifikationsalgorithmus liefert folgende Substitution

TVar	$\mathit{Typ}(T_0, \mathit{TVar}, \Theta)$
α_1	$\alpha_2 \to \alpha_4$
α_3	$[\alpha_4]$
α_5	α_4
α_6	$[\alpha_2] \to [\alpha_4]$
α_7	$[\alpha_4]$
α_8	α_4

Wendet man diese Substitution auf die für `map` gemachte Typannahme an, so erhält man den Typ:

$$(\alpha_2 \to \alpha_4) \to [\alpha_2] \to [\alpha_4].$$

◁

Dieses Basisschema zur Typinferenz muss natürlich zur Behandlung bedingter Gleichungen, where-Abstraktionen etc. entsprechend erweitert werden. Die folgenden beiden einfachen Beispiele zeigen nochmals die prinzipielle Vorgehensweise bei der Typinferenz.

Beispiel:

1. `twice f x = (f (f x))`
   - Analyse der linken Seite:
     `twice` :: $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3$
     Typannahmen:

Bezeichner	`f`	`x`	„result“
Typ	α_1	α_2	α_3

   - Analyse der rechten Seite:

Ausdruck	Typgleichung
`(f x)`	$\alpha_1 = \alpha_2 \rightarrow \alpha_4$
`(f (f x))`	$\alpha_1 = \alpha_4 \rightarrow \alpha_3$

   - Lösen der Typgleichungen liefert die Substitution

$TVar$	$Typ(T_0, TVar, \Theta)$
α_1	$\alpha_4 \rightarrow \alpha_4$
α_2	α_4
α_3	α_4

   Damit ergibt sich als Typ von `twice`: $(\alpha_4 \rightarrow \alpha_4) \rightarrow \alpha_4 \rightarrow \alpha_4$.

2. `doublehd x = (head (head x))`
   - Analyse der linken Seite:
     `doublehd` :: $\alpha_1 \rightarrow \alpha_2$
     Typannahmen:

Bezeichner	`x`	„result“
Typ	α_1	α_2

   - Analyse der rechten Seite:

Ausdruck	Typgleichung		
`(head x)`	$\underbrace{[\alpha_3] \rightarrow \alpha_3}_{\text{Typ von head mit neuen Typvariablen}}$	$=$	$\alpha_1 \rightarrow \alpha_4$
`(head (head x))`	$\underbrace{[\alpha_5] \rightarrow \alpha_5}_{\text{Typ von head mit neuen Typvariablen}}$	$=$	$\alpha_4 \rightarrow \alpha_2$

   - Lösen der Typgleichungen liefert die Substitution

$TVar$	$Typ(T_0, TVar, \Theta)$
α_1	$[[\alpha_2]]$
α_3	$[\alpha_2]$
α_4	$[\alpha_2]$
α_5	α_2

   Damit ergibt sich als Typ von `doublehd`: $[[\alpha_2]] \rightarrow \alpha_2$.

◁

Bei der Typinferenz von Ausdrücken entfällt Schritt 1 der Typinferenz. Die Schritte 2 und 3 werden aber analog durchgeführt.

Beispiel: Zur Typüberprüfung des Ausdrucks `(map map)` stellt man folgende Gleichung auf:

$$typ(\texttt{map}) = typ(\texttt{map}) \rightarrow typ(\texttt{map map})$$

Dabei werden in den Typannahmen der beiden Vorkommen von `map` unterschiedliche Typvariablen gewählt:

$$(\alpha{\rightarrow}\beta){\rightarrow}[\alpha]{\rightarrow}[\beta] = ((\gamma{\rightarrow}\delta){\rightarrow}[\gamma]{\rightarrow}[\delta]){\rightarrow}typ(\texttt{map map})$$

Der Funktionstyp-Operator $\rightarrow$ ist rechtsassoziativ. Daher nimmt der Unifikationsalgorithmus von Robinson folgende Substitutionen vor

$$\begin{array}{lcl} typ(\texttt{map map}) & \mapsto & [\alpha]{\rightarrow}[\beta] \\ \alpha & \mapsto & \gamma{\rightarrow}\delta \\ \beta & \mapsto & [\gamma]{\rightarrow}[\delta] \end{array}$$

Für `(map map)` ergibt sich somit der Typ $[\gamma{\rightarrow}\delta]{\rightarrow}[[\gamma]{\rightarrow}[\delta]]$. Angewendet auf eine Liste von Funktionen liefert der Ausdruck eine Liste von Listenfunktionen. ◁

Das Hindley/Milner Typsystem ist sehr mächtig und erlaubt die Typisierung der meisten Ausdrücke. Nicht typisierbar ist natürlich die Selbstapplikation, die auf diese Weise ausgeschlossen wird. Das Typsystem lässt allerdings auch einige durchaus sinnvolle Ausdrücke nicht zu.

Beispiel: Ein im Hindley/Milner Typsystem nicht typisierbares Programm:

```
length []       =  0
length (x:xs)   =  1 + length xs

funsum f l1 l2  =  (f l1) + (f l2)

not_typable  =  funsum length [1,2,3] "abc"
```

Problematisch ist in diesem Programm lediglich der Ausdruck `not_typable`.

`length` hat den Typ $[\alpha] \rightarrow Int$,
`funsum` hat den Typ $(\alpha \rightarrow Int) \rightarrow \alpha \rightarrow \alpha \rightarrow Int$.

Obwohl der Ausdruck `not_typable` durchaus sinnvoll ist und in einem ungetypten System ohne weiteres ausgewertet werden könnte, ist eine Typisierung im Hindley/Milner System nicht möglich, denn die Typvariable α im Typ von `funsum` müsste dazu auf zwei verschiedene Weisen (`[Int]` und `[Char]`) instantiiert werden. Obwohl also der funktionale Parameter der Funktion `funsum` eine polymorphe Funktion ist, die auf Listen beliebigen Typs appliziert werden kann, ist die Ausnutzung dieser Polymorphie nicht möglich. ◁

Dieses Beispiel zeigt die wesentliche Beschränkung des Hindley/Milner Typsystems: polymorphe Parameter müssen im Rumpf einer Funktion einheitlich instantiiert werden. Innerhalb von Funktionsrümpfen wird die Polymorphie somit nur in eingeschränkter Weise zugelassen. Diese Einschränkung bereitet allerdings in der Praxis nur selten Probleme, die zudem durch einfache Programmtransformationen umgangen werden können.

Beispiel: Eine einfache Transformation der Funktion funsum zu

```
funsum2 f1 f2 l1 l2 = (f1 l1) + (f2 l2)
```

macht die Typisierung und damit Berechnung des obigen Ausdrucks durch Übergang zu dem Ausdruck

```
typable = funsum2 length length [1,2,3] "abc"
```

möglich. ◁

7.3 Typklassen

Eine Typklasse ist eine Menge von Typen, die durch die Namen und Typen der auf sie anwendbaren Operationen charakterisiert ist. Ein Element einer Typklasse heißt *Instanz* der Typklasse. Die Namen der Operationen sind *überladen*, da sie in allen Instanzen der Typklasse gleich heißen, aber unterschiedliche Definitionen haben können. Durch Typklassen wird in Haskell die sogenannte *Ad-hoc-Polymorphie* bereitgestellt. Haskells Typklassen sind hierarchisch organisiert.

7.3.1 Die Klasse `Eq`

Der Test auf Gleichheit und Ungleichheit mit den Operatoren `==` und `/=` ist eine typische überladene Operation. Es liegt keine parametrische Polymorphie vor, da die Gleichheit nicht auf allen Typen in gleicher Weise definiert werden kann. Auf Funktionen ist die Gleichheit i.a. unentscheidbar und daher nicht zugelassen. Außerdem arbeitet der Gleichheitstest je nach Argumenttyp unterschiedlich.
Die Typen, auf denen die Gleichheit definiert ist, sind in Haskell in der Klasse `Eq` zusammengefasst:

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x /= y  = not (x == y)
```

Ein Typ `a` ist Instanz der Klasse `Eq`, falls es überladene Operationen `(==)` und `(/=)` gibt, wobei `(/=)` als Negation von `(==)` definiert ist.
`(==)` hat den Typ `(Eq a) => a -> a -> Bool`. Dabei bezeichnet `Eq a =>` einen sogenannten *Kontext*, d.h. eine Bedingung im Bezug auf die Typvariable `a`. Im Geltungsbereich des Kontextes dürfen für `a` nur Instanzen der Klasse `Eq` eingesetzt werden.
Welche Typen Instanzen der Typklasse `Eq` sind, wird durch Instanzendeklarationen der Form

```
instance <context> Eq <type> where <definitions>
```

spezifiziert. Für benutzerdefinierte Datenstrukturen kann auf diese Weise festgelegt werden, wie die Gleichheit auf den Datenstrukturen definiert sein soll. Dabei kann bei polymorphen Datenstrukturen eine Kontextangabe notwendig sein.

Beispiel: 1. Für einfache Auszählungstypen kann etwa eine Instanzendeklaration folgender Art vorgenommen werden:

```
data WeTage = Samstag | Sonntag

instance Eq WeTage where
  Samstag == Samstag = True
  Sonntag == Sonntag = True
  Samstag == Sonntag = False
  Sonntag == Samstag = False
```

2. Für rekursive Typen erfolgt die Definition der Gleichheit rekursiv. Bei polymorphen Typen muss zusätzlich die Instanzendeklaration mit einer Kontextangabe für die Typvariablen eingeschränkt werden.

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

instance Eq a => Eq (Tree a) where

(Leaf x)      == (Leaf y)      =   x == y
                                   -- hier ist Eq a erforderlich
(Node l1 r1) == (Node l2 r2) = (l1 == l2) && (r1 == r2)
                                   -- rekursive Definition
t1            == t2            = False
                                   -- bei ungleichen Konstruktoren
```

◁

Da die vollständige Angabe solcher Instanzendeklarationen sehr aufwendig ist, ermöglicht Haskell die automatische Ableitung vieler Standardfunktionen für benutzerdefinierte Datenstrukturen mittels einer `deriving`-Deklaration der Form:

```
deriving (<class1>,<class2>,...,<classk>),
```

die einer Typdeklaration nachgestellt wird. Die `deriving`-Deklaration wurde in Kapitel 3 bereits verwendet, um Instanzen der Typklasse `Show` ableiten zu lassen, die eine Funktion `show :: a -> String` zur Ausgabe von Datenobjekten bereitstellt.

