

Unterlagen: Grundlagen der Programmierung 2,
Sommersemester 2018

Prof. Dr. Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt

E-mail: schauss@ki.informatik.uni-frankfurt.de
URL: <http://www.ki.informatik.uni-frankfurt.de/>
Tel: 069 / 798 28597
Fax: 069 / 798 28919

11. April 2018

Kapitel 1

Programmiersprachen: rekursives Programmieren in Haskell

Das generelle Ziel dieses Teils der Vorlesung ist das Verständnis von Programmierparadigmen, insbesondere rekursive Programmierung, funktionaler Programmierung und auch logischer Programmierung. Dazu wird die Programmiersprache Haskell verwendet. Es wird auch auf die Eigenschaften der operationalen Semantik im allgemeinen und speziellen eingegangen, wobei Programmtransformationen eine besondere Rolle spielen.

Konventionen im Skript (teilweise) Im Folgenden sind Bildschirm-Ein- und ausgaben in Schreibmaschinenschrift gegeben und

in grau hinterlegten Kästen mit durchgezogenen Linien und runden Ecken zu finden.

Später werden wir auch sogenannte Quelltexte darstellen, diese sind

in gestrichelten und eckigen Kästen zu finden.

1.1 Programmiersprachen: Paradigmen und Begriffe

Erläuterung einiger Begriffe zu Programmiersprachen:

- Die *Syntax* einer Programmiersprache beschreibt, welche Texte als gültige Programme gelten. Diese wird i.a. als eine kontextfreie Grammatik angegeben, evtl. mit zusätzlichen Bedingungen.

- Die *Semantik* einer Programmiersprache beschreibt die Bedeutung eines gültigen Programms. Es gibt verschiedene Formen der Semantik, die wir noch besprechen werden: denotationale, operationale, axiomatische und transformations-basierte. Wir werden im wesentlichen auf die operationale Semantik und teilweise auf transformations-basierte Semantik eingehen.
- Ein *Interpreter* führt ein Programm aus, (bzw. wertet ein Programm aus), wobei er den Text des Programms als Grundlage hat. Jeder Programmbefehl wird einzeln eingelesen und dann ausgeführt (ausgewertet). Ein Interpreter ist normalerweise eine direkte Implementierung der operationalen Semantik
- Ein *Compiler* (Übersetzer) erzeugt aus einem textuell eingegebenen Programm ein auf einem Rechner ausführbares Modul. Hier werden Programmumstellungen, Optimierungen usw. durchgeführt. Der Effekt des Programms muss dabei gleich bleiben, aber der Ablauf orientiert sich nicht mehr an der ursprünglichen Struktur des eingegebenen Programms.
Die eigentliche Ausführung des Programms erfolgt dann in einer Ablaufumgebung.

Natürlich gibt es Zwischenformen: z.B. Ein Interpreter, der vorher das Programm schon mal transformiert

Bei Ausführung eines Programms unterscheidet man (unabhängig) davon, ob man einen Interpreter bzw. einen Compiler hat, die zwei Zeiträume:

- *Compilezeit*: hiermit klassifiziert man Aktionen bzw. Fehler, die beim Analysieren bzw. Übersetzen des Programms auftreten.
Z.B. statische Typüberprüfung, Optimierung.
- *Laufzeit*: hiermit klassifiziert man Aktionen und Fehler, die während der Laufzeit des Programms auftreten.
Z.B. dynamische Typüberprüfung, Ein-Ausgabe.

1.1.1 Algorithmen und Programmiersprachen

Programmiersprachen lassen sich charakterisieren nach ihren wesentlichen Merkmalen, auch **Programmier-Paradigmen** genannt.

imperativ Ein imperatives Programm besteht aus einer Folge von *Anweisungen*, die sukzessive den *Zustand*, (d.h. den Speicherinhalt) einer Maschine (siehe von-Neumann-Architektur) manipulieren (Seiteneffekte). Das Ergebnis des Programms ist der veränderte Zustand der Maschine nach Durchlaufen aller Anweisungen.

D.h. in einem imperativen Programm wird präzise beschrieben, was (welche Anweisung), wann (Reihenfolge der Anweisungen) womit (Speicherplatz/Variable) zu geschehen hat.

prozedural: Vornehmlich als Mittel zur Strukturierung bieten mittlerweile alle imperativen Programmiersprachen (ausgenommen z.B. Ur-BASIC) die Möglichkeit, Teile des Programmes in separate Unterprogramme auszugliedern. Diese Prozeduren werden dann an den benötigten Stellen mit Argumenten aufgerufen. Diese Prozeduren können nicht nur vom Hauptprogramm aufgerufen werden, sondern auch untereinander. Damit soll die Strukturierung und Übersichtlichkeit des Programms und die Wiederverwendbarkeit von Programmteilen erhöht werden.

funktional: Das Ergebnis eines funktionalen Programms ist durch die *Auswertung* eines *Ausdrucks* gegeben. Ausdrücke bestehen dabei aus der *Anwendung* von Funktionen auf ihre Argumente.

In einem funktionalen Programm formalisiert man also, wie die Lösung zusammengesetzt ist bzw. was berechnet werden muss, und nicht, welche einzelnen Schritte dazu durchzuführen sind.

Es gibt zwei Varianten der funktionalen Programmiersprachen: strikte und nicht-strikte. Nicht-strikte funktionale Programmiersprachen sind pur (d.h. haben keine Seiteneffekte), sind nicht imperativ, und verwenden Funktionen sinngemäß wie Prozeduren. Strikte funktionale Programmiersprachen können pur sein, haben aber im allgemeinen auch imperative Anteile, da die Auswertungsreihenfolge fix ist.

deklarativ: Mit einem deklarativen Programm (Spezifikation) wird formalisiert, wie die Lösung aussieht, aber nicht, wie sie berechnet werden soll. D.h. diese Spezifikation ist zunächst mal nicht algorithmisch.

Es gibt deklarative Spezifikationsmethoden, die eine nicht-algorithmische Spezifikation ausführen können, so dass man eine *Spezifikation* des gesuchten Resultats auch als ein „Programm„ interpretieren kann.

Zur Angabe der Spezifikationen werden meist formale Logiken zu Hilfe genommen (logische Programmierung, Prolog).

objektorientiert: Objektorientierte Programmierung geht einher mit einer Strukturierung eines Programms in Klassen. Auch hier ist das Ziel eine bessere, übersichtlichere Strukturierung und eine Wiederverwendung von Programmteilen.

Die Ausführung des Programms basiert auf dem *Austausch* von *Nachrichten* zwischen *Objekten*. Ein Objekt kann auf eine Nachricht reagieren, indem es weitere Nachrichten an andere Objekte versendet und/oder seinen *internen Zustand* manipuliert.

Normalerweise ist die Objektorientierung ein Konzept, dass in imperativen Programmiersprachen eingesetzt wird.

typisiert: Alle Programmiersprachen sind mehr oder weniger stark getypt. Man unterscheidet zwischen

- statischer Typisierung: Typisierung, die sich auf den Programmtext, d.h. die Variablen, Funktionen, Prozeduren, Daten usw. bezieht.
- dynamischer Typisierung: Zur Ausführungszeit des Programms werden die Daten nach Typen klassifiziert, Die Prozeduren prüfen die Typisierung der Eingabedaten und der verwendeten Übergabedaten.

Man unterscheidet auch zwischen

- schwacher Typisierung: Normalerweise mit dynamischer Typisierung gekoppelt. Es sind Programme erlaubt, die einen Typfehler zur Laufzeit machen können.
- starker Typisierung: Es wird zur Laufzeit kein Typfehler erlaubt, indem der Compiler Programme ausschließt, die bestimmte Typbedingungen nicht erfüllen.

Viele Programmiersprachen vereinigen unterschiedliche Aspekte, z.B. ist Haskell funktional und stark getypt; es ist auch in gewisser Weise deklarativ. Bei der Programmierung in der imperativen Sprache C kann man zu einem gewissen Grad funktionale Prinzipien anwenden, usw.

Desweiteren bestehen zwischen verschiedenen Programmiersprachen ein und derselben Gruppe oft auch Unterschiede in Bezug auf die Unterstützung wichtiger *Strukturierungsprinzipien* wie Abstraktion, Kapselung, Hierarchisierung und Modularisierung.

1.2 Einführung in die funktionale Programmiersprache Haskell

Diese Programmiersprache ist funktional, nicht-strikt, hat ein polymorphes und starkes Typsystem. Sie hat ein reiches und klar strukturiertes Typsystem mit entsprechend flexiblen Datenstrukturen und guten Abstraktionseigenschaften. Wir werden sie in diesem Teil der Vorlesung für Algorithmen, insbesondere rekursive Algorithmen, und zur Demonstration von Programmierkonzepten verwenden.

Wesentliches Konstrukt ist die Funktionsdefinition. Die Ausführung eines Programms besteht in der Auswertung von Ausdrücken, die Anwendungen von Funktionen auf Argumente darstellen. Die Anwendung einer Funktion auf die gleichen Argumente soll immer das gleiche Ergebnis liefern: das ist die Eigenschaft der *referentiellen Transparenz*.

Funktionale pure Programmierung hat in bestimmten Bereichen Vorteile: Haskell Programmierung ergibt Programme die eher eine Spezifikation sind: Man kann unabhängig testen, schnell Programme sicher ändern, und concurrency nutzen in Concurrent Haskell.

Zum Beispiel Facebook benutzt Haskell zur Abwehr von Spam, Hackern und Missbrauch.

Aus einem Artikel von *Simon Marlow 2016*

<https://code.facebook.com/posts/745068642270222/>

[fighting-spam-with-haskell/https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/](https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/)

Es gibt eine Schnittstelle (in Haskell programmiert) zur Eingabe und Ausführung von Regeln. Das sind sogenannte “policies”, die verwendet werden zur Abwehr von Spam, und die oft geändert und erweitert werden.

Vorteile von Haskell, die hierbei wichtig sind:

- Pur funktional und streng getypt]
- Concurrency (Nebenläufigkeit), die man in Concurrent Haskell automatisch hat.
- Leichte und schnelle Code-Aktivierung.
- Performanz.

1.2.1 Algorithmen in Haskell

Haupt-Konstruktionsmethoden von Algorithmen in Haskell sind die Definition von Funktionen, die Anwendung von Funktionen auf Argumente und die Auswertung von Ausdrücken. Die Ausdrücke bestehen i.a. aus geschachtelten Anwendungen von Funktionen auf Argumente, wie z.B. in $f (g\ 4) (h\ 3)$ oder als konkreteres Beispiel: `sin (quadrat 3)`.

Wichtige Eigenschaften **funktionaler Programmiersprachen** wie Haskell:

Referentielle Transparenz Der Wert von Funktionsanwendungen ($f\ t_1 \dots t_n$) ist nur durch den Wert der Argumente t_i bestimmt. Mehrfacher Aufruf einer Funktion mit den gleichen Argumenten ergibt immer den gleichen Wert. D.h. es gibt keine Seiteneffekte, insbesondere keine dauerhaften Speicheränderungen, die von anderen Funktionen aus sichtbar sind.

In imperativen Programmiersprachen wie Python, Java, C, Pascal usw. gilt das i.a. nicht.

Verzögerte Auswertung Nur die für das Resultat notwendigen Unterausdrücke werden ausgewertet.

Polymorphes Typsystem Nur Ausdrücke, die einen Typ haben, sind zulässig. Man kann Funktionen schreiben, die Argumente von verschiedenem Typ bearbeiten können. Z.B. einen Typ $\forall a. a \rightarrow a$ haben, das wäre der Typ einer Funktion, die ihr Eingabeargument wieder als Ausgabe hat. Das Typsystem garantiert, dass niemals Daten vom Typ her falsch interpretiert werden.

Ein wesentliches Feature ist, dass Haskell den allgemeinsten Typ von Ausdrücken und Funktionen herleiten kann, wenn kein Typ angegeben ist. Typen.

Automatische Speicherverwaltung Anforderung von Speicher und Freigabe von unerreichbaren Objekten erfolgt automatisch.

Funktionen sind Datenobjekte Funktionen können wie Zahlen behandelt werden, d.h. sie können in Datenobjekten als Teil auftauchen und sind als Argument und Resultat von Funktionen zugelassen.

Grundprinzipien der Haskell-Programmierung sind:

- Definition von Funktionen.

```
quadrat x = x*x
```
- Aufbau von Ausdrücken: Anwendung der Funktion auf Argumente, die selbst wieder Ausdrücke sein können.

```
a*(quadrat x)
```
- Nur der Wert von Ausdrücken wird bei der Auswertung weitergegeben.

Vordefiniert sind:

- **Zahlen:** ganze Zahlen vom Typ `Int` mit $|n| < 2^{31} - 1 = 2147483647$, beliebig lange ganze Zahlen (vom Typ `Integer`), rationale Zahlen (`Rational`), Gleitkommazahlen (`Float`) und entsprechende arithmetische Operatoren: `+`, `-`, `*`, `/`, `^`. Arithmetische Vergleichsoperatoren: `==`, `/=`, `>=`, `<=`, `>`, `<`, die Infix geschrieben werden.
- Zeichen (Character) `'a'`, `'A'`, ... vom Typ `Char` sind ebenfalls schon vordefiniert.
Strings sind Listen von Zeichen, aber das wird noch genauer besprochen bei der Listenverarbeitung.
- Boolesche Werte: `True`, `False`, die Fallunterscheidung (`if then else`) und Boolesche Verknüpfungen für und, oder, nicht, die folgendermaßen definiert sind:

```
x && y = if x then y      else False
x || y = if x then True  else y
not x  = if x then False else True
```

Wir legen folgende Sprechweise fest: Ein *Basiswert* ist entweder eine Zahl, oder ein Zeichen vom Typ `Char` oder einer der Booleschen Werte `True`, `False`.

1.2.2 Typen

Funktionen haben immer einen Typ, wobei der Typ der formalen Parameter und des Ergebnisses im Typ vorkommen. Der Typ einer Funktion f mit n Argumenten sieht immer so aus:

$f :: a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow b$

Das sind die n Typen a_i der Argumente und b als Typ des Arguments.

Z.B. hat `quadrat` den Typ `Integer -> Integer`. d.h. das Argument kann ein Ausdruck vom Typ `Integer` sein, das Ergebnis ist dann ebenfalls vom Typ `Integer`. Geschrieben wird das als

```
quadrat :: Integer -> Integer
```

`quadrat` hat noch mehr Typen, zB gleichzeitig den Typ `Int -> Int`. Der Typ der arithmetischen Operationen ist analog, z.B.

```
* :: Integer -> Integer -> Integer
```

und auch noch weitere Typen, z.B. `Int -> Int -> Int`
Arithmetische Vergleiche haben als Typ z.B.

```
>= :: Integer -> Integer -> Bool,
```

Boolesche Operatoren haben als Argument- und Ergebnistyp jeweils `Bool`:
z.B.

```
&& :: Bool -> Bool -> Bool
```

Allerdings ist das Typsystem von Haskell allgemeiner, flexibler und damit auch komplizierter. Man kann statt festen Typen auch Typvariablen im Typ verwenden, die eine Menge von Typen kompakt darstellen können. Dies wird noch genauer besprochen.