Beispiel: Im obigen Beispiel haben folgende Deklarationen denselben Effekt wie die expliziten Typdeklarationen:

```
data WeTage = Samstag | Sonntag  deriving (Eq)
data Tree a = Leaf a  | Node (Tree a) (Tree a)
              deriving (Eq)
```

◁

Manchmal kann es aber wichtig sein, eine eigene Instanzendeklaration vorzunehmen. Für die in Abschnitt 3.6 beschriebene Implementierung einer Warteschlange mit zwei Listen wurde die Gleichheit nicht wie üblich interpretiert.

Beispiel: Für die Implementierung der Warteschlange aus Abschnitt 3.6 ist folgende explizite Instanzendeklaration erforderlich:

```
data Queue a = Q [a] [a]

instance Eq a => Eq (Queue a) where
 (Q fs1 ls1) == (Q fs2 ls2)  =  (fs1 ++ reverse ls1) ==
                                (fs2 ++ reverse ls2)
```

◁

Im Typ von polymorphen Funktionen, die die Gleichheitsfunktion in ihrer Definition verwenden, muss gegebenenfalls für Typvariablen ein entsprechender Kontext festgelegt werden:

Beispiel: Für die Definition der Funktion `member`, die überprüft, ob ein Objekt Element einer Liste ist, wird die Gleichheitsfunktion verwendet:

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

Daher ergibt sich als Typ für diese Funktion

```
Eq a => a -> [a] -> Bool
```

mit einschränkendem Typkontext `Eq` für die Typvariable `a`. Nur für Listen über Typen aus der Klasse `Eq` ist die Funktion `member` definiert. Die Funktion `member` wird in Haskell als vordefinierte Funktion `elem` bereitgestellt. ◁

Bis auf die Funktionstypen und die monadischen Typen für Ein/Ausgabe sind in Haskell alle vordefinierten Typen Instanzen der Klasse `Eq`. Der Versuch, die Gleichheit von Funktionen oder Ein-/Ausgabeaktionen zu überprüfen, führt dementsprechend zu einer Fehlermeldung.

7.3.2 Die Klasse `Ord`

Zu Klassen können Unterklassen definiert werden, die die Operationen der Oberklasse erben. In der Klassendeklaration wird die Unterklassenbeziehung durch einen Kontext ausgedrückt.
In Haskell ist die Klasse `Ord`, die alle Typen enthält, auf denen Vergleichsoperationen definiert sind, eine Unterklasse der Gleichheitsklasse `Eq`. Sie wird durch die folgende Deklaration definiert. Als Hilfstyp wird `Ordering` verwendet, ein Aufzählungstyp, der als Zieltyp der Vergleichsfunktion `compare :: Ord => a -> a -> Ordering` verwendet wird.

```
data Ordering = LT | EQ | GT deriving (Eq)

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

    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x >= y      = x
              | otherwise   = y
    min x y   | x <= y      = x
              | otherwise   = y
```

Die Definition der Klasse `Ord` enthält eine Reihe von Defaultdefinitionen für die Vergleichsoperatoren und die Funktion `compare`. Dies ermöglicht auf einem selbstdefiniertem Typ entweder durch Festlegung einer Funktion `compare` oder durch Definition von `<=` eine Ordnung zu definieren. Zu beachten ist, dass die in Klassendefinitionen angegebenen Defaultdefinitionen für Funktionen nur gelten, wenn keine explizite Definition in der Instanzdeklaration für einen Typ angegeben wird.

7.3.3 Haskells Klassenhierarchie

Haskell hat eine umfangreiche vordefinierte Klassenhierarchie. Wir begnügen uns hier mit einer stichwortartigen Beschreibung eines Ausschnittes:

Eq: alle vordefinierten Typen bis auf `->` und `IO`; Operationen `(==)` und `(/=)`

Show: alle vordefinierten Typen;
Funktion `show :: a -> String` für die Ausgabe von Werten

Ord: Unterklasse von `Eq`; alle vordefinierten Typen bis auf `->`, `IO` und `IOError`; Vergleichsoperatoren

Num: Unterklasse von `Eq`; `Int, Integer, Float, Double`; arithmetische Funktionen

Der Typ `Integer` enthält ganze Zahlen mit beliebiger Genauigkeit, während `Int` mit einer festen Wortlänge arbeitet.

Real: Unterklasse von `Ord` und `Num`; `Int, Integer, Float, Double`;

Fractional: Unterklasse von `Num`; `Float, Double`;
Gleitkommaoperationen wie `exp, log, sin, cos, tan` u.v.a.

Enum: Unterklasse von `Ord`; `(), Ordering, Bool, Char, Int, Integer, Float, Double`; Aufzählungsoperationen

Integral: Unterklasse von `Num` und `Real`; `Int, Integer`;
ganzzahlige Operationen wie `div, mod` etc.

Kapitel 8

Auswertungsstrategien

In Ausdrücken treten i.a. mehrere reduzierbare Teilausdrücke (Redexe) auf. Eine *Reduktions- oder Auswertungsstrategie* legt fest, in welcher Reihenfolge die Teilausdrücke ausgewertet werden. Da keine Seiteneffekte auftreten, ist die Auswertungsreihenfolge prinzipiell irrelevant (referentielle Transparenz). Terminierende Auswertungen liefern unabhängig von der Reihenfolge der einzelnen Auswertungsschritte immer das gleiche Resultat. Allerdings garantiert nicht jede Auswertungsstrategie die Termination, selbst wenn eine Normalform des Ausdrucks existiert.

Beispiel: Gegeben seien die Funktionsdefinitionen:

```
inf x       = inf x -- inf definiert die nirgendwo
                    -- definierte Funktion
const_fct y = 42    -- const_fct ist eine konstante Funktion
```

Wertet man in dem Ausdruck `(const_fct (inf 0))` zunächst den inneren Redex `(inf 0)` aus, terminiert die Berechnung nicht. Wertet man zuerst den äußeren Redex aus, so erhält man sofort das Ergebnis 42, die eindeutige Normalform des Ausdrucks. ◁

Üblicherweise werden die folgenden *Standardstrategien* zur Redexauswahl unterschieden:

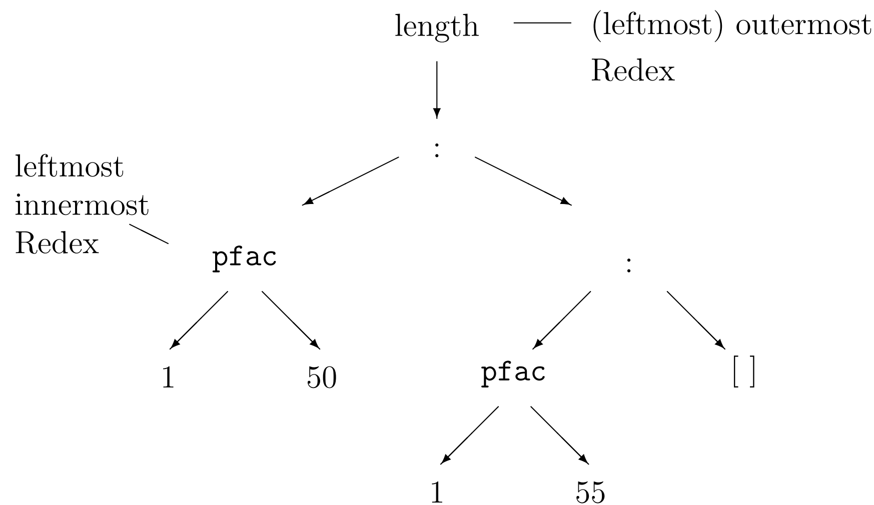
leftmost innermost: Unter den *innersten Redexen*, d.h. denjenigen, die keinen reduzierbaren Teilausdruck enthalten, wird jeweils der *am weitesten links* stehende ausgewertet.

leftmost outermost: Unter den *am weitesten außen* befindlichen Redexen, also reduzierbaren Teilausdrücken, die nicht Teilausdruck eines anderen Redexes sind, wird der am weitesten links stehende ausgewertet.

Beispiel: In dem Ausdruck

$$(\texttt{length}\ ((\texttt{pfac}\ 1\ 50) : ((\texttt{pfac}\ 1\ 55) : [\ ]))$$

werten die Standardstrategien zunächst die gekennzeichneten Redexe aus:



Die *„leftmost outermost"*-Strategie kommt mit sehr viel weniger Auswertungsschritten aus als die *„leftmost innermost"*-Strategie. ◁

Die *„leftmost innermost"*-Auswertungsstrategie (li), die im wesentlichen der „call by value"-Parameterübergabestrategie imperativer Sprachen entspricht, ist einfach zu realisieren, da Funktionsaufrufe immer nur mit bereits ausgewerteten Argumenten evaluiert werden. Von Nachteil ist allerdings, dass immer **alle** Teilausdrücke von Funktionsapplikationen vollständig ausgewertet werden, unabhängig davon, ob ihr Wert zur Bestimmung des Gesamtergebnisses benötigt wird. Man verwendet daher auch die Bezeichnung *„eager evaluation"*. Dies kann insbesondere dazu führen, dass eine Auswertung mit der „leftmost innermost"-Strategie nicht terminiert, obwohl eine Normalform für den betrachteten Ausdruck existiert (siehe oben).

Demgegenüber hat die *„leftmost outermost"*-Strategie (lo) den Vorteil, dass nur Ausdrücke reduziert werden, die zur Berechnung des Gesamtergebnisses benötigt werden (→ lazy evaluation, call by need). Sie beschreibt also eine *bedarfsgesteuerte Auswertung*. Eine effiziente Implementierung dieser Strategie ist allerdings schwierig, da nun Funktionen mit unausgewerteten Argumentausdrücken aufgerufen werden können und somit beliebige unausgewertete Ausdrücke als „Daten" verwaltet werden müssen.