Man beachte, dass die Sprache Haskell ein Typsystem besitzt, das nur typbare Ausdrücke zulässt.¹

Beispiel 1.2.1 *Definition eines Polynoms x^2+y^2 , passend zum Satz des Pythagoras, mit Eingabe x, y . Berechnet wird das Quadrat der Hypotenuse im rechtwinkligen Dreieck. Wir geben auch den Typ mit an in einer sogenannten Typdeklaration. Die Schreibweise `Integer -> Integer -> Integer` bei zweistelligen Funktionen bedeutet, dass die beiden Argumente vom Typ `Integer` sein müssen, und das Ergebnis dann vom Typ `Integer` ist.*

¹in `ghci` kann man mit `:t expr` den Typ von `expr` erhalten, wenn `expr` sinnvoll ist.


```
quadratsumme:: Integer -> Integer -> Integer
quadratsumme x y = (quadrat x) + (quadrat y)
```

```
myuser$ ghci
GHCi, version ...
Loading package ... done.
...
Prelude> :load prg2.hs
*Main> quadratsumme 3 4
25
```

Syntax von Haskell

Definition 1.2.2 Die (vereinfachte) Syntax mittels einer kontextfreien Grammatik für eine Funktionsdefinition ist:

$$\begin{aligned}
 \langle \text{FunktionsDefinition} \rangle &::= \langle \text{Funktionsname} \rangle \langle \text{Parameter} \rangle^* = \langle \text{Ausdruck} \rangle \\
 \langle \text{Ausdruck} \rangle &::= \langle \text{Bezeichner} \rangle \mid \langle \text{Zahl} \rangle \\
 &\quad \mid (\langle \text{Ausdruck} \rangle \langle \text{Ausdruck} \rangle) \\
 &\quad \mid (\langle \text{Ausdruck} \rangle) \\
 &\quad \mid (\langle \text{Ausdruck} \rangle \langle \text{BinInfixOp} \rangle \langle \text{Ausdruck} \rangle) \\
 &\quad \mid \langle \text{If-Ausdruck} \rangle \\
 \langle \text{Bezeichner} \rangle &::= \langle \text{Funktionsname} \rangle \mid \langle \text{Datenkonstruktorname} \rangle \\
 &\quad \mid \langle \text{Parameter} \rangle \mid \langle \text{BinInfixOp} \rangle \\
 \langle \text{BinInfixOp} \rangle &::= * \mid + \mid - \mid / \\
 \langle \text{If-Ausdruck} \rangle &::= \text{if } \langle \text{Ausdruck} \rangle \text{ then } \langle \text{Ausdruck} \rangle \text{ else } \langle \text{Ausdruck} \rangle
 \end{aligned}$$

Definition 1.2.3 Ein Programm ist definiert als eine Menge von Funktionsdefinitionen. Es gibt immer eine Definition der Konstanten `main`.

Zur Erläuterung betrachten wir die Funktionsdefinition

```
quadratsumme x y = (quadrat x) + (quadrat y)
```

Diese Grammatik sagt, dass das eine Funktionsdefinition ist, `quadratsumme` ist ein Funktionsname, `x,y` sind die formalen Parameter. Nach dem Gleichheitszeichen kommt ein Ausdruck, der als `Ausdruck + Ausdruck` interpretiert wird, und `+` ein binärer Infix-Operator ist. Die beiden Ausdrücke um das `+` sind jeweils `(quadrat x)` und `(quadrat y)`, die entsprechend der Grammatik Anwendungen sind: `quadrat` ist ein Ausdruck und `x` ein Ausdruck. Das Hineinschreiben ergibt ebenfalls einen Ausdruck: der linke Ausdruck wird ist die Funktion, der rechte Ausdruck das Argument. D.h. die Funktion selbst ist möglicherweise erst nach einer Berechnung bekannt.

In Haskell sind Bezeichner im allgemeinen Namen, die mit einem Buchstaben beginnen müssen und dann aus Buchstaben und Ziffern bestehen. Es gibt auch Bezeichner, die nur aus Sonderzeichen bestehen: diese sind dann nur als Funktionsname erlaubt.

Die Parameter nennt man auch *formale Parameter*. Wir nennen diese manchmal auch *Variablennamen* bzw Variablen. Die Anzahl der formalen Parameter einer Funktion f wird durch die Definition festgelegt. Wir nennen sie die *Stelligkeit* ($ar(f)$) der Funktion f . In einer Anwendung $(f\ t_1\ \dots\ t_n)$ nennt man t_i die *Argumente* der Funktion f .

Einige Funktionsnamen und Datenkonstruktornamen sind schon im “Prelude“ evtl. vorhanden, weitere sind benutzerdefinierbar. *Datenkonstruktoren* müssen mit Großbuchstaben beginnen, damit sie sich von Funktionsnamen unterscheiden. Beispiele sind: **True**, **False**. Es sind verschiedene Arten von Zahlen verfügbar: kurze ganze Zahlen (**Int**), beliebig lange ganze Zahlen (**Integer**), Brüche (**Rational**), Gleitkommazahlen (**Float**).

Da Haskell sowohl Präfix als auch Infix-Operatoren und auch Prioritäten für Operatoren hat, werden oft Klammern weggelassen, wenn das Programm eindeutig lesbar bleibt. Eine Spezialität ist das Weglassen der Klammern bei (links-)geschachtelten Anwendungen: $s_1\ s_2\ \dots\ s_n$ ist dasselbe wie $((\dots(s_1\ s_2)\ s_3\ \dots)\ s_n)$. Partielle Anwendungen, d.h. Anwendungen von Funktionen auf zuwenig Argumente, sind erlaubt. Diese werden aber erst ausgewertet, wenn in einem späteren Stadium der Auswertung die volle Anzahl der Argumente vorliegt.

Es ist zu beachten, dass es weitere Bedingungen an ein korrektes Programm gibt: Es muss eine korrekte Haskell-Typisierung haben und es muss bestimmte Kontextbedingungen erfüllen: z.B. die formalen Parameter müssen verschiedene Namen haben. Außerdem müssen alle undefinierten Namen im rechten Ausdruck einer Definition formale Parameter sein.

Zur Strukturierung und Trennung können außerdem noch die Klammern “{“, “}“ und das Semikolon “;“ verwendet werden. Diese Syntax werden wir schrittweise erweitern und mit Beispielen erläutern.

Die für den Benutzer relevante Syntax benutzt auch das *Layout* des Programms. D.h. die Einrückungen zeigen eine Strukturierung an, die intern eine Klammerung mit {, } bewirkt. Dies ist eine syntaktische Strukturierung moderner Programmiersprachen. Die Einrückungen, die man ohnehin macht, um das Programm lesbarer zu machen, werden ernst genommen und zur Strukturerkennung des Programms (vom Interpreter bzw Compiler) mit benutzt.

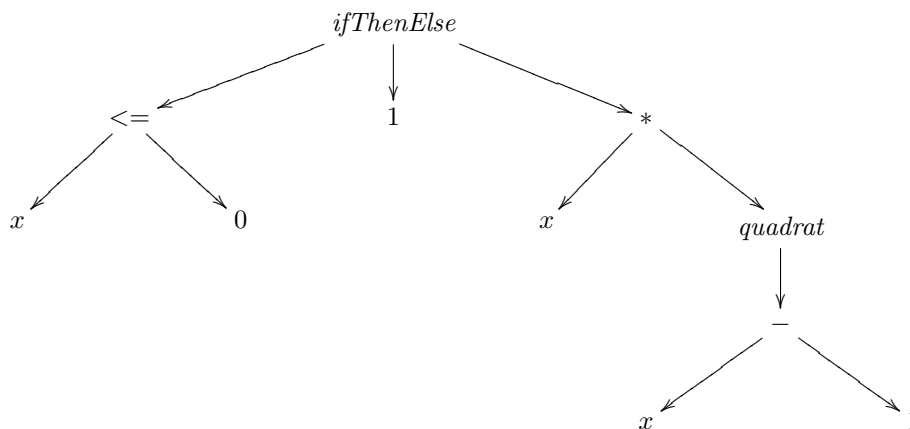
Wie in vielen Programmiersprachen gibt es verschiedene syntaktische Darstellungsweisen des Programms:

- Benutzer-Syntax: diese wird vom Programmierer benutzt, und kann eine Erweiterung, oder eine andere Darstellung sein. In Haskell sind die Abweichungen von der (einfachen) kontextfreien Grammatik: Prioritäten, Weglassen von Klammern, Layout-Regeln, ...

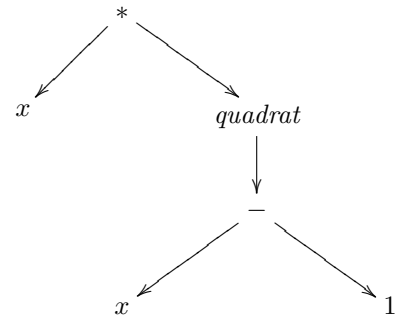
- Interne Syntax: “Linearisierung“. Die Syntax der Sprache, die sich leichter beschreiben lässt, und maschinenintern (leichter) verwendet werden kann: Eine “Entzuckerung“ hat stattgefunden: Die Ausdrücke sind voll geklammert, alle Operatoren sind Präfix, Layout durch Klammern ersetzt, ...
- Ableitungsbaum (Herleitungsbaum) Die Datenstruktur, die vom Compiler nach der syntaktischen Analyse erzeugt wird, und die jedes Zeichen (bzw. Namen, Zahlen usw. : Token) des Programms in einen Baum entsprechend der Grammatik einsortiert.
- Syntaxbaum: Eindeutige Darstellung des Programms und der Ausdrücke in einem markierten Baum. Dieser wird normalerweise vom Compiler/Interpreter intern erzeugt.
Im Kapitel Compilerbau: Man kann den Syntaxbaum aus dem Herleitungsbaum durch Weglassen redundanter Zeichen/Token gewinnen. Evtl. ist eine leichte Umstrukturierung notwendig. Hieraus lässt sich eindeutig die Ausführung des Programms definieren.

Einen Syntaxbaum kann man oft auch direkt von Hand angeben. Es kommt manchmal vor, dass der Interpreter (Compiler) einen anderen Syntaxbaum erzeugt hat als den erwarteten. In diesem Fall hilft manchmal das Einfügen von Klammern in der Benutzer-Syntax.

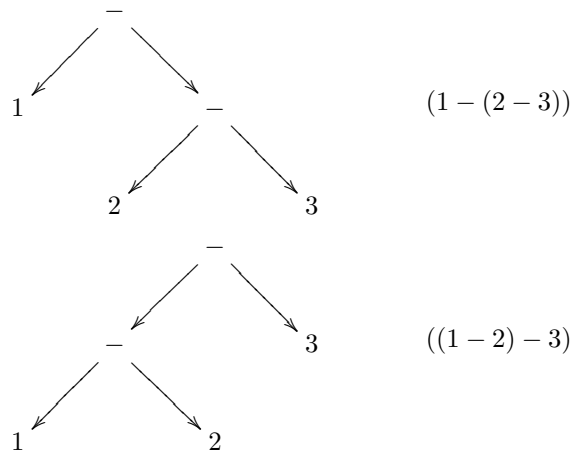
Beispiel 1.2.4 `if x <= 0 then 1 else x*(quadrat (x-1))`



`x*(quadrat (x-1))`



Zwei Syntaxbäume zu 1-2-3,
von denen nur einer in Haskell gültig ist:



1.2.3 Begriffe zur Rekursion

Definition 1.2.5 Man sagt f **referenziert** g **direkt**, wenn g im Rumpf von f vorkommt.

Man sagt f **referenziert** g , wenn f die Funktion g direkt referenziert; oder wenn es Funktionen f_1, \dots, f_n gibt, so dass gilt: f referenziert direkt f_1 , f_1 referenziert direkt f_2 , \dots , f_n referenziert direkt g .

Eine Haskellfunktion f nennt man **direkt rekursiv**, wenn f sich selbst direkt referenziert, d.h. der Rumpf der Funktion wieder einen Aufruf der Funktion f enthält.

Eine Haskellfunktion f nennt man **rekursiv**, wenn f sich selbst referenziert.

Wenn f die Funktion g referenziert und g die Funktion f , dann spricht man auch von **verschränkter Rekursion**. Das überträgt man auch auf allgemeinere Fälle von n Funktionen.

Beachte: Wenn f eine rekursive Funktion aufruft, heißt das noch nicht, dass f rekursiv ist.

Beispiel 1.2.6 Die Funktion

```
quadratsumme x y = (quadrat x) + (quadrat y)
```

ruft direkt die Funktionen $+$ und **quadrat** auf, dies letzte wiederum die (eingebaute) Funktion $*$. Die Funktion **quadratsumme** referenziert die Funktionen $\{\text{quadrat}, *, +\}$ ist somit nicht rekursiv.

Wir betrachten die mathematisch definierte Fakultätsfunktion $n!$,

$$\begin{aligned} 0! &:= 1 \\ n! &:= n * (n-1)! && \text{wenn } n > 0 \end{aligned}$$

die die Anzahl aller Permutationen einer n -elementigen Menge angibt, und schreiben dazu eine Haskell-Funktion, indem wir die Definition direkt kodieren.

```
fakultaet :: Integer -> Integer
fakultaet x = if x <= 0 then 1
              else x*(fakultaet (x-1))
```

Diese Funktion ist rekursiv, da sie im Rumpf sich selbst wieder aufruft. Bei rekursiven Funktionen wie **fakultaet** muss man sich immer klarmachen, dass man zwei Fälle beachten muss:

- den **Basisfall**: Ergebnis 0 wenn das Argument $x \leq 1$ ist.
- den **Rekursionsfall**: Ergebnis: $x * (\text{fakultaet } (x-1))$, wenn $x > 1$ ist.

Zusätzlich muss man sich vergewissern, dass der rekursive Aufruf **fakultaet** $(x-1)$ auch näher an den Basisfall kommt, d.h. in diesem Fall, dass die Argumente kleiner werden und der Basisfall das kleinste Argument ist. Das ist hier

der Fall, da der rekursive Aufruf mit kleinerem Argument (nach Auswertung) erfolgt.

Das funktioniert wie erwartet:

```
*Main> fakultaet 3 ↵
6
*Main> fakultaet 40 ↵
815915283247897734345611269596115894272000000000
```

Man sieht aber auch, dass diese Funktion bei negativen Eingaben auch terminiert, aber immer den (falschen) Wert 1 zurückgibt.

Betrachte folgende fehlerhafte Definition:

```
fakultaet_nt :: Integer -> Integer
fakultaet_nt x = if x == 0 then 1
                  else x*(fakultaet_nt (x-1))
```

Diese Funktion terminiert bei negativen Eingaben nicht, da z.B. `fakultaet_nt (-5)` als rekursiven Aufruf `fakultaet_nt (-6)` hat usw., und somit den Basisfall nicht erreichen kann.

1.2.4 Beispiel: Berechnung von Schaltjahren

Schaltjahre sind in unserem Jahreskalender notwendig, da das tropische Jahr nicht genau 365 Tage ist, sondern 365.242190517 Tage. Dies wird im Gregorianischen Kalender durch Einfügen von Schaltjahren mit 366 Tagen korrigiert.