Strenggenommen lässt sich die bedarfsgesteuerte Auswertungsstrategie nur dann als „leftmost outermost"-Strategie charakterisieren, wenn in Funktionsdefinitionen nur Variablenparameter vorkommen bzw. wenn Pattern Matching durch explizite Test- und Selektorfunktionen und bedingte Ausdrücke ausgedrückt wird. Bei Funktionsdefinitionen mit Parametertermen steht der nächste Redex, der zur Fortsetzung der Berechnung ausgewertet werden muss, nicht unbedingt am weitesten links, sondern an der Position, an der in der definierenden Regel ein Term steht. Es handelt sich aber immer um einen äußersten Redex.

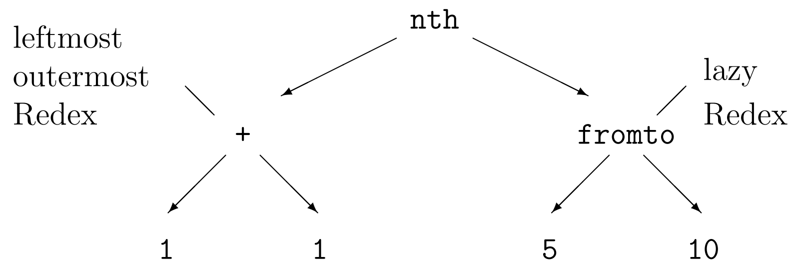
Beispiel: Gegeben seien folgende Funktionsdefinitionen:

```
nth                         :: Int -> [a] -> a
nth n (x:xs) | n==0         =  x
```

```
              | otherwise =  nth (n-1) xs

fromto     :: Int -> Int -> [Int]
fromto n m =  [n..m]
```

In der Funktionsapplikation (nth (1+1) (fromto 5 10)) ist (1+1) der am weitesten links stehende äußerste Redex. Zur Reduktion der äußersten Funktionsapplikation anhand der definierenden Regel für nth ist die Auswertung dieses Redexes allerdings zunächst nicht erforderlich, sondern die Auswertung des Ausdrucks (fromto 5 10).



◁

Die bedarfsgesteuerte Auswertungsstrategie ist immer vollständig, d.h. wenn eine Normalform zu einem Ausdruck existiert, kann sie mit dieser Strategie berechnet werden. Die reine „leftmost outermost"-Strategie ist nur vollständig für Funktionsgleichungssysteme ohne Pattern Matching.

8.1 Behandlung von Pattern Matching

Falls eine Funktion durch mehrere Gleichungen mit Termparametern definiert ist, müssen bei der Auswertung einer Applikation der Funktion nacheinander die verschiedenen Gleichungen überprüft werden. Die aktuellen Funktionsparameter müssen einzeln mit den entsprechenden formalen Termparametern verglichen werden, um festzustellen, ob die Funktionsapplikation eine Instanz der linken Definitionsseite ist. Dabei werden gegebenenfalls Parameter mehrfach untersucht.

Beispiel: Die Längenfunktion für Listen ist bekanntlich wie folgt definiert:

```
length []      = 0
length (x:xs)  = 1 + length xs
```

Bei der Auswertung des Ausdrucks (length [1,2,3]) wird das Argument jeweils mit der leeren Liste und dem Muster (x:xs) verglichen, also bei jedem Reduktionsschritt (bis auf den letzten) zweimal untersucht. Bei der Vertauschung der Definitionen erfolgt beim letzten Reduktionsschritt ein unnötiger Vergleich. ◁

Aus diesem Grund werden Funktionsdefinitionen bei der Compilation von funktionalen Programmen so transformiert, dass Pattern Matching bei der Auswertung der rechten Seite der Definitionen durchgeführt wird. In dem transformierten Programm wird jede Funktion durch genau eine Gleichung mit Variablenparametern definiert. Die Analyse der Funktionsparameter wird durch `case`-Ausdrücke der Form

```
case e of  C1 X1...Xm1  -> e1
           C2 X1...Xm2  -> e2
           ...
           Cr X1...Xmr  -> er
           otherwise    -> e0
```

im Funktionsrumpf definiert. Diese Ausdrücke können geschachtelt werden. Anstelle von Konstruktorapplikationen auf Variablen sind in den Haskell `case`-Ausdrücken als Muster generell beliebige Terme erlaubt. Der `otherwise`-Fall ist optional.

In einfachen einstelligen Funktionen besteht die Pattern-Matching-Compilation im wesentlichen darin, anstelle mehrerer Gleichungen mit unterschiedlichen Termparametern eine Gleichung mit einem `case`-Konstrukt als Rumpf zu verwenden.

Beispiel: Die Definition von `length` kann einfach in

```
length l       = case l of []     -> 0
                           (x:xs) -> 1 + length xs
```

überführt werden. Dies bewirkt bei der Auswertung von Ausdrücken, dass jeweils anhand des obersten Konstruktors des Arguments unmittelbar der korrekte Rumpfausdruck gewählt wird. Unnötige Vergleiche entfallen. ◁

In Funktionen mit mehreren Parametern müssen geschachtelte `case`-Ausdrücke generiert werden. Hierzu wird intern ein Entscheidungsbaum erzeugt, der die Analyse der Parameter optimiert.

Beispiel: Folgende Funktion `subSet` überprüft die Teilmengeneigenschaft von durch Listen dargestellten Mengen.

```
subSet            :: Eq a => [a] -> [a] -> Bool
subSet []      ys =  True
subSet (x:xs)  [] =  False
subSet (x:xs)  ys =  elem x ys && subSet xs ys
```

Die Pattern-Matching-Compilation transformiert diese Definition in folgende Gleichung:

```
subSet l1 l2 = case l1 of []     -> True
                          (x:xs) -> case l2 of [] -> False
                                               ys -> elem x ys &&
                                                     subSet xs ys
```

◁

Geschachtelte Konstruktorterme werden auch in geschachtelte `case`-Ausdrücke transformiert.

Beispiel: Die Funktion `noDups` entfernt aus einer Liste alle aufeinanderfolgenden Duplikate:

```
noDups          :: Eq a => [a] -> [a]
noDups []        =  []
noDups [y]       =  [y]
noDups (y:x:xs) =  if x == y then noDups (x:xs)
                             else y : noDups (x:xs)
```

Mittels des Pattern-Matching-Compilers erhält man folgende äquivalente Definition:

```
noDups l = case l of
             []     -> []
             (y:ys) -> case ys of
                         []     -> [y]
                         (x:xs) -> if x == y then noDups ys
                                             else y:noDups ys
```

◁

Der Pattern-Matching-Compiler erzeugt i.a. flache Muster wie oben angegeben. Der Übergang zu Funktiondefinitionen mit Variablenparametern und `case`-Ausdrücken auf der rechten Seite hat einen Effizienzvorteil, weil jeder aktuelle Parameter jeweils nur noch einmal durchlaufen werden muss, um den passenden Funktionsrumpf zu bestimmen. Außerdem entspricht die „lazy“-Strategie nun der „leftmost outermost“-Strategie und kann damit leichter implementiert werden.

8.2 Auswertungsgrade für Redexe

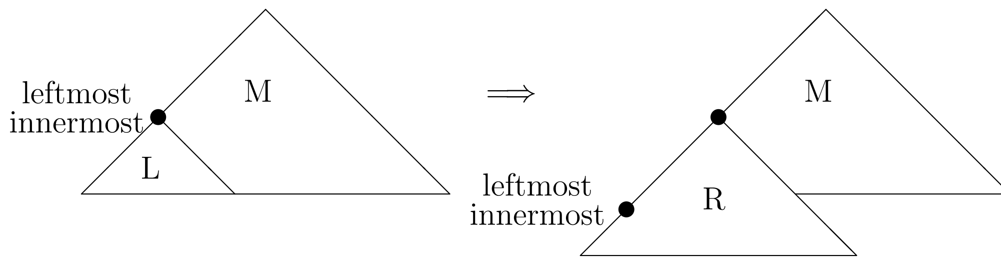
Man unterscheidet zwei verschiedene Auswertungsgrade für Redexe, die in natürlicher Weise den beiden Standardstrategien zugeordnet werden können:

Normalform — „leftmost innermost“-Strategie

Kopfnormalform — „leftmost outermost“-Strategie

Üblicherweise erfolgt die Bestimmung des nächsten Redexes nach *jedem Reduktionsschritt.*

Bei der *„leftmost innermost“*-Strategie entspricht dies der Auswertung von Redexen bis zur *Normalform*, d.h. bis keine Teilredexe mehr vorhanden sind. Denn der „leftmost innermost“-Redex des Teilausdrucks entspricht immer dem „leftmost innermost“-Redex des Gesamtausdrucks:



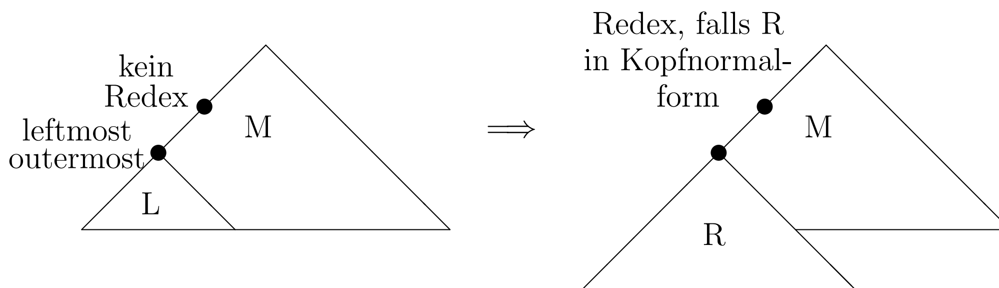
Der „leftmost innermost“-Redex in R entspricht
dem „leftmost innermost“-Redex in M[L/R].

Als Konsequenz braucht man bei Verwendung der „leftmost innermost“-Strategie nicht nach jedem einzelnen Reduktionsschritt, sondern nur nach der vollständigen Auswertung eines Teilredexes die Position des nächsten Redexes zu berechnen.

Bei der „leftmost outermost“-Strategie kann die Auswertung von Teilredexen bis zur sogenannten *Kopfnormalform* vorgenommen werden.

> Ein Ausdruck heißt in *Kopfnormalform*, falls er ein Basiswert, eine Konstruktorapplikation, eine partielle Funktionsapplikation oder eine λ-Abstraktion ist.

Ein Ausdruck in Kopfnormalform ist selbst kein Redex, kann aber Redexe als Teilausdrücke enthalten.
Solange ein „outermost“ Redex nicht zur Kopfnormalform ausgewertet ist, kann aufgrund der speziellen Struktur der Funktionsdefinitionen (Konstruktorterme als formale Parameter) kein äußerer Ausdruck reduzierbar geworden sein.



Beispiel: In der Berechnung

$$\text{(length } \underbrace{\text{(append [] (append [1] [2]))}}_{\text{„leftmost outermost“ Redex}}\text{)}$$

$$\Longrightarrow \text{(length } \underbrace{\text{(append [1] [2])}}_{\text{„leftmost outermost“ Redex}}\text{)}$$

$$\Longrightarrow \underbrace{\text{(length } \underbrace{\text{(1 : (append [] [2]))}}_{\text{Kopfnormalform}}\text{)}}_{\text{„leftmost outermost“ Redex}}$$

wird der äußere Ausdruck erst reduzierbar, wenn der innere Ausdruck zu einer Konstruktorapplikation ausgewertet worden ist. ◁

Dadurch dass Redexe immer nur bis zur Kopfnormalform ausgewertet werden, wird insbesondere die Verarbeitung unendlicher Strukturen möglich. Dies ist der Hintergrund des in den 70er Jahren populären Slogans (Friedman/Wise, Automata, Languages and Programming, Edinburgh Univ. Press 1976):

"Cons" should not evaluate its "arguments".

8.3 Lazy Evaluation

Unter „lazy evaluation" versteht man eine bedarfsgesteuerte Auswertung, die außerdem die Mehrfachauswertung von Teilausdrücken weitgehend vermeidet. Diese Auswertungsstrategie ermöglicht das Arbeiten mit potentiell unendlichen Datenstrukturen sowie die Verarbeitung von partieller Information. In diesem Abschnitt werden Programmiertechniken vorgestellt, die erst durch lazy evaluation möglich werden.

8.3.1 Unendliche Strukturen

Beispiel: In Haskell sind folgende Definitionen möglich:

```
liste1 :: [Int]
liste1 =  1:liste1  -- liste1 definiert die unendl. Liste [1,1,..]

from   :: Int -> [Int]
from n =  n:from (n+1) -- from i def. die unendl. Liste [i,i+1,..]
```

Eine effiziente Berechnung der Fibonacci-Funktion kann mit Hilfe der Definition der unendlichen Liste aller Fibonacci-Zahlen beschrieben werden:

```
fib_nbs :: Int -> Int -> [Int]
fib_nbs    n      m   =  n: fib_nbs m (n+m)
```

Der Ausdruck `(fib_nbs 1 1)` definiert die Liste aller Fibonacci-Zahlen. Durch Selektion entsprechender Elemente aus dieser Liste kann man in einfacher und effizienter Weise Fibonacci-Zahlen bestimmen:

```
fib'  n  =  (nth n (fib_nbs 1 1))
```

Dabei sei `nth` die bereits zuvor definierte Funktion zur Selektion des n-ten Elementes einer nicht-leeren Liste. `nth` dient in der Definition von `fib'` als Kontrollumgebung für die unendliche Liste `fib_nbs`. Wegen der bedarfsgesteuerten Auswertung wird die unendliche Liste nur soweit ausgewertet, wie dies zur Selektion des n-ten Elementes erforderlich ist.

In Haskell kann anstelle der Funktion `nth` auch der Infix-Indexoperator `!!` mit dem Typ `[a]` → `Int` → `a` zur Selektion der `n`-ten Fibonacci-Zahl verwendet werden: `(fib_nbs 1 1)!!n`. ◁

Die Verwendung unendlicher Strukturen ist nur möglich, wenn eine *bedarfsgesteuerte Auswertungsstrategie* (engl. *lazy evaluation*) zugrundegelegt wird. In Ausdrücken werden nur Teilausdrücke ausgewertet, die zur Bestimmung des Gesamtergebnisses unbedingt erforderlich sind.

Beispiel: (Sieb des Eratosthenes)

Das folgende Beispielprogramm beschreibt einen Primzahltest unter Verwendung des sogenannten „Sieb des Eratosthenes“: aus der Folge aller natürlichen Zahlen ≥ 2 werden die Primzahlen durch Elimination (Heraussieben) aller echten Vielfachen bereits gefundener Primzahlen herausgefiltert. Die kleinste verbleibende Zahl ist jeweils die nächste Primzahl.

Ausgangsliste:	2	3	4	5	6	7	8	9	10	11	12	...
	↑	1. Primzahl: 2										
nach dem Herausfiltern echter Vielfacher von 2:	~~2~~	3		5		7		9		11		...
		↑	2. Primzahl: 3									
nach dem Herausfiltern echter Vielfacher von 3:	~~2~~	~~3~~		5		7				11		...
				↑	3. Primzahl: 5							

etc.

Der Primzahltest überprüft, ob die zu testende Zahl in der so bestimmten Liste der Primzahlen auftritt.

```
not_multiple        :: Int -> Int -> Bool
not_multiple x y     =  (y ‘mod‘ x) > 0

sieve               :: [Int] -> [Int]
sieve []            =  []
sieve (x:xs)        =  x: sieve (filter (not_multiple x) xs)

from                :: Int -> [Int]
from n              =  n:from (n+1)

member              :: Ord a => a -> [a] -> Bool
member n []         =  False
member n (x:xs)     =  (n==x) || (n >= x && member n xs)

is_prime            :: Int -> Bool
is_prime n          =  member n (sieve (from 2))
```

Der eigentliche Primzahltest ist durch die Funktion `is_prime` definiert. Der Ausdruck `(sieve (from 2))` bestimmt die unendliche Liste aller Primzahlen. Mit der

Funktion `member` wird überprüft, ob die zu testende Zahl in dieser Liste vorkommt. Dabei wird ausgenutzt, dass die Liste der Primzahlen monoton steigend ist, d.h. die Funktion `member` liefert den Wert `False` zurück, wenn ein Listenelement gefunden wird, das größer als die Zahl `n` ist.

Die Liste der Primzahlen wird durch sukzessives Herausfiltern von Nichtvielfachen bereits gefundener Primzahlen (Funktion `filter`) aus der Liste aller natürlichen Zahlen ≥ 2 (`from 2`) erzeugt. ◁

Dieses Beispiel verdeutlicht den *modularen Programmierstil*, der sich in der Trennung von Daten und Kontrolle bei der Verwendung unendlicher Strukturen äußert. Bei der Bereitstellung der Daten braucht keine künstliche Beschränkung unendlicher Strukturen vorgenommen zu werden. Dies wird auch in dem folgenden Beispiel deutlich.

Beispiel: (Newtonscher Algorithmus)

Die Quadratwurzel $\sqrt{a}$ einer reellen Zahl a mit $a > 0$ kann nach Newton aus einem Anfangwert $x_0 > 0$ mit der Folge

$$x_{i+1} = (x_i + a/x_i)/2$$

approximiert werden. Denn falls die Folge $(x_i)_{i \geq 0}$ für $i \to \infty$ gegen den Grenzwert x konvergiert, so gilt

$$x = (x + a/x)/2 \Leftrightarrow x = \sqrt{a}.$$

Der numerische Test auf Abbruch der Iterationen lautet: Falls $|x_{i+1} - x_i| < \varepsilon$, ist x_{i+1} die Ausgabe.

Das Verfahren besteht demnach aus folgenden Komponenten, die separat entwickelt werden können:

Iterationsfunktion: Abbildung einer Approximation x_i auf die nächste x_{i+1}

```
next :: Float -> Float -> Float
next    a        x     =  (x + a/x) /2
```

Erstellen der Liste aller Approximationen: `next a` ist die Abbildung einer Approximation auf die nächste. Mit $f =$ `next a` ist

$$[x_0, f\ x_0, f\ (f\ x_0), \ldots]$$

die Liste aller Approximationen. Sie kann mit der in Haskell vordefinierten Funktion `iterate` erzeugt werden.

```
iterate :: (a -> a) -> a -> [a]
iterate    f           x =  x : iterate f (f x)
```

Der Ausdruck `iterate (next a)  x0` definiert die obige unendliche Liste, die allerdings nur bis zur Erfüllung des Abbruchkriteriums ausgewertet wird.

Abbruchkriterium: Die Differenz zweier aufeinanderfolgender Folgenglieder ist kleiner als ε.

```
within                    :: Float -> [Float] -> Float
within eps (x1:x2:l)
      | abs (x1-x2) < eps  =  x2
      | otherwise          =  within eps (x2:l)
```

Durch Zusammensetzung der obigen Definitionen ergibt sich damit

```
my_sqrt          :: Float -> Float -> Float -> Float
my_sqrt x0 eps a =  within eps (iterate (next a) x0)
```

Diese Definition ist einfach modifizierbar. Das Abbruchkriterium oder die Iterationsfunktion können unabhängig voneinander geändert werden. ◁

8.3.2 Verarbeitung partieller Informationen

Unter Ausnutzung der bedarfsgesteuerten Auswertung können sehr trickreiche Programme entwickelt werden, die in der Lage sind, *partielle Informationen* zu verarbeiten. Als Beispiel betrachten wir ein Programm, das in einem Durchlauf durch einen Binärbaum alle Einträge durch den minimalen Eintrag ersetzt. Wie ist das möglich? Immerhin wird grundsätzlich sowohl für die Bestimmung des Minimums als auch für die Ersetzung jedes Eintrags ein vollständiger Baumdurchlauf benötigt. Der Trick besteht darin, beim ersten und einzigen Baumdurchlauf jeden Eintrag bereits durch das noch nicht berechnete Minimum zu ersetzen, d.h. durch den Ausdruck, mit dem das Minimum berechnet werden kann.