Der Haskell-Algorithmus dazu folgt. Er verwendet den Operator `mod`, der durch Einklammern `'mod'` in einen Infix-Operator verwandelt werden kann.

```
ist_ein_schaltjahr n =
  if n > 1582
  then n 'mod' 400 == 0
       || (n 'mod' 4 == 0 && n 'mod' 100 /= 0)
  else error
       "Jahreszahl vor Einfuehrung des Gregorianischen Kalenders"
```

Eine rekursive Funktion zum Ermitteln des nächsten Schaltjahres mit $j \geq n$ ist:

```
naechstes_schaltjahr n = if (ist_ein_schaltjahr n)
                          then n
                          else naechstes_schaltjahr (n+1)
```

Der orthodoxe Kalender hat bei den durch 100 teilbaren Jahreszahlen eine andere Definition der Schaltjahre:

```

ist_ein_schaltjahr_ortho n =
  if n > 1582 then
    (n `mod` 100 == 0
     && (n `mod` 900 == 200 || n `mod` 900 == 600))
    || (n `mod` 4 == 0 && n `mod` 100 /= 0)
  else error "Jahreszahl vor Einfuehrung"

```

Man kann dazu folgendermaßen die mittlere Jahreslänge bestimmen:

```

mittlere_jahreslaenge::Double
mittlere_jahreslaenge = 365.242190517
mittlere_jahreslaenge_gregor =
  fromInteger (mittlere_jahreslaenge_sum 2000 2399 0) / 400.0
mittlere_jahreslaenge_sum von bis summe =
  if von > bis then summe
  else if ist_ein_schaltjahr von
    then mittlere_jahreslaenge_sum (von+1) bis (summe + 366)
    else mittlere_jahreslaenge_sum (von+1) bis (summe + 365)

mittlere_jahreslaenge_ortho::Double
mittlere_jahreslaenge_ortho =
  fromInteger
    (mittlere_jahreslaenge_ortho_sum 2000 2899 0) / 900.0
mittlere_jahreslaenge_ortho_sum von bis summe =
  if von > bis then summe
  else if ist_ein_schaltjahr_ortho von
    then
      mittlere_jahreslaenge_ortho_sum (von+1) bis (summe + 366)
    else
      mittlere_jahreslaenge_ortho_sum (von+1) bis (summe + 365)

```

Das erste unterschiedlich behandelte Jahr kann man ermitteln mit folgender Funktion

```

schaltjahr_anders n =
  if ist_ein_schaltjahr n == ist_ein_schaltjahr_ortho n
  then schaltjahr_anders (n+1)
  else n

```

1.3 Auswertung in Haskell

Wir gehen hier auf die operationale Semantik von Haskell ein, d.h. wir werden einen Interpreter spezifizieren, nur auf der Grundlage des Programmtextes. Diese operationale Semantik basiert auf Transformationen der Programme, ist somit auch eine Form einer Transformations-Semantik.

1.3.1 Auswertung, operationale Semantik von einfachen Haskell-Programmen

Jetzt haben wir genug Beispiele, um einen einfachen Ausschnitt aus der *operationalen Semantik* von Haskell zu definieren und zu verstehen. Auch wenn diese operationale Semantik zunächst etwas gewöhnungsbedürftig ist, so hat sie später den Vorteil, dass formal sauber geklärt werden kann, wie Funktionen auszuwerten sind, die auch Funktionen als Argumente oder Werte haben können (sog. Funktionen höherer Ordnung). Ein weiterer Vorteil ist die Unabhängigkeit von Compilern und spezifischen Rechnern (die Portabilität).

Wir beschreiben die Auswertung von Programmen als eine **Folge von Transformationen**, die ein Anfangsprogramm P_0 nacheinander in Programme P_1, P_2, \dots transformiert, bis ein Programm entstanden ist, das sich nicht weiter transformieren lässt. Hieraus ist der Wert des Programms extrahierbar als Ausdruck der an `main` gebunden ist.

Wir werden 2 verschiedene *Strategien der Auswertung* betrachten, die **normale Reihenfolge der Auswertung**, die auch in Haskell verwendet wird, und die **applikative Reihenfolge der Auswertung**, die auch in Programmiersprachen verwendet wird.

Wenn die Strategie festgelegt ist, dann ist die Auswertungs-Transformation, wenn sie möglich ist, für jedes Programm eindeutig definiert.

Das Prinzip der Berechnung ist:

Auswertung = Folge von Transformationen des Programms
bis ein Basiswert erreicht ist

D.h. es gibt eine Folge $\text{main} \rightarrow t_1 \rightarrow \dots t_n \dots$, so dass im Falle, dass es keine weitere Transformationsmöglichkeit gibt und t_n ein Basiswert ist, der Wert berechnet ist. D.h. entweder endet die Folge mit einem Basiswert (sie terminiert), oder die Folge ist unendlich (sie terminiert nicht), oder es gibt einen Fehler. Der erreichte Basiswert wird als Ergebnis des Programms definiert. Im Falle eines Fehlers oder Nichtterminierung ist das Ergebnis undefiniert.

Wir benötigen folgende **Transformationen**, die überall im Programm verwendet werden können:

- die *Definitionseinsetzung* (δ -Reduktion):

$$(f\ t_1 \dots t_n) \rightarrow (\text{Rumpf}_f[t_1/x_1, \dots, t_n/x_n])$$

Wenn f die Stelligkeit n hat und die formalen Parameter im Rumpf von f die Namen x_1, \dots, x_n haben und Rumpf_f der Rumpf der Funktionsdefinition von f ist.

Der Ausdruck $(\text{Rumpf}_f[t_1/x_1, \dots, t_n/x_n])$ entsteht durch (paralleles bzw. unabhängiges) Ersetzen der formalen Parameter x_i durch die jeweiligen Ausdrücke (die Argumente) t_i . D.h. die Ausdrücke, die die Argumente sind, werden textuell für die formalen Parameter eingesetzt. Das müssen keine Basiswerte sein!

- *Auswertung von arithmetischen Ausdrücken* $s\ op\ t \rightarrow r$, falls s, t arithmetische Basiswerte sind, und op ein zweistelliger arithmetischer Operator, und r das Resultat der arithmetischen Operation ist.
- *Auswertung von arithmetischen Ausdrücken* $op\ t \rightarrow r$, falls s ein arithmetischer Basiswert ist, und op ein einstelliger arithmetischer Operator, und r das Resultat der arithmetischen Operation ist.
- Boolesche Operatoren: werden transformiert wie Funktionen, so wie diese im Haskell-Prelude definiert sind.
Wir verwenden ersatzweise:

```
x && y = if x then y      else False
x || y = if x then True  else y
not x  = if x then False else True
```

- *Auswertung einer Fallunterscheidung (if-Reduktion):*

$$\begin{aligned} (\text{if True then } e_1 \text{ else } e_2) &\rightarrow e_1 \\ (\text{if False then } e_1 \text{ else } e_2) &\rightarrow e_2 \end{aligned}$$

Es ist klar, dass die Regeln für **if-then-else** nur angewendet werden können, wenn der Bedingungsausdruck zu **True** oder **False** ausgewertet ist.

Wir nennen eine Transformation auch *Reduktion* und eine Folge von Programmtransformationen auch *Reduktionsfolge* (oder *Auswertung*).

Offenbar gibt es Situationen, in denen man an verschiedenen Stellen eines Ausdrucks die Ersetzungsregeln anwenden kann.

Beispiel 1.3.1 Für den Ausdruck `quadrat(4 + 5)` gibt es 3 verschiedene *Reduktionsfolgen*.

1. `quadrat(4 + 5) → (4 + 5) * (4 + 5) → 9 * (4 + 5) → 9 * 9 → 81`
2. `quadrat(4 + 5) → (4 + 5) * (4 + 5) → (4 + 5) * 9 → 9 * 9 → 81`

3. `quadrat(4 + 5) → (quadrat 9) → 9 * 9 → 81`

Es gibt zwei wichtige *Reduktionsstrategien*, die jeweils eindeutige Auswertungs-Transformationen bestimmen. Eine Kurzcharakterisierung ist:

Normale Reihenfolge:

Von außen nach innen;
von links nach rechts;
 δ -Reduktion vor Argumentauswertung

Applikative Reihenfolge:

Von innen nach außen;
von links nach rechts;
Argumentauswertung vor δ -Reduktion

Beachte, dass die Kurzbeschreibung nur informell ist und dass die Suche nach der Reduktionsstelle im Syntaxbaum jeweils von außen nach innen vorgeht.

Genauere Beschreibung anhand der Fälle:

- applikative Reihenfolge: Der Ausdruck t_0 ist auszuwerten: Folgende Fälle sind zu betrachten:
 - Der Ausdruck ist bereits ein Basiswert. Dann fertig.
 - Der Ausdruck ist eine Anwendung $s\ t$ und s ist kein Funktionsname und keine Anwendung. Dann wende applikative Reihenfolge auf s an.
 - Der Ausdruck ist eine Anwendung $f\ t_1 \dots t_n$ und f ein Funktionsname. Sei m die Stelligkeit von f . Wenn $m \leq n$, dann wende die applikative Reihenfolge der Auswertung auf den ersten Ausdruck t_i an, mit $i \leq m$, der noch kein Basiswert ist. Wenn alle t_i für $i \leq m$ Basiswerte sind, dann benutze δ -Reduktion. Wenn $n < m$, dann keine Reduktion.
 - Der Ausdruck ist `if b then e_1 else e_2` . Wenn b ein Basiswert ist, dann wende if-Reduktion an. Wenn b kein Basiswert, dann wende applikative Reihenfolge der Auswertung auf b an.
- normale Reihenfolge der Auswertung: Der Ausdruck t_0 ist zu transformieren.
 - Der Ausdruck ist bereits ein Basiswert. Dann fertig.
 - Der Ausdruck ist eine Anwendung $s\ t$ und s ist kein Funktionsname und keine Anwendung. Dann wende normale Reihenfolge auf s an.
 - Der Ausdruck ist eine Anwendung $f\ t_1 \dots t_n$ und f ist ein Funktionsname. Sei m die Stelligkeit von f . Wenn $m \leq n$, dann benutze δ -Reduktion. Ansonsten keine Reduktion.

- Der Ausdruck ist ein `if b then e1 else e2`. Wenn b ein Basiswert ist, dann wende if-Reduktion an. Wenn b kein Basiswert, dann wende normale Reihenfolge der Auswertung auf b an.
- Der Ausdruck ist ein arithmetischer Ausdruck $e_1 \text{ op } e_2$ (bzw. $\text{op } e$), so dass Stelligkeit von `op` 2 ist (bzw. 1 für $\text{op } e$). Wende von links nach rechts die normale Reihenfolge der Auswertung an.

In Beispiel 1.3.1 kann man die Auswertungen jetzt klassifizieren.

Die erste Reduktionsfolge ist in normaler Reihenfolge, Die dritte Reduktionsfolge ist in applikativer Reihenfolge. Die zweite ist weder das eine noch das andere d.h. es gibt auch andere Reduktionsfolgen, die zum richtigen Ergebnis führen. Die applikative Reihenfolge ist die „call by value“-Variante, während die normale Reihenfolge der entfernt der „call by name“-Auswertung entspricht.

Beispiel 1.3.2 Die Auswertung von `main` mit der Definition `main = fakultaet 4` ist eine Folge von Transformationen. Genauer: das Programm P_0 ist

```
main = fakultaet 4
fakultaet x = if x <= 1 then 1
              else x*(fakultaet (x-1))
```

Wir geben hier die **normale Reihenfolge der Auswertung** wieder. Im folgenden ist nur Transformation auf dem Ausdruck für `main` angegeben. Die verwendete Regel ist manchmal mit angegeben.

`fakultaet 4`

(Definitionseinsetzung)

`if 4 <= 1 then 1 else 4*(fakultaet (4-1))`

(Auswertung arithmetische Ungleichung)

`if False then 1 else 4*(fakultaet (4-1))`

(if-Auswertung)

`4*(fakultaet (4-1))`

(Arithmetik: zweites Argument von `*` wird ausgewertet per Transformation)

`4*(if (4-1) <= 1 then 1 else (4-1)*(fakultaet ((4-1)-1)))`

$(4 - 1)$ wird jetzt ausgewertet: erzwungen durch `if` und den Vergleich. In der puren Transformationssemantik werden die drei Ausdrücke $(4 - 1)$ als unabhängig betrachtet. In Implementierungen werden alle Ausdrücke $(4 - 1)$ gleichzeitig ausgewertet. Hier verwenden wir die Auswertungsschritte in der puren Transformationssemantik:

```

4*(if 3 <= 1 then 1 else (4-1)*(fakultaet ((4-1)-1)))

4*(if False then 1 else (4-1)*(fakultaet ((4-1)-1)))

4*((4-1)*(fakultaet ((4-1)-1)))

4*(3*(fakultaet ((4-1)-1)))

4*(3*(if ((4-1)-1) <= 1 then 1 else ((4-1)-1)*(fakultaet (((4-1)-1)-1))))

4*(3*(if (3-1) <= 1 then 1 else ((4-1)-1)*(fakultaet (((4-1)-1)-1))))

4*(3*(if 2 <= 1 then 1 else ((4-1)-1)*(fakultaet (((4-1)-1)-1))))

4*(3*(if False then 1 else ((4-1)-1)*(fakultaet (((4-1)-1)-1))))

4*(3*(((4-1)-1)*(fakultaet (((4-1)-1)-1))))
4*(3*((3-1)*(fakultaet (((4-1)-1)-1))))
4*(3*(2*(fakultaet (((4-1)-1)-1))))
4*(3*(2*( if (((4-1)-1)-1) <= 1 then 1
           else (((4-1)-1)-1)*(fakultaet (((4-1)-1)-1)-1))))
4*(3*(2*( if ((3-1)-1) <= 1 then 1
           else (((4-1)-1)-1)*(fakultaet (((4-1)-1)-1)-1))))
4*(3*(2*( if (2-1) <= 1 then 1
           else (((4-1)-1)-1)*(fakultaet (((4-1)-1)-1)-1))))

4*(3*(2*( if 1 <= 1 then 1
           else (((4-1)-1)-1)*(fakultaet (((4-1)-1)-1)-1))))
4*(3*(2*( if True then 1
           else (((4-1)-1)-1)*(fakultaet (((4-1)-1)-1)-1))))
4*(3*(2* 1))
4*(3*2)
4*6
24

```

Das sind 24 Auswertungsschritte.

Beispiel 1.3.3 Die **applikative Auswertung** von `main` mit der Definition `main = fakultaet 4` und gleichem Programm:

```

fakultaet 4

if 4 <= 1 then 1 else 4*(fakultaet (4-1))

if False then 1 else 4*(fakultaet (4-1))

```

```
4*(fakultaet (4-1))
```

(Arithmetik: zweites Argument von * wird ausgewertet; wegen der applikativen Reihenfolge: zuerst das Argument von fakultaet.)

```
4*(fakultaet 3)
```

```
4*(if 3 <= 1 then 1 else 3*(fakultaet (3-1)))
```

```
4*(if False then 1 else 3*(fakultaet (3-1)))
```

```
4*(3*(fakultaet (3-1)))
```

```
4*(3*(fakultaet 2))
```

```
4*(3*(if 2 <= 1 then 1 else 2*(fakultaet (2-1))))
```

```
4*(3*(if False then 1 else 2*(fakultaet (2-1))))
```

```
4*(3*(2*(fakultaet (2-1))))
```

```
4*(3*(2*(fakultaet 1)))
```

```
4*(3*(2*(if 1 <= 1 then 1 else 1*(fakultaet (1-1)))))
```

```
4*(3*(2*(if True then 1 else 1*(fakultaet (1-1)))))
```

```
4*(3*(2* 1))
```

```
4*(3*2)
```

```
4*6
```

```
24
```

Das sind 18 Auswertungsschritte.