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

```
repMin   :: Ord a =>  Tree a -> Tree a
repMin t =  minTree
            where (minimum,minTree) = minRep t minimum

-- minRep bestimmt den minimalen Baumeintrag und ersetzt alle
-- Eintraege durch das zweite Argument

minRep                 :: Ord a => (Tree a) -> a -> (a,Tree a)
minRep (Leaf x)      y =  (x, Leaf y)
minRep (Node x l r)  y =  (mini, Node y ly ry)
                          where mini     = min x (min ml mr)
                                (ml, ly) = minRep l y
                                (mr, ry) = minRep r y
```

Die entscheidende Stelle in diesem Programm ist die rekursive Definition des Minimums in

`(minimum,minTree) = minRep t minimum`.

Nur weil `minRep` sein zweites Argument nicht benötigt, um Baumeinträge durch dieses zu ersetzen, funktioniert obiges Programm. Zur Verdeutlichung betrachten wir folgende Beispielauswertung:

```
repMin (Node 5 (Leaf 1) (Leaf 3))
=> minTree
   where (minimum, minTree)
                = minRep (Node 5 (Leaf 1) (Leaf 3)) minimum

minRep (Node 5 (Leaf 1) (Leaf 3)) minimum
=> (mini, Node minimum ly ry)
   where mini     = min 5 (min ml mr)
         (ml, ly) = minRep (Leaf 1) minimum => (1, Leaf minimum)
         (mr, ry) = minRep (Leaf 3) minimum => (3, Leaf minimum)
=> (mini, Node minimum ly ry)
   where mini = min 5 (min 1 3) => min 5 1 => 1
         ly   = Leaf minimum
         ry   = Leaf minimum
=> (1, Node minimum (Leaf minimum) (Leaf minimum))
```

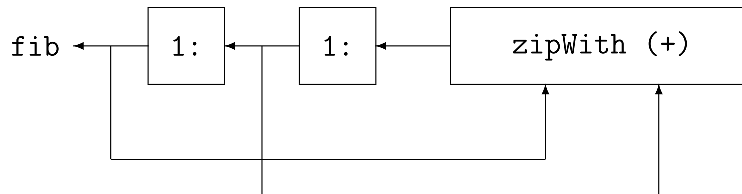
Damit folgt `minimum = 1` und `minTree = Node 1 (Leaf 1) (Leaf 1)`. ◁

8.3.3 Prozessnetze

Der Umgang mit rekursiv definierten unendlichen Listen ist oft unübersichtlich und schwer nachvollziehbar. Hilfreich können graphische Veranschaulichungen des Datenflusses sein. Man nennt diese auch *Prozessnetze*, weil die Knoten des Graphen aktive Komponenten (Funktionsanwendungen) repräsentieren und die Kanten den Datenfluss (Parameter).
Folgende Definition der unendlichen Liste der Fibonacci-Zahlen ist direkt rekursiv.

```
fibs :: [Int]
fibs =  1 : 1 : zipWith (+) fibs (tail fibs)
```

Graphisch kann diese Definition wie folgt veranschaulicht werden:



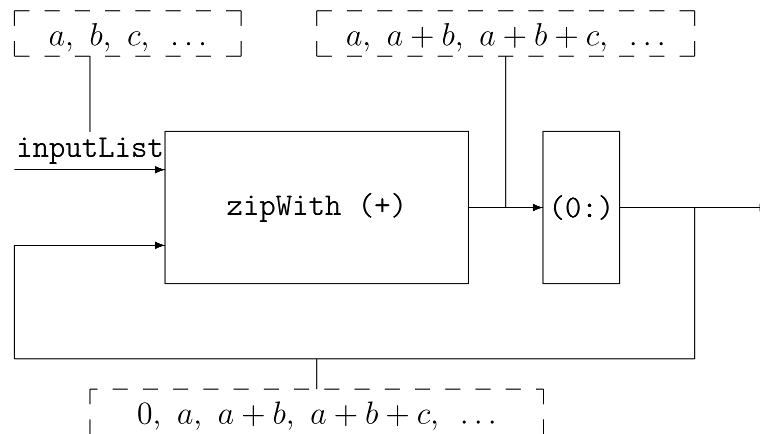
Analog können viele Zahlenreihen in einfacher Weise definiert werden. Zur Bestimmung aller Präfixsummen

$$[0, a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots]$$

einer gegebenen Liste

$$[a_0, a_1, a_2, \ldots]$$

können ebenfalls die bereits ermittelten Summen als Eingabe verwendet werden, um die nächsten Elemente zu berechnen. Zu jedem Element muss lediglich die zuvor bestimmte Summe addiert werden. Dies führt zu folgendem Prozessnetz:



Dieses Netz kann nun wie folgt als Programm geschrieben werden:

```
listSums :: [Int]     -> [Int]
listSums    inputList =  out
                         where out = 0 : zipWith (+) inputList out
```

8.3.4 Strombasierte Ein-/Ausgabe

Die vordefinierte Funktion `interact :: (String -> String) -> IO ()` ermöglicht die strombasierte Ein-/Ausgabe-Programmierung mit Monaden. Sie konvertiert eine Abbildung über Zeichenketten in ein interaktives Programm, das Ausgaben inkrementell erzeugt, sobald entsprechende Eingabedaten zur Verfügung stehen. Ein interaktives Programm kann also einfach als Abbildung über Zeichenketten definiert und mittels `interact` „aufgerufen“ werden.

Beispiel: Ein einfaches Programm, das Zeilen von der Eingabe liest und gespiegelt wieder ausgibt, kann wie folgt definiert werden:

```
readAndReverse :: String -> String
readAndReverse =  lines >.> mapUntil (==[]) reverse >.> unlines

mapUntil            :: (a -> Bool) -> (a->a) -> [a] -> [a]
mapUntil p f []     =  []
mapUntil p f (x:xs)
        | (p x)     =  [f x]
        | otherwise =  (f x) : mapUntil p f xs
```

Die vordefinierte Funktion `lines` zerlegt eine Zeichenkette in Zeilen, d.h. die Teilketten zwischen `\n`-Zeichen, und unlines kehrt eine solche Zerlegung um.

Der Operator `(>.>) :: (a->b) -> (b->c) -> (a->c)` definiert die Funktionskomposition von links nach rechts:

```
infixl 9 >.>
g >.> f = f.g
```

Der Aufruf (`interact readAndReverse) :: IO ()` bewirkt, dass die Eingabe des Benutzers zeilenweise gespiegelt ausgegeben wird. Dasselbe Ein-/Ausgabeverhalten wird mit folgendem monadischen Programm erzielt:

```
readRevWrite :: IO ()  -- einzelne Zeile bearbeiten
readRevWrite =  do  line <- getLine
                    if l == [] then return ()
                               else do  putStr ((reverse l) ++ "\n")
                                        readRevWrite
```

In beiden Fällen wird das Programm durch die Eingabe einer leeren Zeile beendet. ◁

Die strombasierte Programmierung von Ein-/Ausgaben stellt die Verarbeitung der Daten in den Vordergrund. Der genaue Ablauf der interaktiven Verarbeitung bleibt in der Funktion `interact` verborgen. Dagegen beschreibt die monadische Lösung genau den sequentiellen Ablauf der einzelnen Ein/Ausgabeaktionen. Wie die Ausgabedaten von den Eingabedaten abhängen, wird weniger deutlich.
Auch Eingabeaufforderungen und erläuternde Ausgaben können mit `interact` produziert werden. Zur Illustration dient eine Version des Palindromtestprogramms:

Beispiel: Das folgende Programm hat die gleiche Grundstruktur wie das obige Beispielprogramm. Der Aufruf erfolgt mittels `interact test`.

```
palindrome line = (line == reverse line)

test :: String -> String
test =  (lines >.> mapUntil (==[]) process >.> unlines)
        >.> ("Type in a line, please?\n" ++)

process :: String -> String
process line | line == []       = "Good bye\n"
             | palindrome line  = "Is a palindrome!\n"     ++ prompt
             | otherwise        = "Is not a palindrome!\n" ++ prompt
             where prompt = "Type in a line, please?\n"
```

Die Funktion `("Type in a line, please?\n" ++)` entspricht dem äußeren Funktionsaufruf und wird daher als erstes ausgewertet. Dies bewirkt die Ausgabe der ersten Eingabeaufforderung. Anschließend wird die Funktion `process` auf die Eingaben angewendet, bis eine leere Zeile eingegeben wird. ◁

Auch in diesem Beispiel erfolgt im Vergleich zum monadischen Programm eine klarere Beschreibung der eigentlichen Datenverarbeitung.

Zum Abschluss dieses Kapitels sind die Vor- und Nachteile der Standardstrategien in Abbildung 8.1 nochmals zusammengefasst.

Strategie	*Vorteile*	*Nachteile*
leftmost innermost, eager evaluation	einfache Realisierung	Auswertung aller Teilausdrücke, Nichttermination bei Divergenz der Auswertung eines “nicht benötigten Teilausdrucks”
leftmost outermost, lazy evaluation	bedarfsgesteuert, unendliche Datenstrukturen, modularer Programmierstil Prozessnetze	aufwendige Realisierung

Abbildung 8.1: Gegenüberstellung der Standardstrategien

Kapitel 9

Logik-Programmierung

Zu den deklarativen Programmiersprachen gehören neben den funktionalen Sprachen die *Logik-Sprachen*, deren mathematische Grundlage in der Prädikatenlogik 1. Stufe liegt. Im Zentrum der mathematischen Logik stehen Aussagen oder Formeln und deren Gültigkeit. Die Prädikatenlogik ermöglicht insbesondere Aussagen über Objekte und Beziehungen zwischen Objekten, also Relationen bzw. Prädikate zu machen. Logik-Programme bestehen aus einer Menge von speziellen erfüllbaren prädikatenlogischen Formeln. Eine Anfrage an ein Logik-Programm ist ebenfalls eine Formel, zu der festgestellt werden soll, ob es Objekte gibt, für die die Anfrageformel logisch aus dem Programm folgt.

In den letzten 10–15 Jahren konnte gezeigt werden, dass es möglich ist, funktionale und Logik-Sprachen zu integrieren. Dies führte zu den sogenannten *funktional-logischen Programmiersprachen*, die wie funktionale Sprachen aufgebaut sind, aber einen allgemeineren Auswertungsmechanismus verwenden. Auf diese Weise wird es möglich, Logik-Auswertungen im funktionalen Sprachrahmen durchzuführen.
In diesem Kapitel wird eine kurze Einführung in die Logik-Programmierung mit der Sprache Prolog gegeben. Für eine ausführlichere Behandlung der Programmierung in Prolog verweisen wir auf das Lehrbuch

L. Sterling, E. Shapiro: The Art of Prolog, MIT Press 1986.

9.1 Prolog-Programme

Ein Prolog-Programm besteht aus einer Menge von Axiomen, die eine reine Spezifikation von Wissen und Annahmen darstellen. Es enthält i.a. keinerlei Steuerungs- oder Ausführungsanweisungen. Die Axiome sind besondere Formeln der Prädikatenlogik erster Stufe, sogenannte definite Hornklauseln. Man unterscheidet

Fakten: $\underbrace{p}_{\textit{Prädikats-bezeichner}} \underbrace{(t_1, \ldots, t_n)}_{\textit{Parameter-terme}} .$ und

Regeln: $\underbrace{p(t_1, \ldots, t_n)}_{\textit{Regelkopf}} \leftarrow \underbrace{q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_k(t_{k1}, \ldots, t_{kn_k})}_{\textit{Regelrumpf}} .$

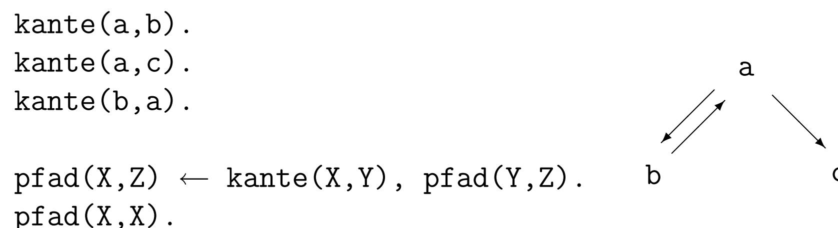
Fakten sind einfache atomare Formeln, d.h. Applikationen von Prädikationssymbolen auf Parameterterme, die wie in funktionalen Sprachen aus Variablen und Konstruktoren aufgebaut sind. Atomare Formeln oder negierte atomare Formeln werden auch *Literale* genannt.
Regeln sind spezielle Implikationen, deren Prämisse oder Regelrumpf eine Konjunktion von atomaren Formeln ist. Die Konklusion oder der Regelkopf besteht wie ein Fakt aus einer atomaren Formel. In Prolog wird anstelle des Implikationszeichens $\leftarrow$ das Doppelzeichen `:-` verwendet. Alle in Fakten und Regeln vorkommenden Variablen sind implizit allquantifiziert, d. h. die durch sie formulierten Aussagen gelten für alle möglichen Belegungen der Variablen mit Werten.

Anfragen an ein Logikprogramm haben die Form einer Regel mit leerem Regelkopf, d.h. die Konklusion der Implikation ist der Wahrheitswert „falsch“. Die Bedeutung einer Anfrage besteht darin festzustellen, ob der Rumpf der Anfrage eine logische Folgerung der Axiome des Logik-Programms ist.

Anfragen: $\leftarrow q_1(t_{11}, \ldots, t_{1n_1}), \ldots, q_k(t_{k1}, \ldots, t_{kn_k})$.

In Prolog wird anstelle des Implikationspfeils in Anfragen das Doppelzeichen `?-` verwendet.

Beispiel: Das folgende Programm P_{graph} definiert einen Graphen mit den Knoten `a`, `b` und `c`:



In Form von Fakten wird zunächst explizit angegeben, welche Kanten in dem Graphen existieren. Das binäre Prädikatssymbol `kante` bezeichnet die Kantenrelation des Graphen.

Des weiteren wird eine Pfadrelation `pfad` durch eine Regel und einen Fakt festgelegt. Die Regel besagt, dass für alle `X, Y` und `Z` ein Pfad von `X` nach `Z` führt, falls eine Kante von `X` nach `Y` existiert und ein Pfad von `Y` nach `Z` führt. Der Fakt drückt aus, dass für alle `X` ein (leerer) Pfad von `X` nach `X` führt.

Eine Anfrage an dieses Programm könnte zum Beispiel lauten

$$\leftarrow \texttt{pfad(U,b).}$$

Diese Anfrage bedeutet: „Von welchem Knoten `U` gibt es einen Pfad nach b?“

`pfad` und `kante` sind binäre Prädikatssymbole, `a`, `b` und `c` bezeichnen Konstanten, d. h. nullstellige Konstruktorsymbole. `X, Y, Z` und `U` sind Variablen. ◁

Funktionen werden in Prolog-Programmen relational, d. h. durch ihren Graphen, spezifiziert. Wie wir später noch sehen werden, hat diese Darstellung den Vorteil, dass neben der Funktion auch die Umkehrfunktionen bzw. -relationen berechnet werden können. Der größte Nachteil besteht allerdings darin, dass die Kenntnis der Funktionseigenschaft verlorengeht.

Beispiel: Verwendet man den nullstelligen Konstruktor 0 zur Repräsentation der Zahl 0 und den einstelligen Konstruktor s zur Repräsentation der Nachfolgerfunktion, so können natürliche Zahlen n durch Konstruktorterme der Gestalt

$$\underbrace{s(s(\ldots s(}_{\text{n mal}}0)\ldots))$$

dargestellt werden.

Mit diesen Vereinbarungen wird die Additionsfunktion durch folgende Regeln definiert:

```
add(X,0,X).
add(X,s(Y),s(Z)) ← add(X,Y,Z).
```

Der Fakt besagt, dass die Addition von `X` und `0` für alle `X` wieder `X` ergibt. Die Regel drückt aus, dass für alle `X, Y` und `Z` die Summe von `X` und `s(Y)` durch `s(Z)` gegeben ist, falls `Z` der Summe von `X` und `Y` entspricht. ◁

9.2 Auswertungsmechanismus

Die *prozedurale oder operationelle Semantik von Prolog-Programmen* definiert, auf welche Weise zu einer Anfrageformel festgestellt werden kann, ob es eine Variablenbelegung gibt, so dass die Formel eine logische Folgerung aus dem Programm darstellt. Eine Berechnung kann als ein konstruktiver Beweis der Zielaussage (Anfrageformel) aus den Axiomen des Programms interpretiert werden.
Die Regeln des Prolog-Programms werden in der prozeduralen Semantik als Ersetzungsregeln verwendet. Ein Anfrageliteral, das mit einem Regelkopf durch eine geeignete Belegung der Variablen unifiziert werden kann, wird durch den Rumpf der Regel ersetzt, in dem die Variablen entsprechend der vorgenommenen Unifikation substituiert werden. Dieser Ersetzungsschritt wird *Resolution* genannt.
Führt eine Folge von Resolutionsschritten dazu, dass alle Anfrageliterale abgearbeitet werden können und eine „leere“ Folge von Anfrageliteralen erreicht wird, so liegt eine erfolgreiche Berechnung vor. Die Komposition der während der Unifikationen durchgeführten Variablensubstitutionen liefert die *Antwortsubstitution*. Wendet man diese auf die Konjunktion der ursprünglichen Anfrageliterale an, so erhält man eine Formel, die eine logische Folgerung aus dem Prolog-Programm ist.
Jeder Berechnungsschritt ist wie folgt organisiert:

Literalauswahl: Es wird ein Literal b_i aus der Anfrage $\leftarrow b_1, \ldots, b_k$ ausgewählt.

Regelauswahl: Es wird eine (variablendisjunkte Variante einer) Programmregel

$$a \leftarrow c_1, \ldots, c_m$$

bestimmt, deren Kopf a mit b_i unifiziert werden kann. Fakten werden in diesem Kontext als Programmregeln mit leerem Rumpf betrachtet.

Unifikation: Bestimmung des allgemeinsten Unifikators *sub* von a und b_i unter Verwendung des Unifikationsalgorithmus von Robinson (siehe Abschnitt 6.2).

Resolutionsschritt: Die Anfrage $\leftarrow b_1, \ldots, b_k$ wird durch die Anfrage

$$\leftarrow (b_1, \ldots, b_{i-1}, c_1, \ldots, c_m, b_{i+1}, \ldots, b_k)\ sub$$

ersetzt.

Man notiert einen solchen Berechnungsschritt in der Form

$$(\leftarrow b_1, \ldots, b_k) \vdash_{sub} (\leftarrow b_1, \ldots, b_{i-1}, c_1, \ldots, c_m, b_{i+1}, \ldots, b_k)\ sub$$

Beispiel: Das Graph-Programm erlaubt zu der Anfrage $\leftarrow$ `pfad (U,a).` etwa die folgende Reihe von Resolutionsschritten:

$\leftarrow$ `pfad (U, a).`

$\vdash_{\{X/U,Z/a\}}$	$\leftarrow$	$\mathtt{kante}(\mathtt{U},\mathtt{Y}), \mathtt{pfad}(\mathtt{Y},\mathtt{a}).$	Resolution mit der `pfad`-Regel
$\vdash_{\{U/a,Y/b\}}$	$\leftarrow$	$\mathtt{pfad}(\mathtt{b},\mathtt{a}).$	Resolution mit `kante(a,b)`
$\vdash_{\{X'/b',Z'/a\}}$	$\leftarrow$	$\mathtt{kante}(\mathtt{b},\mathtt{Y}'), \mathtt{pfad}(\mathtt{Y}',\mathtt{a}).$	Resolution mit der `pfad`-Regel
$\vdash_{\{Y'/a\}}$	$\leftarrow$	$\mathtt{pfad}(\mathtt{a},\mathtt{a}).$	Resolution mit `kante(b,a)`
$\vdash_{\{X''/a\}}$	$\leftarrow$		Resolution mit `pfad(X,X)`

Ausgabe : $\mathtt{U} = \mathtt{a}$

Durch Wahl anderer Regeln zur Resolution ist folgende alternative Berechnung möglich:

$\leftarrow$ `pfad (U, a).`

$\vdash_{\{U/X,Z/a\}}$	$\leftarrow \mathtt{kante}(\mathtt{X},\mathtt{Y}),$	$\mathtt{pfad}(\mathtt{Y},\mathtt{a}).$	Resolution mit der `pfad`-Regel
$\vdash_{\{X/b,Y/a\}}$	$\leftarrow$	$\mathtt{pfad}(\mathtt{a},\mathtt{a}).$	Resolution mit `kante(b,a)`
$\vdash_{\{X'/a\}}$	$\leftarrow .$		Resolution mit `pfad(X,X)`

Hier erhält man als Ausgabe `U = b`. ◁

Die prozedurale Semantik von Prolog-Programmen ist auf zwei Arten inhärent nichtdeterministisch:

Regelauswahlnichtdeterminismus: Auswahl der Regel zur Resolution
Ein Anfrageliteral kann mit den Köpfen mehrerer Regeln unifizierbar sein, so dass alternative Resolutionsschritte möglich sind.