Beispiel 1.3.4 Betrachte die Definitionen

```
main = const 5 (fakultaet 4)
fakultaet x = if x <= 1 then 1
              else x*(fakultaet (x-1))
const x y = x
```

Die Reduktionen von const 5 (fakultaet 4) unter applikativer Reihenfolge der Auswertung benötigt 19 Reduktionen:

$\text{const } 5 \text{ (fakultaet } 4) \xrightarrow{18} \text{const } 5 \text{ } 24 \xrightarrow{1} 5$

Die Reduktionen von const 5 (fakultaet 4) unter normaler Reihenfolge der

Auswertung benötigt nur 1 Reduktion:

`const 5 (fakultaet 4) $\xrightarrow{1}$ 5.`

Damit sieht man: die Reduktionslängen unter normaler und applikativer Reihenfolge sind i.a. nicht vergleichbar.

Es gilt:

Satz 1.3.5 *Unter der Bedingung, dass die Reduktion erfolgreich mit einem Basiswert terminiert, sind die Ergebnisse unter verschiedenen Strategien stets gleich.*

Das Typsystem und die zugehörige Typüberprüfung von Haskell sorgen dafür, dass Ausdrücke, die einen arithmetischen Typ haben bei Auswertung ein entsprechend getyptes Resultat liefern, falls die Auswertung ohne Fehler terminiert.

Eine optimale Zahl von Reduktionen bei der normalen Reihenfolge der Auswertung Transformationen wird erreicht, wenn man bei der normalen Reihenfolge darauf achtet, welche Unterausdrücke kopiert wurden, deren Transformationen parallel macht, und die parallel ausgeführten Reduktionen nur als eine Reduktion zählt. In Implementierungen wird dies dadurch erreicht, dass keine Kopie eines (auswertbaren) Ausdrucks gemacht wird, sondern intern ein gerichteter Graph statt eines Baumes erzeugt wird. Diese Auswertungsstrategie wird von Haskell verwendet. Sie wird *verzögerte Auswertung* oder auch *lazy evaluation* genannt. Die gemeinsam verwendeten Ausdrücke werden durch die Variablenvorkommen in den Rümpfen der Funktionen bestimmt. Zum Beispiel bei `quadrat x = x * x` wird das Argument t in einem Aufruf `quadrat t` gemeinsame verwendet

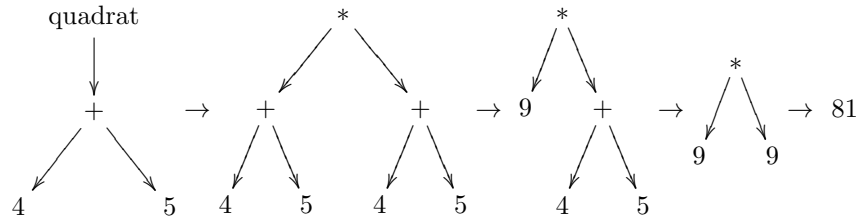
Es gilt:

Aussage 1.3.6 *Die verzögerte Reihenfolge der Auswertung ergibt eine optimale Anzahl von Reduktionen.*

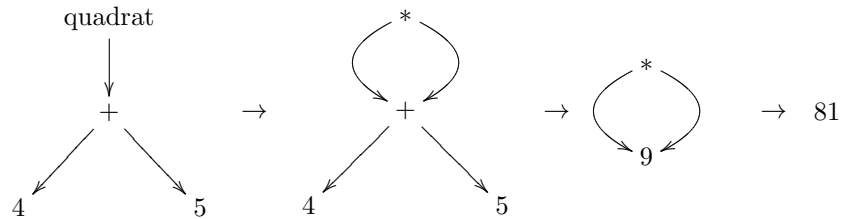
Aussage 1.3.7 *Wenn die applikative Reihenfolge keine unnötigen Argumente in Funktionsanwendungen reduziert, dann benötigt sie weniger Reduktionsschritte als die normale Reihenfolge der Auswertung.*

Im Beispiel `quadrat(4 + 5)` vergleichen wir mal die Reduktion in normaler Reihenfolge und in verzögerter Reihenfolge:

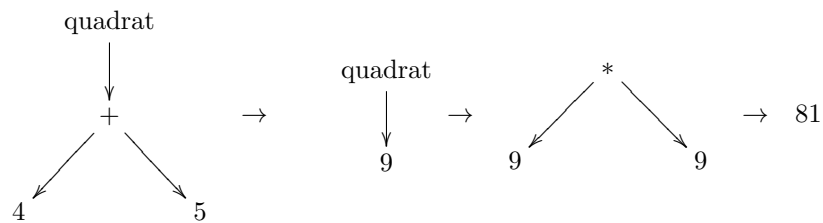
Beispiel 1.3.8 $\text{quadrat}(4 + 5) \rightarrow (4 + 5)^{(1)} * (4 + 5)^{(1)} \rightarrow 9 * 9 \rightarrow 81$, wobei mit $(4 + 5)^{(1)}$ angedeutet werden soll, dass dies intern derselbe Ausdruck ist.



und verzögerte Reihenfolge:



und applikative Reihenfolge:



Beispiel 1.3.9 Für das gleiche Beispiel wie oben geben wir noch die verzögerte Auswertung von `main` mit der Definition `main = fakultaet 4` an:

```
fakultaet 4
```

```
if 4 <= 1 then 1   else 4*(fakultaet (4-1))
```

```
if False then 1 else 4*(fakultaet (4-1))
```

```
4*(fakultaet (4-1))
```

```
4*(if (4-1) <= 1 then 1   else (4-1)*(fakultaet ((4-1)-1)))
```

```
4*(if 3  <= 1 then 1   else 3*(fakultaet (3-1)))
```

```
4*(if False then 1   else 3*(fakultaet (3-1)))
```

```
4*(3*(fakultaet (3-1)))
```

```
4*(3*(if (3-1) <= 1 then 1   else (3-1)*(fakultaet ((3-1)-1))))
```

```
4*(3*(if 2 <= 1 then 1   else 2*(fakultaet (2-1))))
```

```
4*(3*(if False then 1   else 2*(fakultaet (2-1))))
```

```
4*(3*(2*(fakultaet (2-1))))
```

```
4*(3*(2*( if (2-1) <= 1 then 1   else (2-1)*(fakultaet ((2-1)-1)))))
```

```

4*(3*(2*( if 1 <= 1 then 1 else 1*(fakultaet (1-1)))))
4*(3*(2*( if True then 1 else 1*(fakultaet (1-1)))))
4*(3*(2* 1))
4*(3*2)
4*6
24

```

Das sind 18 Schritte, genau wie bei der applikativen Auswertung.

Ein Vorteil der verzögerten Auswertung gegenüber der applikativen besteht darin, dass Programmtransformationen die zur Compilezeit mittels verzögerter Auswertung durchgeführt werden, korrekt sind im Sinne der operationalen Semantik. Das gilt nicht für die applikative Reihenfolge der Auswertung, d.h. auch nicht für Programmiersprachen, die die applikative Reihenfolge der Auswertung verwenden, da dadurch terminierende in nicht-terminierende Programme verwandelt werden können.

Folgende Tabelle gibt eine Aufstellung der benötigten Anzahl der Reduktionen zu den Beispielvarianten.

	verzögerte R.	applikative R.	normale R.
(fakultaet 4)	18	18	24
main	2	20	2
wobei	main = const 5 (fakultaet 4)		

1.3.2 Nachweis der Anzahl der Reduktionsschritte mit vollständiger Induktion

Wir zeigen als Beispiel, dass die Anzahl der Reduktionsschritte von (fakultaet n) für $n \geq 2$ unter verzögerter Reduktion genau $5 * (n - 1) + 3$ sind.

Aussage 1.3.10 *Die Anzahl der Reduktionsschritte von (fakultaet n) für $n \geq 2$ unter verzögerter Reduktion ist genau $5 * (n - 1) + 3$.*

Beweis Wir beziehen uns auf die Funktionsdefinition

```

fakultaet x = if x <= 1 then 1
              else x*(fakultaet (x-1))

```

Zuerst zeigen wir mit vollständiger Induktion, dass der Aufruf fakultaet ($n-1$) für $n \geq 2$ unter verzögerter Reihenfolge der Reduktion $5 * (n - 2) + 4$ Schritte benötigt:

Basis Wenn $n = 2$ ist, dann sind die Reduktionen wie folgt:


```
fakultaet (2-1)
if (2-1) <= 1 then 1 else ...
if 1 <= 1 then 1 else ...
if True then 1 else ...
1
```

Das sind 4 Reduktionsschritte.

Da $5 * (2 - 2) + 4 = 4$, gilt die Formel in diesem Fall.

Induktionsschritt Wenn $n > 2$ ist, dann sind die Reduktionen wie folgt:

```
fakultaet (n-1)
if (n-1) <= 1 then ... (n-1)
if n1 <= 1 then ... (n1) -- n1 ist Basiswert > 1
if False then ...
n1*fakultaet (n1-1) -- Wert n2 als Ind-hypothese berechnet
n1*n2 -- Produkt-Berechnung zaehlt noch dazu
n3
```

Das sind $5 + 5 * (n1 - 2) + 4 = 5 * (n - 2) + 4$ Reduktionsschritte

Jetzt fehlt noch der eigentlich Aufruf. Der Aufruf (`fakultaet 1`) benötigt drei Schritte, und der Aufruf (`fakultaet n`) benötigt folgende Schritte:

```
fakultaet n
if n <= 1 then ... -- n ist Basiswert > 1
if False then ...
n*fakultaet (n-1) -- Hier Induktionshypothese
n*n2 -- Produkt-Berechnung zaehlt noch dazu
n3
```

Das sind 4 Schritte für den Aufruf, und $5 * (n - 2) + 4$ für `fakultaet (n-1)`. Zusammen sind es $5 * (n - 1) + 3$ Schritte. \square

1.4 Rekursive Auswertung in Haskell

Wir haben in Beispielen schon rekursive Funktionen definiert und ausgewertet. In diesem Abschnitt sollen einige unterschiedliche Formen der Auswertungsprozesse, die durch Rekursion bewirkt werden, klassifiziert werden. Zur Vereinfachung und Analyse wird eine feste rekursive Funktion f betrachtet und die Auswertung entsprechend analysiert. Ein Unterausdruck ($f\ t_1 \dots t_n$) im Rumpf von f wird manchmal auch als *rekursiver Aufruf* bezeichnet.

Für die folgenden Betrachtungen verwenden wir die **applikative Reihenfolge der Auswertung**. Diese kann man im Haskell-Programm auch erreichen, indem man die Funktionen so definiert, dass die applikative Reihenfolge der Auswertung erzwungen wird.

Wenn man dies in Haskell definieren will, kann man die eingebaute Funktion `seq` verwenden: `seq s t` wertet zuerst `s` aus und dann `t` und gibt den Wert von `t` zurück.

Man kann zu jeder Funktion `f` eine applikative Variante `f_applikativ` definieren, die zuerst die Argumente auswertet.

Vorsicht: Es gilt aber i.a., dass die Funktionen `f` und `f_applikativ` semantisch verschieden sind: sie haben evtl. verschiedene Terminierungseigenschaft.

Die Fakultätsfunktion mit der mathematischen Definition

$$\begin{aligned} 0! &:= 1 \\ n! &:= n * (n-1)! && \text{wenn } n > 0 \end{aligned}$$

haben wir schon in Haskell definiert:

```
fakultaet x = if x <= 1 then 1
              else x*(fakultaet (x-1))
```

Die (applikative) Auswertung können wir veranschaulichen, wenn wir die Folge der Transformationen betrachten, die zum Ergebnis führt, einige Zwischenschritte auslassen, und nur den Ausdruck unmittelbar vor dem nächsten rekursiven Aufruf notieren. In der Sprache der Transformationen: wenn der nächste Ausdruck `fakultaet n` transformiert wird.

```
(fakultaet 6)
(6 * (fakultaet (6-1)))
(6 * (5 * (fakultaet (5-1))))
(6 * (5 * (4 * (fakultaet (4-1)))))
(6 * (5 * (4 * (3 * (fakultaet (3-1)))))
(6 * (5 * (4 * (3 * (2 * (fakultaet (2-1))))))
(6 * (5 * (4 * (3 * (2 * 1)))))
(6 * (5 * (4 * (3 * 2))))
(6 * (5 * (4 * 6)))
(6 * (5 * 24))
(6 * 120)
720
```

Dieses Verhalten eines Auswertungsprozesses nennt man auch linear rekursiv. Charakteristisch ist, dass jeweils nur eine rekursive Funktionsanwendung (der betrachteten Funktion) auftritt, der Gesamtausdruck aber beliebig groß werden kann.

Eine alternative Berechnung der Fakultätsfunktion erhält man, wenn man folgende Ersetzung iteriert, und dann entsprechend programmiert.

```
Produkt  ⇒  Produkt * Zähler
Zähler   ⇒  Zähler + 1
```

```
fakt_iter produkt zaehler max =
  if zaehler > max
  then produkt
  else fakt_iter (zaehler * produkt) (zaehler + 1) max

fakultaet_lin n = fakt_iter 1 1 n
```

Die Definition der strikten Variante **fakt_iter_strikt** der Funktion **fakt_iter**, die unter Normalordnung das gleiche Verhalten zeigt, ist im Programmfile angegeben.

Der Auswertungsprozess sieht folgendermaßen aus, wenn man einige Zwischenschritte weglässt:

```
(fakultaet_lin 6)
(fakt_iter 1 1 6)
(fakt_iter 1 2 6)
(fakt_iter 2 3 6)
(fakt_iter 6 4 6)
(fakt_iter 24 5 6)
(fakt_iter 120 6 6)
(fakt_iter 720 7 6)
720
```

Diese Art eines Auswertungs-Prozesses nennt man end-rekursiv (tail recursive):

Genauer: wenn die Auswertung eines Ausdrucks $(f\ a_1 \dots a_n)$ nach einigen Auswertungen (unter applikativer Reihenfolge der Auswertung) als Resultat den Wert eines Ausdruck der Form $(f\ b_1 \dots b_n)$ auswerten muss; hierbei können die Argumente a_i, b_i Ausdrücke sein. Dies muss für die gesamte Rekursion gelten.

Charakteristisch ist, dass man als Zwischenaufgabe nur hat, einen Aufruf von $(f\ b_1 \dots b_n)$ auszuwerten, und am Ende nur den Wert des letzten rekursiven Ausdrucks braucht. Es ist nicht nötig, die Rekursion rückwärts wieder aufzurollen und den Wert an den ersten Aufruf zurückzugeben, da keine Berechnungen mehr stattfinden.

Endrekursion in imperativen Programmiersprachen ist normalerweise nicht optimiert, so dass man bei einer Rekursion der Tiefe n am Ende n Rückgabeschritte hat und auch entsprechend viel Platz benötigt. Bei optimierter Endrekursion spricht man auch von einem iterativen Prozess bzw. von **Iteration**. Genauer: wenn die Auswertung eines Ausdrucks $(f\ a_1 \dots a_n)$ nach einigen Auswertungen als Wert einen Ausdruck der Form $(f\ b_1 \dots b_n)$ auswerten muss. Hierbei müssen die Argumente a_i, b_i jeweils Basiswerte sein. Dies muss für die gesamte Rekursion gelten. Der obige Auswertungsprozess ist somit iterativ.