Literalauswahlnichtdeterminismus: Auswahl des Anfrageliterals
Eine Anfrage besteht im allgemeinen aus einer Konjunktion von Literalen, so dass verschiedene Literale für den nächsten Resolutionsschritt gewählt werden können.

Im Hinblick auf parallele Implementierungen lassen sich die beiden Nichtdeterminismusarten als implizite Parallelität ausnutzen. In *UND-parallelen* Implementierungen werden Berechnungen durch gleichzeitige Bearbeitung von Anfrageliteralen parallelisiert (Literalauswahlnichtdeterminismus), wohingegen bei *ODER-parallelen* Implementierungen gleichzeitig Resolutionsschritte mit alternativen Programmklauseln durchgeführt werden (Regelauswahlnichtdeterminismus).

Der Literalauswahlnichtdeterminismus ist bei vollständiger Behandlung des Regelauswahlnichtdeterminismus unwesentlich, d.h. solange man alle Alternativen bei der Regelauswahl berücksichtigt, erhält man bei einer festen Literalauswahlregel immer alle möglichen Lösungen. Meistens werden die Literale einer Anfrage daher von links nach rechts abgearbeitet.

Bei festgelegter Literalauswahlreihenfolge kann man alle Berechnungen systematisch in einem sogenannten *Berechnungs-* oder *Beweisbaum* darstellen. Die Knoten dieses Baumes sind mit den Anfragen beschriftet, die Kanten mit den Substitutionen. Der Berechnungsbaum zeigt in seinen Verzweigungen den Regelauswahlnichtdeterminismus an. Blätter, die mit der leeren Anfrage beschriftet sind, repräsentieren erfolgreiche Berechnungen.
Im Berechnungsbaum kann man drei verschiedene Arten von Berechnungspfaden unterscheiden:

- erfolgreiche Berechnungen (Erfolg, engl. success):

  Herleitung der leeren Anfrage $\leftarrow .$, Ausgabe der Antwortsubstitution

- scheiternde Berechnungen (Fehlschlag, engl. failure):

  Herleitung einer Anfrage, in der keines der Literale mit einem Regelkopf des Logik-Programms unifizierbar ist

- unendliche Berechnungen:

  nichtabbrechende Folge von Resolutionsschritten

Aus Effizienzgründen verwendet Prolog als Auswertungsstrategie eine Tiefensuche im Berechnungsbaum, obwohl diese beim Vorhandensein von unendlichen Berechnungspfaden unvollständig ist, d.h. nicht immer alle möglichen Antworten auf eine Anfrage generiert. Die Literale einer Anfrage werden von links nach rechts, die Programmregeln entsprechend ihrer im Programm angegebenen Reihenfolge abgearbeitet. Durch eine geschickte Anordnung der Literale in Anfragen und Regelrümpfen und der Klauseln im Programmtext hat der Programmierer die Möglichkeit, die Effizienz seines Programms zu steuern.
Der Berechnungsbaum wird nach dem Prinzip des rekursiven Abstiegs von links nach rechts durchlaufen. Für jedes Anfrageliteral wird versucht, einen Fakt oder den Kopf einer Regel zu finden, mit dem es unifiziert werden kann. Unter Berücksichtigung der

vorgenommenen Variablenbelegungen werden die weiteren Literale bzw. der Rumpf der Regel analog weiterbearbeitet. Kann kein Fakt und kein Regelkopf gefunden werden, der mit einem Literal unifizierbar ist, so erfolgt *Backtracking*: man geht zu dem zuletzt unifizierten Literal zurück, hebt die vorgenommenen Bindungen auf und sucht nach alternativen Fakten oder Regelköpfen, mit denen unifiziert werden kann. Der Mechanismus der Unifikation und Resolution bzw. des Backtrackings und Auflösen von Variablenbindungen wird wiederholt, bis entweder eine Variablenbelegung gefunden ist, die die Anfrage aus dem Prologprogramm folgen lässt, oder bis keine Alternative mehr existiert.

9.3 Strukturen und logische Variablen

Strukturen sind in Logik-Programmen wie die algebraischen Datenstrukturen in funktionalen Programmen aufgebaut. Allerdings existiert in Prolog kein strenges Typkonzept. Es wird lediglich die Stelligkeit von Konstruktor- und Prädikatssymbolen berücksichtigt.
Ein wichtiger Spezialfall von Strukturen sind Listen. *Listen* beinhalten beliebige Objekte in geordneter Reihenfolge. Eine Liste ist wie in funktionalen Sprachen entweder die leere Liste `[]` oder eine Struktur der Form

.(*Kopf*, *Restliste*) = (Baum mit Wurzel „.“)

Listenelement / Kopf / *head* — Restliste / *tail* („.“ ist ein zweistelliger Konstruktor)

Für Listen wird in Logik-Sprachen die folgende vereinfachte Notation verwendet:

$[X \mid L]$	für	$.(X,\ L)$
$[X1, \ldots,\ Xn]$	für	$.(X1,\ .(X2,\ \ldots\ .(Xn, []) \ldots))$.
$[X1, \ldots,\ Xn \mid L]$	für	$.(X1,\ .(X2,\ \ldots\ .(Xn,\ L) \ldots))$.

Der wesentliche Unterschied zu den funktionalen Listen besteht darin, dass durch das polymorphe Typkonzept die Listeneinträge in funktionalen Listen zwar einen beliebigen, aber einheitlichen Typ haben, während die Listen in Prolog beliebige Objekte beinhalten können. Es wird lediglich eingeschränkt, dass das zweite Argument des Listenkonstruktors eine Liste definiert:

```
is_list([]).
is_list([X | Xs])  :-  is_list(Xs).
```

Beispiel: Prädikate zur Listenverarbeitung in Logik-Programmen:

- `member` testet, ob ein Objekt `X` in einer Liste `[Y|Ys]` vorkommt.

```
member (X, [X|Ys]).
member (X, [Y|Ys]) :- member (X,Ys).
```

Mögliche Anfragen sind

`?- member (b,[a,b,c]).` → Antwort `yes`.

```
?- member (X,[a,b,c]).
?- member (b,Xs).
```

→ 3 Lösungen.
→ unendlich viele Lösungen.

- `append` definiert den Graphen der Listenkonkatenationsfunktion.

```
append ([], Ys, Ys).
append ([X|Xs], Ys, [X|Zs]) :- append (Xs, Ys, Zs).
```

Mögliche Anfragen sind

`?- append ([a,b],[c,d],L).`	→ L = [a,b,c,d].	Konkatenation
`?- append (L,[c,d],[a,b,c,d]).`	→ L = [a,b].	Präfixberechnung
`?- append ([a,b],L,[a,b,c,d]).`	→ L = [c,d].	Suffixberechnung
`?- append (L1,L2,[a,b,c,d]).`	→ 5 Lösungen	Listenaufteilung

Diese Anfragen zeigen die große Flexibilität von Prädikatsdefinitionen in Prolog-Programmen. Wegen der Unifikation sind alle Prädikatsargumente bidirektional verwendbar. So können neben der Listenkonkatenation mit `append` auch die Umkehrfunktionen und -relationen berechnet werden.

- `reverse` definiert die Listenspiegelung.

```
reverse ([],[]).
reverse ([X|Xs],Ys)  :-  reverse(Xs,Zs),
                         append(Zs,[X],Ys).
```

Wie bei funktionalen Sprachen kann auch hier eine alternative effizientere Definition angegeben werden:

```
reverse (Xs,Ys)     :-  rev (Xs,[],Ys).

rev ([],Ys,Ys).
rev ([X|Xs],Ys,Zs)  :-  rev(Xs,[X|Ys],Zs).
```

In einem Akkumulatorargument (2. Argument von `rev`) wird die gespiegelte Liste aufgebaut. Während der Aufwand der ersten Definition von `reverse` quadratisch in der Länge der zu spiegelnden Liste ist, arbeitet die zweite Definition mit einem linearen Aufwand.

◁

Logische Variablen sind existentiell quantifizierte Variablen, wie sie in Anfragen und manchmal in Regelrümpfen auftreten. Die Bedeutung einer Anfrage ist die Frage nach einer Belegung der in der Anfrageformel auftretenden Variablen derart, dass die Konjunktion der Anfrageliterale eine logische Folgerung der Programmformeln ist. Logische Variablen in Regeln sind solche, die im Regelrumpf, aber nicht im Regelkopf auftreten.

Beispiel: In der Regel

```
pfad(X,Z) :- kante(X,Y), pfad(Y,Z).
```

ist `Y` eine logische Variable, denn explizit bedeutet obige Regel:

$\forall$ X $\forall$ Y $\forall$ Z `pfad(X,Z) :- kante(X,Y), pfad(Y,Z).`

Dies ist äquivalent zu

$\forall$ X $\forall$ Z `pfad(X,Z) :-` $\exists$ `Y (kante(X,Y), pfad(Y,Z)).`

`Y` ist also im Rumpf der Regel existentiell quantifiziert. ◁

Logische Variablen sind der Schlüssel zur besonderen Ausdrucksstärke und Flexibilität von Logikprogrammen. Die Erzeugung von Bindungen solcher logischen Variablen bei der Bearbeitung von Anfragen erfolgt durch die Unifikation von Anfragevariablen mit Regelköpfen bzw. Fakten. Die Unifikation ist somit im Gegensatz zum Pattern-Matching funktionaler Sprachen ein *bidirektionaler* Parameterübergabemechanismus.
Dadurch dass im allgemeinen verschiedene Variablenbindungen existieren, die eine Anfrage zu einer Folgerung eines Logik-Programms machen, ist die operationelle Semantik von Logik-Programmen inhärent nichtdeterministisch.

9.4 Suchbasierte Berechnungen

Der inhärente Nichtdeterminismus in der prozeduralen Semantik von Prolog-Programmen kann dazu genutzt werden, den in einigen Problemen vorhandenen Nichtdeterminismus auszudrücken. Die Auswertungsstrategie von Prolog löst den Nichtdeterminismus zwar durch eine sequentielle Tiefensuche mit Backtracking auf und stellt auf diese Weise lediglich eine deterministische Approximation nichtdeterministischen Verhaltens zur Verfügung, aber dennoch wird die nichtdeterministische Programmierung erheblich erleichtert.
Bei der „generate-and-test“-Technik nutzt man das suchbasierte Berechnungsmodell von Prolog aus, indem fortlaufend Lösungsvorschläge erzeugt werden, die anschließend jeweils auf ihre Gültigkeit überprüft werden:

```
find(S) :- generate(S), test(S).
```

Falls ein Lösungsvorschlag nicht gültig ist, wird Backtracking zu `generate(S)` ausgelöst, damit ein neuer Lösungsvorschlag erzeugt wird. Diese Technik ist sehr einfach, aber relativ ineffizient, weil der gesamte Lösungsraum systematisch durchsucht wird. Sie wird häufig bei NP-vollständigen Problemen verwendet, da für diese i.a. ohnehin keine effizienten Lösungsverfahren bekannt sind.
Die Effizienz von „generate-and-test“-Programmen kann häufig erheblich dadurch gesteigert werden, dass der Test soweit wie möglich mit der Erzeugung von Lösungsvorschlägen verzahnt wird. Auf diese Weise wird sichergestellt, dass partielle Lösungen, die nicht mehr zu einer vollständigen Lösung fortgesetzt werden können, frühzeitig verworfen werden.

Beispiel: Ein typisches Beispielprogramm, das die „generate-and-test“-Methode verwendet, ist das folgende Sortierprogramm `permutationsort`, kurz `psort`.

Das Programm sortiert Listen natürlicher Zahlen, indem alle möglichen Permutationen einer Liste erzeugt werden und hinsichtlich der Ordnung ihrer Elemente

untersucht werden, bis die Permutation, die der geordneten Liste entspricht, gefunden ist.

```
psort (Xs,Ys)  :-  permut (Xs,Ys), ordered (Ys).

permut (Xs,[Z|Zs])  :-  select (Z,Xs,Ys), permut (Ys,Zs).
permut ([],[]).

select (X,[X|Ys],Ys).
select (X,[Y|Ys],[Y|Zs])  :-  select (X,Ys,Zs).

ordered ([]).
ordered ([X]).
ordered ([X|[Y|Ys]])  :-  less (X,Y), ordered ([Y|Ys]).

less (0,s(Y)).
less (s(X),s(Y))  :-  less (X,Y).
```

Die erste Regel des Beispielprogramms besagt, dass die Liste `Ys` eine Sortierung von `Xs` ist, falls `Ys` eine Permutation von `Xs` und geordnet ist. Das Prädikat `permut` definiert, wann eine Liste eine Permutation einer anderen Liste ist. Es wird durch eine Regel und einen Fakt definiert. Eine nicht-leere Liste `[Z|Zs]` ist eine Permutation der Liste `Xs`, falls `Z` aus `Xs` selektiert wird und die Restliste `Zs` eine Permutation der nach Selektion von `Z` aus `Xs` verbleibenden Liste `Ys` ist. Der Fakt besagt lediglich, dass die leere Liste eine Permutation von sich selbst ist.

Das Prädikat `select` ist erfüllt, wenn die Selektion des ersten Argumentes aus dem zweiten Argument dem dritten Argument entspricht. Die Prädikate *ordered* und *less* testen, ob eine Liste geordnet ist bzw. ob zwei Zahlen in der Kleiner-Beziehung stehen.

Zur Sortierung einer vorgegebenen Liste l stellt man an das Programm eine Anfrage der Form $\leftarrow$ `psort (`l`,S)`, wobei `S` eine logische Variable ist, an die nach einer erfolgreichen Berechnung die sortierte Liste gebunden ist. Das Programm könnte aber auch verwendet werden, um zu einer sortierten Liste Permutationen zu erzeugen, bzw. um zu zwei gegebenen Listen zu überprüfen, ob die zweite Liste eine Sortierung der ersten ist. ◁

Anhang A

Kurze Einführung in das HUGS System

Hugs ist ein frei verfügbarer Interpreter für Haskell und eine Umgebung für die Entwicklung von Haskell Programmen. Das System kann auf verschiedenen Plattformen installiert werden und kann mitsamt des Source Codes mittels anonymem ftp oder über das World-Wide Web (WWW) geladen werden.
Aktuelle Informationen über Haskell und frei verfügbare Compiler und Interpreter sind im WWW unter der Adresse

`http://www.haskell.org`

zu finden.

A.1 Start des Systems

Hugs wird durch Eingabe von

```
hugs [option | file]
```

gestartet. Zunächst lädt das System die Datei `prelude.hs` mit den vordefinierten Funktionen und Deklarationen. In der Kommandozeile angegebene Dateien werden anschließend geladen. Alternativ kann man das System ohne Angabe von Dateien starten und anschließend Dateien mittels des Hugs-Kommandos

```
:load f1 f2 ... fn
```

laden.

A.2 Hugs-Kommandos

Hugs stellt eine Reihe von Kommandos zur Auswertung von Ausdrücken, zum Laden von Dateien, zur Inspektion und Modifikation des Systems bereit. Fast alle Kommandos beginnen mit einem Doppelpunkt `:`, gefolgt von einem kurzen Kommandowort. Im allgemeinen genügt es, den ersten Buchstaben des Kommandos anzugeben. Zum

Beispiel können `:l, :s` und `:q` als Abkürzung für `:load`, `:set` und `:quit` verwendet werden.
Bei Eingabe eines Ausdrucks wird dieser ausgewertet und das Ergebnis ausgegeben. Folgende Tabelle enthält eine Kurzbeschreibung der wichtigsten Hugs-Kommandos:

`:?`	Ausgabe einer Liste aller Hugs-Kommandos
`:quit`, `:q`	Beenden von Hugs
`:load`, `:l [<file>]`	Laden und Compilieren einer Programmdatei
`:reload`, `:r`	Erneutes Laden der zuletzt geladenen Datei
`:![<command>]`	Ausführung eines shell-Kommandos
`:edit`, `:e [<file>]`	Editor starten

A.3 Syntaktische Besonderheiten in Haskell

Die Bezeichner für Funktionen und Variablen beginnen in Haskell mit einem Kleinbuchstaben. Die Bezeichner für Typen, Konstruktoren und Klassen beginnen mit einem Großbuchstaben. Es gibt eine Reihe von reservierten Worten wie beispielsweise `case, class, data, default, ...`
Kommentare beginnen mit `--` und erstrecken sich bis zum Zeilenende. Innere Kommentare werden mit {`- -`} geklammert.

Die Funktionsapplikation bindet stärker als Operatoren, d.h. der Ausdruck `f n - 1` ist gleichbedeutend mit `(f n) - 1`. In rekursiven Aufrufen müssen also unbedingt Klammern gesetzt werden: `f (n-1)`.

Der Gültigkeitsbereich von Definitionen hängt in Haskell vom Layout der Programme ab. Die sogenannte *Abseitsregel* legt fest, wo eine neue Definition beginnt. Sie lautet:

> Eine Definition in Haskell endet, wenn Text am selben linken Rand oder links davon auftritt.

Beispiel: Folgende Funktionsdefinition ist zulässig, aber natürlich nicht zu empfehlen:

```
test x y =
  (x*x)
   + 5
test2  x = 7
```

◁

In lokalen Definitionen wird der linke Rand des Geltungsbereichs durch das erste Symbol nach `where` bzw. `let` festgelegt. Zeilen, die weiter rechts von diesem Rand beginnen, setzen die vorherige Definition fort. Weitere Definitionen desselben Geltungsbereichs müssen exakt auf derselben Position beginnen.

Folgendes Layout wird für Haskell-Programme empfohlen:

```
f t1 ... tn                    f t1 ... tn
  = exp                          | bexp1  =  e1
  where l1 = ...                 | bexp2  =  e2
        l2 = ...                 ...
        ...                      | bexpk  =  ek
                                          where ...
```

In Haskell kann auch eine *formatfreie Syntax* verwendet werden. Die Zeichen { und } kennzeichnen den Anfang und das Ende eines Geltungsbereiches. Das Semikolon ; wird als Trennzeichen zwischen Definitionen verwendet.

Literatur

Lehrbücher

S. Thompson: *Haskell - The Craft of Functional Programming*, 2nd edition, Addison Wesley 1999, ISBN 0-201-34275-8.

P. Hudak: *The Haskell School of Expression — Learning Functional Programming Through Multimedia*, Cambridge University Press 2000, ISBN 0-521-64408-9.

P. Thiemann: *Grundlagen der funktionalen Programmierung*, Teubner Verlag 1994.

R. Bird: *Introduction to Functional Programming using Haskell, second edition*, Prentice Hall Europe 1998.

Zeitschriftenartikel

– P. Hudak, J. Fasel: *A gentle introduction to Haskell*, ACM SIGPLAN Notices, 27(5): T1-T53.

– P. Hudak: *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, Vol. 21, No. 2, September 1989, 359–411.

– J. Hughes: *Why Functional Programming Matters*, in: D.Turner (ed.): *Research Topics in Functional Programming*, Addison Wesley 1990.

– J. Backus: *Can Programming Be Liberated from the von Neumann Style? — A Functional Style and Its Algebra of Programs*, Communications of the ACM, Vol.21, No.8, August 1978.

Zeitschriften

– Journal of Functional Programming, Cambridge University Press

– Journal of Functional and Logic Programming, EAPLS

– ACM Transactions on Programming Languages and Systems (TOPLAS)