In imperativen Programmiersprachen gibt es im allgemeinen mehrere Programmkonstrukte, mit denen man Iteration ausdrücken kann: **for ...do**, **while**, oder **repeat ...until**. Diese sind im Falle einer fehlenden Optimierung

der Endrekursion auch notwendig, damit man effizient eine Iteration programmieren kann.

Bei der Verwendung von Haskell ist zu beachten, dass die verzögerte Auswertung verwendet wird, die zwar höchstens soviele Reduktionsschritte wie die applikative Reihenfolge benötigt, aber ohne die richtigen Striktheitsanweisung oft einen linear rekursiven Prozess erzeugt statt eines iterativen Prozesses.

In Haskell bzw. unter verzögerter Reihenfolge nennt man eine Funktion f linear rekursiv, wenn die Ausdrücke, die aus $f\ t_1 \dots t_n$ (wobei t_i Basiswerte sind) bei Auswertung entstehen, höchstens einen rekursiven Aufruf von f enthalten. In Haskell bzw. unter verzögerter Reihenfolge nennt man eine Funktion f endrekursiv, wenn die Ausdrücke, die aus $f\ t_1 \dots t_n$ (wobei t_i Basiswerte sind) bei Auswertung entstehen, die Definitionseinsetzung für f nur dann stattfindet, wenn der Ausdruck die Form $f\ t'_1 \dots t'_n$ hat.

In Haskell bzw. unter verzögerter Reihenfolge nennt man eine Funktion *iterativ*, wenn sie unter applikativer Auswertungsreihenfolge einen iterativen Prozess erzeugt.

Beispiel 1.4.1 *Die Auswertung des Ausdrucks (fakultaet_lin 5) hat bei verzögerter Auswertung folgende Zwischenstufen.*

```
(fakultaet_lin 5)
(fakt_iter 1 1 5)
(fakt_iter (1*1) (1+1) 5)
(fakt_iter (2*(1*1)) (2+1) 5)
(fakt_iter (3*(2*(1*1))) (3+1) 5)
(fakt_iter (4*(3*(2*(1*1)))) (4+1) 5)
(fakt_iter (5*(4*(3*(2*(1*1))))) (5+1) 5)
(5*(4*(3*(2*(1*1)))))
120
```

Unter verzögerter Reihenfolge der Auswertung ist das eine lineare Rekursion, es ist auch eine Endrekursion, aber die Rekursion selbst ist nicht iterativ, da die Argumente wachsen und auch keine Basiswerte sind. Die Funktion fakultaet_iter ist aber trotzdem iterativ:

Die Auswertung in applikativer Reihenfolge ist iterativ:

```
(fakultaet_lin 5)
(fakt_iter 1 1 5)
(fakt_iter 1 2 5)
(fakt_iter 2 3 5)
(fakt_iter 6 4 5)
(fakt_iter 24 5 5)
(fakt_iter 120 6 5)
120
```

Beispiel 1.4.2 Zur Berechnung der rekursiven Fibonacci-Funktion.

1, 1, 2, 3, 5, 8, 13, 21, ...

Die Fibonacci Zahlen sind definiert durch folgende rekursive Gleichung:

$$Fib(n) := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sonst} \end{cases}$$

Die naive rekursive Programmierung der Funktion ergibt:

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

Bei der Berechnung von `fib 5` ergibt sich folgende Zwischenberechnungen:

```
fib 5 ....
fib 4 + fib 3
(fib 3 + fib 2) + (fib 3)
((fib 2 + fib 1) + fib 2) + (fib 2 + fib 1)
(((fib 0 + fib 1) + fib 1) + (fib 0 + fib 1))
+ ((fib 0 + fib 1) + fib 1)
```

Das ergibt folgende Statistik:

```
fib 3 wird 2 mal berechnet;
fib 2 wird 3 mal berechnet;
fib 1 wird 5 mal berechnet.
```

Eine Analyse ergibt, dass bei Berechnung von `fib n` für $n \geq 2$ der Aufruf `fib 1` jeweils `(fib n)`-mal erfolgt.

Es gilt, dass $fib\ n \approx \frac{\Phi^n}{\sqrt{5}}$ wobei $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$

Der Wert von `fib` wächst somit exponentiell. Die Anzahl der Reduktionen der Funktion `fib` ist ebenfalls exponentiell abhängig von n , d.h.

Optimierung:

Die Berechnung von `fib` kann beschleunigt werden durch eine andere Berechnungsidee. Eine Beobachtung dazu ist, dass man zur Berechnung von `fib(n)` höchstens die Werte `fib(i)` benötigt für alle $1 \leq i \leq n$. D.h. die sukzessive Berechnung einer Wertetabelle ergibt bereits einen Algorithmus, der schneller ist als `fib`.

Eine einfache, verbesserte Variante ist die Idee, jeweils nur `fib(n-1)` und `fib(n-2)` zu speichern und daraus den nächsten Wert `fib(n)` zu berechnen.

Rechenvorschrift: $(a, b) \rightarrow (a + b, a)$

```

fib_lin n = (fib_iter 1 0 n)

fib_iter a b zaehler =
    if zaehler <= 0
    then b
    else fib_iter (a + b) a (zaehler - 1)

```

Prozess für `(fib_lin 5)` bei applikativer Auswertung:

```

(fib_lin 5)
(fib_iter 1 0 5)
(fib_iter 1 1 4)
(fib_iter 2 1 3)
(fib_iter 3 2 2)
(fib_iter 5 3 1)
5

```

Offenbar benötigt diese Version von `fib` linear viele Reduktionen in Abhängigkeit von n , und die Größe aller Ausdrücke, die während des Prozesses erzeugt werden, hat ein obere Schranke, also ist der Platzbedarf konstant (d.h. unabhängig) von n . Allerdings nur, wenn man die Größe der Darstellungen der Zahlen vernachlässigt.

D.h. dieser Prozess ist somit **iterativ**.

1.4.1 Baumrekursion

Wir nehmen als Beispiel dafür die Berechnung der Fibonacci-Zahlen, auch wenn diese Berechnung effizienter programmiert werden kann. Baumrekursion entspricht den primitiv rekursiven Funktionen.

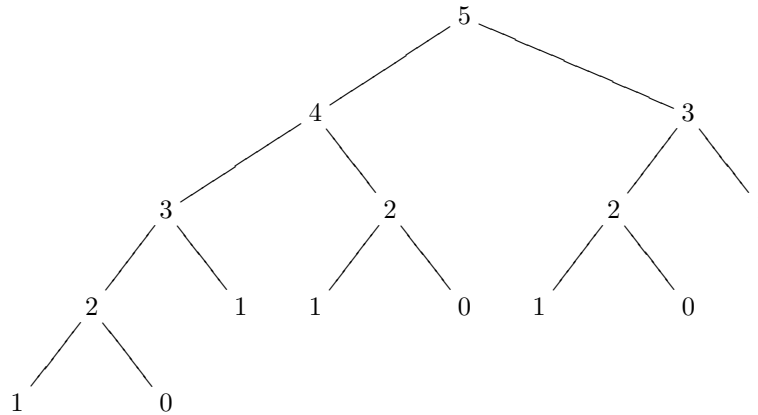
Beispiel 1.4.3 Berechnung der Fibonacci Zahlen:

Der induzierte Auswertungsprozess ergibt folgende Ausdrücke, wobei wir die applikative Reihenfolge der Auswertung verwenden:

```

fib 5
fib 4 + fib 3
(fib 3 + fib 2) + fib 3
((fib 2 + fib 1) + fib 2) + fib 3
(((fib 1 + fib 0) + fib 1) + fib 2) + fib 3
(((1+0) + fib 1) + fib 2) + fib 3
((1 + fib 1) + fib 2) + fib 3
((1+1) + fib 2) + fib 3
(2 + fib 2) + fib 3
(2 + (fib 1 + fib 0)) + fib 3
.....

```



Einen solchen Auswertungsprozess nennt man auch baumrekursiv (Baumrekursion). Charakteristisch ist, dass die Ausdrücke unbegrenzt wachsen können und dass mehrere rekursive Aufrufe vorkommen. Diese sollten nicht geschachtelt sein, d.h. in den Argumenten einer rekursiven Funktion sollten keine rekursiven Funktionsaufrufe mehr vorkommen.

Bemerkung 1.4.4 Geschachtelte Baumrekursion

Der allgemeine Fall ist geschachtelte Baumrekursion, wobei nicht nur rekursive Aufrufe vorkommen, sondern diese auch in den Argumenten wieder rekursive Aufrufe derselben Funktion enthalten können. Diese Form der Rekursion wird selten benötigt, denn die damit definierten Funktionen sind i.a. nicht effizient berechenbar. Ein Beispiel ist die Ackermannfunktion, die folgendermaßen definiert ist:

```

----- Ackermanns Funktion -----
ack 0 y = 1
ack 1 0 = 2
ack x 0 | x >= 2 = x+2
ack x y | x > 0 && y > 0 = ack (ack (x-1) y) (y-1)
  
```

Hier wird eine spezielle Programmiertechnik von Haskell verwendet: Es werden mehrere Definitionsgleichungen pro Funktion verwendet. Von oben nach unten wird probiert, welche Definitionsgleichung (links vom Gleichheitszeichen) passt:

- 1) Argumente anpassen, evtl. auswerten
- 2) Bedingung rechts vom | prüfen

Bei komplizierteren Datenstrukturen als Argumente wird Pattern-Matching verwendet (siehe XXX)

```

----- Ackermanns Funktion optimiert -----
ackopt 0 y = 1
ackopt 1 0 = 2
ackopt x 0 = x+2
ackopt x 1 = 2*x
ackopt x 2 = 2^x
ackopt x y | x > 0 && y > 0 = ackopt (ackopt (x-1) y) (y-1)

```

Was erhalten wir bei der Eingabe (ackopt 5 3):

```

*Main> logI10 (ackopt 5 3)
19728.301029995662

```

Die Zahl 19728 ist die Anzahl Dezimalstellen von $(\text{ackopt } 5 \ 3) = 2^{65536}$.

Der Aufruf (ack 4 4) ist ein Exponentialterm $2^{2^{\dots}}$ der Höhe 65536. Diese Zahl kann man nicht mehr in vernünftiger Weise in Ziffern berechnen.

Diese Funktion wächst sehr schnell und benötigt sehr viele Ressourcen zur Berechnung. Sie wurde konstruiert, um nachzuweisen, dass es sehr schnell wachsende Funktionen gibt, die nicht primitiv rekursiv sind, d.h. nicht mit einem speziellen Rekursions-Schema definiert werden können.² Sie hat auch Anwendung in der theoretischen Informatik (Komplexitätstheorie).

Tabellarische Darstellung der verschiedenen rekursiven Prozesse.

↑ „impliziert“	geschachtelt baumrekursiv	mehrere rekursive Unterausdrücke auch in den Argumenten der rekursiven Unterausdrücke erlaubt
	baumrekursiv	mehrere rekursive Unterausdrücke erlaubt, aber Argumente der rekursiven Unterausdrücke ohne weitere Rekursion
	linear rekursiv	maximal ein rekursiver Unterausdruck
	endrekursiv	linear rekursiv und Gesamtergebn ist Wert des rekursiven Unterausdrucks
	iterativ	endrekursiv und Argumente des rekursiven Unterausdrucks sind Basiswerte

1.5 Komplexitäten von Algorithmen und O-Schreibweise

Die O-Schreibweise wird verwendet, um die asymptotische Größenordnung von numerischen Funktionen nach oben abzuschätzen. Wir verwenden sie, um die

²Die Einschränkung kann man informell so beschreiben: es ist eine lineare Rekursion, in der man zur Wertberechnung nur Addition und als Konstante nur die 1 verwenden darf. Man darf auch Zählervariablen verwenden, und man darf primitiv rekursiv definierte Funktionen wiederverwenden. Addition, Multiplikation, Potenz, Fibonacci usw. können alle primitiv rekursiv definiert werden.

Laufzeit, d.h. die Anzahl der Reduktionen, eines Algorithmus in Abhängigkeit von der Eingabegröße asymptotisch nach oben abzuschätzen. Teilweise kann man auch eine optimale asymptotische Abschätzung angeben. Im folgenden meinen wir im allgemeinen diese optimale Abschätzung. Damit kann man z.B. Algorithmen und Problemklassen bzgl der worst-case-Laufzeit (bzw. best-case-, average-case-) grob klassifizieren.

Beispiel 1.5.1 Wenn man `fib` in der Größe der eingegebenen Zahl n , misst bzw. in der Größe der Darstellung der Zahl d , und die Anzahl der Reduktionen eines Algorithmus `alg` bei einer Eingabe der Größe n als red_{alg} notiert, dann gilt

$$\begin{aligned} red_{fib}(n) &= O(1.62^n) \\ red_{fib_lin}(n) &= O(n) \\ red_{fib_lin}(d) &= O(10^d) \end{aligned}$$

Es gibt einige Komplexitäten von Algorithmen, die häufig vorkommen. Die Sprechweisen sind:

$O(1)$	konstant
$O(\log(n))$	logarithmisch
$O(n)$	linear
$O(n * \log(n))$	fastlinear (oder auch n-log-n)
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^k)$	polynomiell
$O(c^n)$	exponentiell für eine Konstante $c > 1$.

Hier eine beispielhafte Tabelle zur intuitiven Veranschaulichung der unterschiedlichen Komplexitäten.

Eingabedaten	10	100	1000
Algorithmus			
$\log_2(n)$	0.000003 sec	0.000007 sec	0.00001
n	0.00001 sec	0.0001 sec	0.001 sec
n^2	0.0001 sec	0.01 sec	1 sec
n^3	0.001 sec	1 sec	15 min
2^n	0.001 sec	$4 * 10^{16}$ Jahre	nahezu unendlich

1.5.1 Optimierung am Beispiel der Potenzberechnung

Beispiel 1.5.2 Berechnung von Potenzen b^n für positive ganze Zahlen n . Die Potenzen sind rekursiv definiert durch

$$b^n := \begin{cases} 1 & \text{falls } n = 0 \\ b * b^{n-1} & \text{sonst} \end{cases}$$

Direkte Kodierung des Algorithmus ergibt ein rekursives Programm:

```

potenz b n = if n == 0
              then 1
              else b * (potenz b (n - 1))

```

Platz- und Zeitbedarf sind von der Größenordnung $O(n)$. Der Zeitbedarf lässt sich verbessern, indem man folgende Idee ausnutzt: statt $b^8 = b*b*b*b*b*b*b*b$ berechne

- $b^2 = b * b$
- $b^4 = b^2 * b^2$
- $b^8 = b^4 * b^4$

Als allgemeine Rechenvorschrift halten wir fest:

$$b^n := \begin{cases} 1 & \text{falls } n = 0 \\ (b^{n/2})^2 & \text{falls } n \text{ gerade und } \geq 2 \\ b * b^{n-1} & \text{falls } n \text{ ungerade} \end{cases}$$

```

potenz_log b n = if n == 0 then 1
                  else if even n
                        then quadrat (potenz_log b (n 'div' 2))
                        else b * (potenz_log b (n - 1))

```

Der Platz- und Zeitbedarf in der Zahl n ist $O(\log(n))$, z.B. für $n = 1000$ benötigt dieser Algorithmus nur 14 Multiplikationen. Beachtet man noch den extra Aufwand für die Multiplikationen, die bei beliebigen langen Zahlen (Integer) mit der Länge der Zahl steigt, so ist der wahre Aufwand höher: je nachdem wie diese Multiplikation implementiert ist. Bei normaler Implementierung: $O((\log(n))^3)$,

1.5.2 Der größte gemeinsame Teiler

Idee zur Berechnung von $\text{ggT}(a, b)$ (Euklids Algorithmus)

Teile a durch b gibt Rest r ,
wenn $r = 0$, dann $\text{ggT}(a, b) := b$
wenn $r \neq 0$, dann berechne rekursiv $\text{ggT}(b, r)$.

Beispiel 1.5.3 $\text{ggT}(30, 12) = \text{ggT}(12, 6) = 6$

```

ggT a b = if b == 0
           then a
           else ggT b (rem a b)

```

Es gilt:

Satz 1.5.4 (*Lamé, 1845*): Wenn der Euklidische ggt-Algorithmus k Schritte benötigt, dann ist die kleinere Zahl der Eingabe $\geq \text{fib}(k)$.

Daraus kann man sofort herleiten, dass dieser Algorithmus Platz- und Zeitbedarf in der Größenordnung $O(\log(n))$ hat: Wenn n die kleinere Zahl ist und der Algorithmus k Schritte benötigt, dann ist $n \geq \text{fib}(k) \approx \Phi^k$. Also ist $k = O(\log(n))$. D.h. der Euklidische ggt-Algorithmus benötigt (im schlimmsten Fall) $O(\log(n))$ Schritte bei Eingabe der Zahl n .

Komplexitäten von Algorithmen:

Aufruf	Zeitaufwand - Abschätzung	
Arithmetische Operationen als $O(1)$ angenommen		
fakultaet n	$O(n)$	
fib n	$O(1, 62^n)$	
fib_lin n	$O(n)$	
ggt m n	$O(\log(\max(m, n)))$	
$m + n$	$O(1)$	$m, n :: \text{Int}$
$m * n$	$O(1)$	$m, n :: \text{Int}$
quadratsumme m n	$O(1)$	$m, n :: \text{Int}$
Arithmetische Operationen auf großen Zahlen		
$m + n$	$O(\log(m + n))$	$m, n :: \text{Integer}$
$m * n$	$O(\log(m + n))$	$m, n :: \text{Integer}$
quadratsumme m n	$O(\log(m + n))$	$m, n :: \text{Integer}$

1.6 Funktionen auf Listen

Wir führen Listen und Funktionen auf Listen hier ein, um jetzt schon komplexere Funktion schreiben zu können, die Besprechung von Datentypen, insbesondere eine genauere des Datentyps Liste, ist im nächsten Abschnitt.

Listen sind eine Datenstruktur für Folgen von gleichartigen, gleichgetypten Objekten.
Der Typ einer Liste ist von der Form $[a]$, wobei a der Typ der Elemente ist.

Beispiel 1.6.1 $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ ist die Liste der 10 Ziffern. Der Typ der Liste ist `[Integer]`; d.h. Liste von `Integer`.

$[]$ ist die Liste ohne Elemente (leere Liste, `Nil`); deren Typ ist $[a]$, wobei a eine Typvariable ist.

$['a', 'b']$ ist eine Liste mit den 2 Zeichen 'a' und 'b'. Der Typ ist `[Char]`; d.h. Liste von `Char`, auch abgekürzt als `String`.

Die Liste $['a', 'b']$ wird auch als "ab" angezeigt, d.h. als `String`.

Man kann auch Listen von Listen verwenden:

$[[], [0], [1, 2]]$ ist eine Liste mit drei Elementen. Der Typ ist `[[Integer]]`, d.h. eine Liste von Listen von `Integer`-Objekten.

In Haskell sind auch potentiell unendliche Listen möglich und erlaubt:

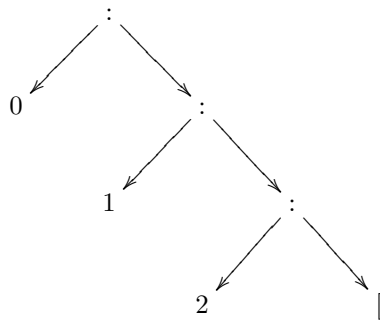
Der Haskell-Ausdruck $[1..]$ erzeugt nacheinander alle natürlichen Zahlen:

$[1, 2, 3, 4, 5, \dots]$

Die komplette Auswertung der Liste, um diese anzuzeigen, terminiert natürlich nicht. Wir werden das Programmieren von Strömen anhand von unendlichen Listen demonstrieren.

Es gibt zwei verschiedene Schreibweisen für Listen: $[0, 1, 2]$ wird auch als $(0 : (1 : (2 : [])))$ geschrieben. Die letzte Darstellung entspricht der internen Darstellung, Beide sind in Haskell intern vollkommen gleich dargestellt. Die interne Darstellung wird mit den zwei Konstruktoren `:` und `[]` aufgebaut. Hierbei ist `:` ein zweistelliger Konstruktor, der im Haskellprogramm infix geschrieben wird, und `[]` eine Konstruktorkonstante.

Eine Baumdarstellung der Liste $(0 : (1 : (2 : [])))$ sieht so aus:



Eingebaute, listenerzeugende Funktionen sind:

- `[n..]` erzeugt die Liste der Zahlen ab n .
- `[n..m]` erzeugt die Liste der Zahlen von n bis m
`[1..10]` ergibt `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
- `[n,m..]` erzeugt die Liste der Zahlen ab n mit Abstand $m - n$.
- `[n,m..k]` erzeugt die Liste der Zahlen von n bis k mit Abstand $m - n$.

Es gibt vordefinierte Funktionen auf Listen in Haskell. Die sind aber vom Verhalten her nicht zu unterscheiden von selbst definierten Funktionen.

Oft sind die Definitionen der Listen-Funktionen aufgeteilt in zwei definierende Gleichungen: Es werden die Fälle der Struktur des Listen-Arguments unterschieden: Das Listenargument kann die leere Liste sein oder eine nichtleere Liste. Wenn es mehr Listenargumente sind, können es auch entsprechend mehr Fälle sein.

Ein erstes Beispiel ist die Funktion zum Berechnen der Länge einer Liste: Da `length` die vordefinierte Funktion zum Berechnen der Länge einer Liste ist, nehmen wir einen anderen Namen.

```
lengthr []      = 0
lengthr (_:xs) = 1 + lengthr xs
```

Der Typ der Längenfunktion ist `lengthr :: [a] -> Int`

Die Funktion `map` wendet eine Funktion f auf alle Elemente einer Liste an.

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

Die erste Funktion `map` wendet eine Funktion auf jedes Listenelement an und erzeugt die Liste der Resultate. Ihr Typ ist `map :: (a->b) -> [a] -> [b]`, wobei a, b Typvariablen sind. D.h. man kann jeden Typ für a, b einsetzen. Verwendet man `map` beim Programmieren, so erzwingt der Typ bestimmte Eigenschaften der Typen der Argumente:

Ist z.B. das erste Argument die Funktion `quadrat :: Integer->Integer`, dann muss das zweite Argument, die Liste, vom Typ `[Integer]` sein, und die Ausgabe wird vom Typ `[Integer]` sein.

Nimmt man die Funktion `ord :: Char -> Int`, dann ist das zweite Argument eine Liste vom Typ `[Char]`, d.h. `String`, und die Ausgabe ist eine Liste vom Typ `[Int]`.

Da man Typen auch schachteln kann, noch ein komplizierteres Beispiel: `map length xs`. Hierbei erzwingt der Typ von `map`, `length`, dass der Typ von `xs` gerade `[[a]]` ist, d.h. eine Liste von Listen wobei der Typ der inneren Elemente keine Rolle spielt, und die Ausgabe ist vom Typ `[Int]`.

Die Ausdrücke `[]` und `(x:xs)` nennt man **Muster**. Man kann sie analog zu formalen Parametern verwenden, wobei die Voraussetzung ist, dass das Argu-

ment zum entsprechenden Muster passt, evtl. nach einigen Auswertungsschritten. Zum Beispiel wird `map quadrat (1:[])` per Definitionseinsetzung unter Benutzung der zweiten Gleichung zu `f 1 : map f []` reduziert, wobei 1 für `x` und `[]` für `xs` eingesetzt wurde.

Ein weiteres Beispiel ist:

```
map (+ 1) [1..10]
```

Diese Funktion addiert zu jedem Listenelement die Zahl 1. Das Ergebnis ist die Liste `[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]`.

Strings oder Zeichenketten in Haskell sind Listen von Zeichen und können auch genauso verarbeitet werden.

Die folgende Funktion hängt zwei Listen zusammen:

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

In Haskell wird diese Funktion als `++` geschrieben und Infix benutzt.

Beispiel 1.6.2

```
*Main> 10:[7,8,9]
[10,7,8,9]
*Main> length [3,4,5]
3
*Main> length [1..1000]
1000
*Main> let a = [1..] in (length a,a)
ERROR - Garbage collection fails to reclaim sufficient space
Main> map quadrat [3,4,5]
[9,16,25]
*Main> [0,1,2] ++ [3,4,5]
[0,1,2,3,4,5]
```

Weitere wichtige Funktionen auf Listen sind:

Filtern von Elementen aus einer Liste:

```
filter f []      = []
filter f (x:xs)  = if (f x) then x : filter f xs
                  else filter f xs
```

Die ersten n Elemente der Liste `xs`:

```
take 0 _        = []
take n []       = []
take n (x:xs)   = x : (take (n-1) xs)
```

Die verzögerte Auswertung wird so gesteuert, dass man bei einem Ausdruck der Form $f\ s_1 \dots s_n$ die Definitionseinsetzung erst macht, wenn die Argumente bei denen eine Fallunterscheidung notwendig ist, ausgewertet sind. Diese Auswertung erfolgt von links nach rechts. Es wird nur soviel von den Argumenten ausgewertet wie nötig, um die Fallunterscheidung machen zu können.

Ein Beispiel zur Reihenfolge der Auswertung von Ausdrücken mit `take`:

```
repeat x = x : repeat x
```

Auswertung:

```
take 10 (repeat 1)
take 10 (1:repeat 1)
1:(take (10-1) (repeat 1))      ## (x:xs) = (1:(repeat 1))
1:(take 9 (repeat 1))
1:(take 9 (1:(repeat 1)))
1:(1:(take (9-1) (repeat 1)))
...
1:(1: ... 1:(take (1-1) (repeat 1))
1:(1: ... 1:(take 0 (repeat 1)))      ## n = 0
1:(1: ... 1:[]
```

1.6.1 Rekursionsformen mit Listenargumenten

Bei Verwendung von Listenargumenten bleiben die Definitionen der Begriffe linear rekursiv, end-rekursiv, Baum-rekursiv und verschachtelt Baum-rekursiv unverändert, jedoch muss der Begriff iterativ angepasst werden.

Wir sprechen von einem **iterativen Auswertungsprozess**, wenn die Auswertung eines Ausdrucks $(f\ a_1 \dots a_n)$ nach einigen Auswertungen als (Rückgabe-)Wert einen Ausdruck der Form $(f\ b_1 \dots b_n)$ auswerten muss, und dies für die ganze Rekursion gilt. Hierbei muss für die Argumente pro Index i und für die gesamte Rekursion folgendes gelten: Entweder sind a_i , b_i Basiswerte, oder a_i , b_i sind komplett ausgewertete, endliche Listen.

Von einer *iterativen Version* f_{iter} einer rekursiven Funktion f spricht man, wenn f und f_{iter} bei Eingabe von Basiswerten bzw. endlichen, komplett ausgewerteten Listen, die gleichen Ergebnisse berechnen, und f_{iter} bei Eingabe von endlichen, ausgewerteten Listenargumenten einen iterativen Prozess erzeugt.

Bei Listenargumenten spricht man von

ausgewerteten Argumenten, wenn das Argument soweit ausgewertet ist, dass man die Fallunterscheidung machen kann, und von

komplett ausgewerteten Argumenten, wenn die Argumente nur aus Konstrukturen und Abstraktionen (d.h. Funktionsnamen) bestehen

Die applikative Reihenfolge der Auswertung wertet Listen komplett aus, aber wertet Funktionen, lambda-Ausdrücke nicht weiter aus.

Bei der applikativen Reihenfolge der Auswertung von `(f s1 ... sn)` werden alle Listenargumente `si` vollständig ausgewertet, bevor die Funktion `f` auf die Argumente angewendet wird.

Folgende Version der `length`-Funktion benutzt die iterative Funktion `length_linr`:

```
length_lin xs      = length_linr 0 xs
length_linr s []   = s
length_linr s (x:xs) = strikt_1 s (length_linr (s+1) xs)
```

Diese Funktion ist iterativ, wenn man die applikative Reihenfolge der Auswertung zugrunde legt. Wenn man die normale Reihenfolge bzw. die verzögerte Auswertung betrachtet, dann kann man in Haskell durch Steuerungsanweisungen die Auswertungsreihenfolge so beeinflussen, dass auch Zwischenvarianten möglich sind, bis hin zur applikativen Reihenfolge.

Allgemeine Funktionen auf Listen

Zwei allgemeine Funktionen (Methoden), die Listen verarbeiten sind `foldl` und `foldr`³ und z.B. „die Summe aller Elemente einer Liste“ verallgemeinern. Die Argumente sind:

- eine zweistellige Operation,
- ein Anfangselement (Einheitselement) und
- die Liste.

Hiermit kann z.B. die Summe über eine Liste von Zahlen gebildet werden:

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

Aus Effizienzgründen ist abhängig vom zweistelligen Operator mal `foldr` und mal `foldl` besser geeignet.

```
produkt xs = foldl (*) 1 xs          -- oder (foldl' (*) 1 xs)
concat xs  = foldr (++) [] xs
```

Für weitere Funktionen auf Listen siehe Prelude der Implementierung von Haskell bzw. Dokumentation in Handbüchern oder in www.haskell.org.

³Diese werden noch genauer besprochen

1.7 Lokale Funktionsdefinitionen, anonyme Funktion, Lambda-Ausdrücke in Haskell

Im folgenden stellen wir weitere Möglichkeiten von Haskell vor. Zu diesen Erweiterungen werden wir auch i.a. die Definition der Auswertung angeben. Man kann Funktionen direkt an der Stelle der Benutzung definieren: Die Syntax für einen Lambda-Ausdruck⁴ ist:

$$\backslash x_1 \dots x_n \rightarrow \langle \text{Ausdruck} \rangle$$

In der Syntax von Haskell hat obiger Lambda-Ausdruck den Status eines Ausdrucks. Damit kann man z.B. die Quadrat-Funktion auf eine weitere Weise definieren, die aber in der Wirkung völlig äquivalent zur bisherigen Definition ist.

```
quadrat = \x -> x*x
quadriere_liste xs = map (\x->x*x) xs
```

Bemerkung 1.7.1 *Der Ausdruck $\backslash x_1 \rightarrow (\backslash x_2 \rightarrow \dots (\backslash x_n \rightarrow t) \dots)$ ist bei Berechnung von Werten äquivalent zu $\backslash x_1 x_2 \dots x_n \rightarrow t$. Es gibt nur einen leichten Unterschied bei Anwendung auf weniger als n Argumente, wie wir sehen werden.*

Wir bezeichnen im folgenden die syntaktische Gleichheit von Ausdrücken mit \equiv , d.h. $s \equiv t$, wenn s und t die gleichen Ausdrücke sind (inklusive aller Namen).

Man kann lokale Bindungen (und damit auch lokale Funktionen) mit *let* definieren.

Die Syntax des *let* ist wie folgt:

$$\text{let } \{x_1 = s_1; \dots; x_n = s_n\} \text{ in } t$$

wobei x_i verschiedene Variablennamen sind und s_i und t Ausdrücke. Dies ist ein sogenanntes *rekursives let*, das auch in Haskell benutzt wird. Es ist z.B. erlaubt, dass x_1 in s_1, s_2 vorkommt, und dass gleichzeitig x_2 in s_1 und s_2 vorkommt. D.h. der Gültigkeitsbereich der definierten Variablen x_i ist der ganze *let*-Ausdruck.

Die Fakultätsfunktion und deren Anwendung lassen sich dann folgendermaßen programmieren:

```
let fakt = \x -> if x <= 1 then 1 else x*(fakt (x-1)) in fakt
```

⁴Kommt vom Lambda-Kalkül von Church, der das Zeichen λ (Lambda) als Bindungsoperator für Variablen verwendet. Haskell verwendet das Zeichen \backslash .

Die Auswertung von Ausdrücken mit `let` kann man ebenfalls als Transformationen angeben.

Bemerkung 1.7.2 *Let-Ausdrücke bieten als weitere benutzerfreundliche syntaktische Möglichkeit, Funktionen direkt in einem rekursiven `let` zu definieren, ohne syntaktisch einen Lambda-Ausdruck zu schreiben. Die Definition einer Funktions-Variablen wird direkt als Lambda-Ausdruck umgesetzt. D.h. das ist sogenannter syntaktischer Zucker, der wegtransformiert werden kann:*

$$\text{let } \{f \ x_1 \ \dots \ x_n = s; \dots\} \text{ in } t$$

ist das gleiche wie:

$$\text{let } \{f = \lambda x_1 \ \dots \ x_n . s; \dots\} \text{ in } t$$

1.7.1 Freie und Gebundene Variablen

Die Verwendung von Geltungsbereichen von Variablennamen, insbesondere geschachtelte Geltungsbereiche, kommt in (fast) allen Programmiersprachen vor, die Prozeduren mit Argumenten verwenden. Lambda- und let-Ausdrücke benutzen ebenfalls geschachtelte Geltungsbereiche. Wir nehmen das zum Anlass, genauer die Regeln für den Geltungsbereich von lokalen Namen (Variablen) zu klären. Dazu gehören die Begriffe freie und gebundene Variablen.

Wenn man erst mal durchschaut hat, wo das Problem der gebundenen und freien Variablen liegt, kann man es sich einfach machen und dafür sorgen, dass alle Namen von gebundenen Variablen (formale Parameter) verschieden sind. Danach kann man sorglos damit umgehen. Beim Umgang mit Lambdaausdrücken muss insbesondere darauf achten, bei Kopien von Unterausdrücken in den verschiedenen Kopien die Namen der gebundenen Variablen auch verschieden zu benennen.

Da der gleiche Sachverhalt auch in anderen Programmiersprachen und auch in der Prädikatenlogik auftaucht, wollen wir es hier genauer analysieren. Zudem zeigt erst die genaue Analyse, wie man richtig umbenennt, d.h. wie man dafür sorgt, dass die Variablennamen verschieden werden, ohne die operationelle Semantik des Programms zu verändern.

Der Begriff freie und gebundene Variablen soll jetzt eingeführt werden. Im Ausdruck $\lambda x. x * x$ ist die Variable x gebunden: sie wird von λ (bzw. \backslash) gebunden. Betrachtet man nur den Ausdruck $x * x$ so ist die Variable x frei in $x * x$.

Analog ist in der Formel $\forall x. x * x > 0$ die Variable x durch den Quantor \forall gebunden. Im Ausdruck $x * x > 0$ alleine ist die Variable x frei.

Wir definieren die Begriffe „freie Variablen“ bzw. „gebundene Variablen“ zunächst nur für die bereits vorgestellte Haskell-Syntax.

Definition 1.7.3 (Freie Variablen in t) Für den Bindungsbereich der Variablen in einem Lambda-Ausdruck definiert man zunächst den Begriff freie Variablen eines Ausdrucks. Als Variablen kann man nur Bezeichner (Namen) verwenden, die keine Konstruktoren sind (d.h. True, False, Zahlenkonstanten und Charakterkonstanten sind nicht zulässig).

Wir definieren $FV(t)$, die Menge der freien Variablen eines Ausdrucks t . Hier ist zu beachten, dass t hier als Programmtext bzw. Syntaxbaum gemeint ist. Das Ergebnis ist eine Menge von Namen.

- $FV(x) := \{x\}$, wenn x ein Variablenname ist.
- $FV(a) := \emptyset$, wenn a eine Konstante ist.
- $FV((s \ t)) := FV(s) \cup FV(t)$.
- $FV(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) := FV(t_1) \cup FV(t_2) \cup FV(t_3)$.
- $FV(\backslash x_1 \dots x_n \rightarrow t) := FV(t) \setminus \{x_1, \dots, x_n\}$. Dies gilt, da die Variablen x_1, \dots, x_n von $\backslash x_1 \dots x_n$ gebunden werden.
- $FV(\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } t) := (FV(t) \cup FV(s_1) \cup \dots \cup FV(s_n)) \setminus \{x_1, \dots, x_n\}$.
- $FV(\text{let } f \ x_1 \dots x_n = s \text{ in } t) := FV(\text{let } f = \backslash x_1 \dots x_n \rightarrow s \text{ in } t)$

• Der allgemeine Fall ist:

$$FV(\text{let } \left\{ \begin{array}{l} f_1 \ x_{1,1} \dots x_{1,n_1} \\ \dots \\ f_m \ x_{m,1} \dots x_{m,n_m} \end{array} = \begin{array}{l} s_1, \\ \dots \\ s_m \end{array} \right\} \text{ in } t) :=$$

$$FV(\text{let } \left\{ \begin{array}{l} f_1 = \backslash x_{1,1} \dots x_{1,n_1} \rightarrow s_1, \\ \dots \\ f_m = \backslash x_{m,1} \dots x_{m,n_m} \rightarrow s_m \end{array} \right\} \text{ in } t)$$

Freie Vorkommen von Variablen in t sind die Unterausdrücke x , die nicht unter einem Lambda bzw. **let** stehen, das x bindet.

Beispiel 1.7.4 Zur Berechnung der freien Variablen:

$$\begin{aligned} FV(\backslash x \rightarrow (f \ x \ y)) &= FV(f \ x \ y) \setminus \{x\} \\ &= \dots \\ &= \{x, f, y\} \setminus \{x\} \\ &= \{f, y\} \end{aligned}$$

Entsprechend zu freien Variablen werden die gebundenen Variablen eines Ausdrucks definiert: Es ist die Menge aller Variablen in t , die von einem λ oder einem **let** gebunden werden.

Definition 1.7.5 Gebundene Variablen $GV(t)$ von t

- $GV(x) := \emptyset$.
- $GV(a) := \emptyset$, wenn a eine Konstante ist.
- $GV((s \ t)) := GV(s) \cup GV(t)$.
- $GV(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) := GV(t_1) \cup GV(t_2) \cup GV(t_3)$
- $GV(\backslash x_1 \dots x_n \rightarrow t) := GV(t) \cup \{x_1, \dots, x_n\}$
- $GV(\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } t) := (GV(t) \cup GV(s_1) \cup \dots \cup GV(s_n) \cup \{x_1, \dots, x_n\})$
- $GV(\text{let } f \ x_1 \dots x_n = s \text{ in } t) := GV(\text{let } f = \backslash x_1 \dots x_n \rightarrow s \text{ in } t)$
- Der allgemeine Fall ist:

$$GV(\text{let } \left\{ \begin{array}{l} f_1 \ x_{1,1} \dots x_{1,n_1} = s_1, \\ \dots \dots \dots \\ f_m \ x_{m,1} \dots x_{m,n_m} = s_m \end{array} \right\} \text{ in } t) :=$$

$$GV(\text{let } \left\{ \begin{array}{l} f_1 = \backslash x_{1,1} \dots x_{1,n_1} \rightarrow s_1, \\ \dots \dots \dots \\ f_m = \backslash x_{m,1} \dots x_{m,n_m} \rightarrow s_m \end{array} \right\} \text{ in } t)$$

Beispiel 1.7.6 Zur Berechnung von gebundenen Variablen:

$$\begin{aligned} GV(\backslash x \rightarrow (f \ x \ y)) &= GV(f \ x \ y) \cup \{x\} \\ &= \dots \\ &= \emptyset \cup \{x\} \\ &= \{x\} \end{aligned}$$

Die Variablen x_i im Rumpf t des Lambda-Ausdrucks $\backslash x_1 \dots x_n \rightarrow t$ sind gebundene Variablen im Ausdruck $\backslash x_1 \dots x_n \rightarrow t$.

Die Definition der freien Variablen definiert gleichzeitig den Begriff des Gültigkeitsbereichs einer Variablen. Haskell verwendet **lexikalische Gültigkeitsbereiche (lexical scoping)** von Variablen.

- In $\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } t$ werden genau die Vorkommen der freien Variablen $x_i, i = 1, \dots, n$, die in den Ausdrücken $s_i, i = 1, \dots, n$ und t vorkommen, gebunden. Das ist genau der Gültigkeitsbereich der Variablen $x_i, i = 1, \dots, n$.
- In $\backslash x_1 \dots x_n \rightarrow t$ werden genau die freien Variablen x_1, \dots, x_n in t gebunden. Das ist der Gültigkeitsbereich der Variablen x_1, \dots, x_n .

Beispiel 1.7.7 Im Ausdruck $\backslash x \rightarrow (x \ (\backslash x \rightarrow x * x))$ ist x gebunden, aber in zwei Gültigkeitsbereichen (Bindungsbereichen.)

Im Unterausdruck $s \equiv (x \ (\backslash x \rightarrow x * x))$ kommt x frei und gebunden vor. Benennt man die gebundene Variable x in y um, so ergibt sich: $(x \ (\backslash y \rightarrow y * y))$.

Beispiel 1.7.8 Der folgende Ausdruck hat zwei Gültigkeitsbereichen (Bindungsbereiche) für verschieden erklärte, aber gleich benannte Bezeichner x .

```
let x = 10 in (let x = 100 in (x+x)) + x
```

Macht man die Bindungsbereiche explizit und benennt um, dann ergibt sich:

```
let x1 = 10 in (let x2 = 100 in (x2+x2)) + x1
```

Dieser Term wertet zu 210 aus.

Die Eingabe eines Ausdrucks wie

```
let x = (x*x) in (x+x)
```

führt zu Nichtterminierung. Im Haskell ergibt sich ein Laufzeitfehler.

Beispiel 1.7.9

```
let    y = 20*z
      x = 10+y
      z = 15
      in x
```

Oder in anderer Syntax:

```
let    {y = 20*z; x = 10+y ; z = 15}    in x
```

Dies wertet aufgrund der obigen Bindungsregel aus zu : 310.

```
Beispiel 1.7.10 let {x = 1;y = 2}
                  in (let {y =3;z = 4}
                      in (let z = 5
                          in (x+y+z)))
```

$x = 1; y = 2$

$y = 3; z = 4$

$z = 5$

$x + y + z$

Die zugehörige Bindung zu einer Variablen ist immer die innerste Möglichkeit. Z.B. z ist in der innersten Umgebung an 5 gebunden. Auswertung ergibt: $1 + 3 + 5 = 9$

Beispiel 1.7.11 Ein Programm mit den Definitionen $f_i := e_i \mid i = 1, \dots, n$ und dem auszuwertenden Ausdruck `main` kann als „großes“ `let` betrachtet werden:

```
let {f1 := e1; ...; fn := en} in main
```

Beispiel 1.7.12 Dieses Beispiel soll zeigen, dass man mit einem `let` leicht redundante Auswertungen vermeiden kann.

Die Funktion $f(x, y) := x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$ lässt sich effizienter berechnen durch Vermeidung von Doppelberechnungen, wenn man definiert:

$$\begin{aligned} a &:= 1 + xy \\ b &:= 1 - y \\ f(x, y) &:= xa^2 + yb + ab \end{aligned}$$

Der zugehörige Ausdruck ist:

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

1.7.2 Bemerkungen zur Auswertung mit `let` und `Lambda`

Eine vereinfachte Vorstellung (Sharing-Variante zur Optimierung) von `let`-Ausdrücken und deren Wirkungsweise ist folgende: Im Ausdruck `let x = s, ... in t` ist der Ausdruck s nur einmal im Speicher, und jedes Vorkommen von x wird so behandelt, als würde dort ein Verweis auf das gemeinsam verwendete s stehen.

D.h. diese Sichtweise würde als äquivalent ansehen:

```
let x = quadrat 1 in x*x
```

und

$(\text{quadrat1})^{(1)} * (\text{quadrat1})^{(1)}$.

Diese Vorstellung ist in vielen Fällen richtig, aber nicht in allen:

1. Das `let` ist rekursiv, und somit handelt es sich evtl. um zyklische Verweise, so dass ein iteriertes Kopieren mit Gleichheitsmarkierungen nicht jedes `let` eliminieren kann.
2. Die Sharing-Variante löst nicht den Konflikt zwischen der Notwendigkeit, beim Kopieren von Abstraktionen den Rumpf wirklich zu verdoppeln, aber die bisherigen Gleichheitsmarkierung zu beachten. Z.B.

```
let x = quadrat 2, z = \y. y+x in (z 3)+ (z 5)
```

3. Die Gleichheitsmarkierungen zeigen nur in einfachen Fällen, was gemeint ist. In komplizierteren Fällen, z.B. bei Bindungen $x = y$, oder wenn nach Auswertungsschritten ein Ausdruck mehrere Markierungen hat, ist die `let`-Methode exakter, und kann die Zweifelsfälle klären. Insbesondere, wenn die Frage der genauen Gültigkeitsbereich nach Transformationen und Reduktionen zu klären ist.

Eine Möglichkeit, die operationale Semantik eindeutig und sauber zu definieren, ist die Angabe auf Transformationsregeln auf den Ausdrücken mit `let`, zusammen mit einer Auswertungsstrategie.

1.8 Funktionen als allgemeine Methoden

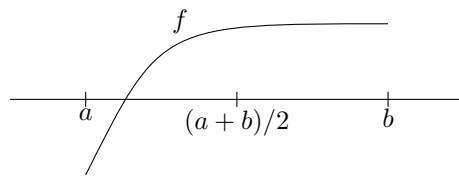
In diesem Abschnitt zeigen wir, dass es relativ einfach ist, Funktionen zu definieren, die als allgemeine Methoden verwendet werden können, wobei diese wieder Funktionen als Argumente haben können.

Als Beispiele betrachten wir zwei Algorithmen, die auf (arithmetischen) Funktionen definiert sind, wie z.B. Nullstellenbestimmung, Integrieren, Differenzieren, ..., und die wir hier als Anwendungsbeispiele besprechen wollen.

Formuliert man diese Algorithmen, so ist es bequem, wenn man die Funktionen direkt als Argumente an eine definierte Haskell-Funktion zur Nullstellenbestimmung übergeben kann, die die allgemeinere Methode implementiert.

Beispiel 1.8.1 *Bestimmung von Nullstellen einer stetigen Funktion mit Intervallhalbierung*

Idee: Sei f stetig und $f(a) < 0 < f(b)$,



- wenn $f((a+b)/2) > 0$, dann suche die Nullstelle im Intervall $[a, (a+b)/2]$.
- wenn $f((a+b)/2) < 0$, dann suche die Nullstelle im Intervall $[(a+b)/2, b]$.

Eingabe der Methode `intervall_halbierung` sind

- Name der Haskell-Funktion, die die arithmetischen Funktion implementiert.
- Intervall-Anfang

- *Intervall-Ende*
- *Genauigkeit der Nullstelle (absolut).*

Wir geben den Typ der Funktion an. Haskell gibt folgendes aus:

```
*Main> :t suche_nullstelle
suche_nullstelle
  :: (Double -> Double) -> Double -> Double -> Double -> Double
```

```
suche_nullstelle f a b genau =
  let fa = f a
      fb = f b
  in
    if fa < 0 && fb > 0
    then suche_nullstelle_steigend f a b genau
    else if fa > 0 && fb < 0
    then suche_nullstelle_fallend f a b genau
    else error ("Werte haben gleiches Vorzeichen" ++
               (show a) ++ (show b))
suche_nullstelle_steigend f a b genau =
  suche_nullstelle_r f a b genau
suche_nullstelle_fallend f a b genau =
  suche_nullstelle_r f b a genau
suche_nullstelle_r f a b genau =
  let m = (mittelwert a b)
  in if abs (a - b) < genau
    then m
    else let fm = f m
         in if fm > 0
           then suche_nullstelle_r f a m genau
           else if fm < 0
                then suche_nullstelle_r f m b genau
           else m
```

```
*Main> suche_nullstelle sin 2 4 0.00001
3.14159012
```

Den Aufwand der Intervallhalbierungsmethode kann man bestimmen durch die maximale Anzahl der Schritte: $\lceil \log_2(L/G) \rceil$, wobei L die Länge des Intervalls und G die Genauigkeit ist. D.h der Zeitbedarf ist $O(\log(L/G))$, während der Platzbedarf Größenordnung $O(1)$ hat. Hierbei muss man annehmen, dass die Implementierung der arithmetischen Funktion f $O(1)$ Zeit und Platz braucht.

Beispiel 1.8.2 Verwendung der Nullstellensuche

n -te Wurzel aus einer Zahl a : $\sqrt[n]{a}$

```
nnte_wurzel n a = suche_nullstelle (\x-> x^n -a) 1 (a^n) 0.00000001
```

```
*Main> nte_wurzel 10 1024 ↵
2.00000000372529
*Main> nte_wurzel 10 10 ↵
1.2589254114675787
*Main> (nte_wurzel 10 10)^10 ↵
9.999999974058156
```

1.8.1 Funktionen als Ergebnis

Es kann sehr nützlich sein, auch Funktionen als Ausgabe einer Funktion zu erlauben, z.B., wenn man die Nullstellenbestimmung auf zusammengesetzte Funktionen anwenden will. Das einfachste Beispiel ist die Komposition von Funktionen:

```
komp::(a -> b) -> (c -> a) -> c -> b
komp f g x = f (g x)
```

```
*Main> suche_nullstelle (sin 'komp' quadrat) 1 4 0.00000001 ↵
1.772453852929175
```

`(sin 'komp' quadrat)` entspricht $\sin(x^2)$ und `quadrat 'komp' sin` entspricht $(\sin(x))^2$. Der Typ von `komp` $(a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$ bedarf der Erklärung:

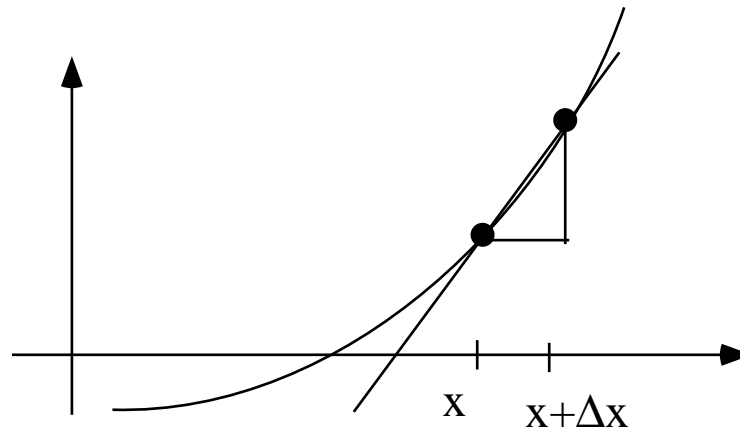
Es gibt drei Argumenttypen: $(a \rightarrow b)$, $(c \rightarrow a)$ und c , wobei a, b, c Typvariablen sind.

D.h. Argument eins und zwei (f_1 , und f_2) sind vom Funktionstyp, Argument 3 ist irgendein Typ. Diese Typen müssen so sein, dass f_2 angewendet auf das dritte Argument, den Typ a ergibt, und f_1 angewendet auf dieses Ergebnis den Typ b .

Wendet man `komp` nur auf zwei Argumente an, dann hat das Resultat den Typ: $c \rightarrow b$.

In Haskell ist Komposition schon vordefiniert `sin 'komp' quadrat` wird einfach als `sin . quadrat` geschrieben.

Beispiel 1.8.3 Ein weiteres Beispiel für allgemeine Methoden, die auch Funktionsargumente haben, ist näherungsweise Differenzieren



$$Df(x) := \frac{f(x + dx) - f(x)}{dx}$$

```
ableitung :: (Double -> Double) -> Double -> (Double -> Double)
ableitung f deltax = \x -> ((f(x+deltax)) - (f x)) / deltax
```

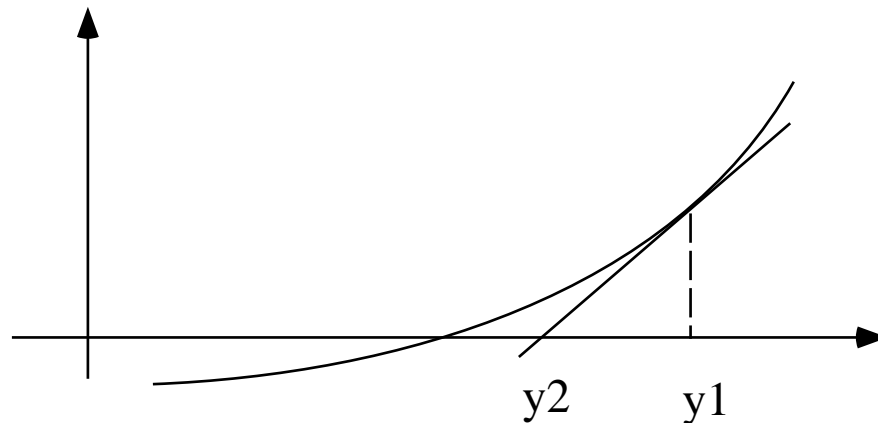
Dies liefert als Resultat eine neue Funktion, die die Ableitung annähert.

```
*Main> ((ableitung (\x -> x^3) 0.00001) 5)
75.00014999664018
```

(korrekter Wert: 75)

Hiermit kann man die Newtonsche Methode zur Bestimmung der Nullstellen beliebiger (gutartiger) Funktionen implementieren.

Vorgehen: Ersetze den Schätzwert y jeweils durch den verbesserten Schätzwert und nehme als Ableitung der Funktion eine approximative Ableitungsfunktion.



$$y - \frac{f(y)}{Df(y)}.$$

```

newton_nst :: (Double -> Double) -> Double -> Double
newton_nst f y =
    if (abs (f y)) < 0.0000000001
    then y
    else newton_nst f (newton_nst_verbessern y f)

newton_nst_verbessern y f = y - (f y) / (ableitung f 0.00001 y)

```

Damit kann man leicht näherungsweise Nullstellen bestimmen:

```

*Main> newton_nst (\x -> x^2-x) 4 ↵
1.00000000000001112
*Main> newton_nst cos 2 ↵
1.5707963267948961

```

Dies berechnet die Nullstelle der Funktion $x^2 - x$ mit Startwert 4 und als zweites eine Nullstelle der Funktion \cos , die $\pi/2$ approximiert.

Kapitel 2

Datenstrukturen, Listen und List-Komprehensionen in Haskell

2.1 Datenstrukturen und Typen in Haskell

Bisher haben wir nur die eingebauten Basisdatentypen wie Zahlen und Wahrheitswerte benutzt.

Basisdatentypen

Ganze Zahlen (`Int`) umfassen in Haskell nur einen bestimmten Zahlbereich.

Wie in den meisten Programmiersprachen wird der Zahlbereich durch die Binärzahl b begrenzt, die in ein Halbwort (2 Byte), Wort (4 Byte), Doppelwort (8 Byte) passen, so dass der Bereich $-b, b$ darstellbar ist: $-2^{31}, 2^{31}-1$. Die Darstellung ist somit für die Operatoren nicht ausreichend, da die Ergebnisse außerhalb dieses Bereichs sein könnten. Das bedeutet, man muss damit rechnen, dass bei Operationen ($+$, $-$, $*$, $/$) das Ergebnis nicht in den Zahlbereich passt, d.h. dass ein Überlauf stattfindet.

Unbeschränkte ganze Zahlen (`Integer`) Diese kann man in Haskell verwenden, es gibt diese aber nicht in allen Programmiersprachen. Hier ist die Division problematisch, da die Ergebnisse natürlich nicht immer ganze Zahlen sind. Verwendet wird die ganzzahlige Division. Die Division durch 0 ergibt einen Laufzeitfehler.

Rationale Zahlen Werden in einigen Programmiersprachen unterstützt; meist als Paar von `Int`, in Haskell auch exakt als Paar von `Integer`.

Komplexe Zahlen sind in Haskell in einem Module `Complex` verfügbar, der einige Funktionen implementiert, u.a. auch `exp` und die Winkelfunktio-

nen für komplexe Eingaben. Komplexe Zahlen sind z.B. in Python direkt verfügbar.

Gleitkommazahlen (Gleitpunktzahlen) (Float, Double). Das sind Approximationen für reelle Zahlen in der Form `[Mantisse, Exponent]`, z.B. $1.234e - 40$. Die Bedeutung ist $1.234 * 10^{-40}$. Die interne Darstellung ist getrennt in Mantisse und Exponent. Die arithmetischen Operationen sind alle definiert, aber man muß immer damit rechnen, dass Fehler durch die Approximation auftreten (Rundungsfehler, Fehler durch Abschneiden), dass ein Überlauf bzw. Unterlauf eintritt, wenn der Bereich des Exponenten überschritten wurde, oder dass Division durch 0 einen Laufzeitfehler ergibt. Beachte, dass es genaue IEEE-Standards gibt, wie Rundung, Abschneiden, die Operationen usw. für normale Genauigkeit (4 Byte) oder für doppelte Genauigkeit (8 Byte) funktionieren, damit die Ergebnisse von Berechnungen (insbesondere finanzielle) auf allen Rechnern den gleichen Wert ergeben.

Meist sind diese Operationen schon auf der Prozessorebene implementiert, so dass man diese Operationen i.a. über die Programmiersprache aufruft.

Zeichen, Character. Sind meist ASCII-Zeichen (1 Byte). Ein in Haskell verfügbarer Standard, der 4 Bytes pro Zeichen verwendet und der es erlaubt, viel mehr Zeichen zu kodieren, und der für (fast) alle Sprachen Kodierungen bereithält, ist der Unicode-Standard (www.unicode.org). Es gibt drei Varianten, die es erlauben, Kompressionen zu verwenden, so dass häufige Zeichen in einem oder 2 Byte kodiert werden. In Unicode gibt es u.a. zusammengesetzte Zeichen, z.B. wird ü bevorzugt mittels zwei Bytes kodiert.

Funktionen dazu in Haskell (im Modul `Char`): `ord` liefert die Hex-Codierung eines Zeichens und `chr` ist die Umkehrfunktion von `ord`.

Wir werden nun größere, zusammengesetzte Datenobjekte als Abstraktion für Daten verwenden.

2.1.1 Einfache Typen

Beispiel 2.1.1 Rationale Zahlen *Eine rationale Zahl kann man als zusammengesetztes Objekt verstehen, das aus zwei Zahlen, dem Zähler und dem Nenner besteht. Normalerweise schreibt man $\frac{x}{y}$ für die rationale Zahl mit Zähler x und dem Nenner y .*

Die einfachste Methode, dies in Haskell darzustellen, ist als Paar von Zahlen (x, y) . Beachte, dass in Haskell rationale Zahlen bereits vordefiniert sind, und die entsprechenden Paare (x, y) als $x\%y$ dargestellt werden. Z.B.:

```
*Main> (3%4)*(4%5)
3 % 5
*Main> 1%2+2%3
7 % 6
```

Datenkonversionen macht man mit `toRational` bzw. `truncate`. Es gibt rationale Zahlen mit kurzen und beliebig langen ganzen Zahlen.

Paare von Objekten kann man verallgemeinern zu n -Tupel von Objekten:

(t_1, \dots, t_n) stellt ein n -Tupel der Objekte t_1, \dots, t_n dar.

Die Typen von Tupeln werde ebenfalls als Tupel geschrieben.

Beispiel 2.1.2 Wir geben Tupel und Typen dazu an. Da der Typ von Zahlen etwas allgemeiner ist in Haskell, ist auch der Typ der Tupel mit Zahleinträgen in Haskell allgemeiner.

<code>(1,2,3,True)</code>	<code>Typ (Int, Int, Int, Bool)</code>
<code>(1,(2,True),3)</code>	<code>Typ (Int, (Int, Bool), Int)</code>
<code>("hallo",False)</code>	<code>Typ (String, Bool)</code>
<code>(fakt 100,\x-> x)</code>	<code>Typ (Integer, a -> a)</code>

Um mit komplexeren Datenobjekten in einer Programmiersprache umgehen zu können, benötigt man:

Datenkonstruktoren: Hiermit wird ein Datenobjekt neu konstruiert, wobei die Teile als Parameter übergeben werden.

Datenselektoren: Funktionen, die gegeben ein Datenobjekt, bestimmte Teile daraus extrahieren.

Zum Beispiel konstruiert man ein Paar (s, t) aus den beiden Ausdrücken s und t . Da man die Ausdrücke wieder aus dem Paar extrahieren können muß, benötigt man die Selektoren `fst` und `snd`, für die gelten muß: `fst(s, t) = s` und `snd(s, t) = t`.

In der Syntax betrachten wir nur Ausdrücke $(c\ t_1 \dots t_n)$, bei denen $ar(c) = n$ gilt.

Für ein n -Tupel benötigt man n Datenselektoren, auch wegen der Typisierbarkeit, für jede Stelle des Tupels einen. Die Definition dieser Selektoren wird in Haskell syntaktisch vereinfacht durch sogenannte *Muster (Pattern)*. Einfache Beispiele einer solchen Definition sind:

```
fst (x,y) = x
snd (x,y) = y
selektiere_3_von_5 (x1,x2,x3,x4,x5) = x3
```

Diese Muster sind syntaktisch überall dort erlaubt, wo formale Parameter (Variablen) neu eingeführt werden, d.h. in Funktionsdefinitionen, in Lambda-Ausdrücken und in `let`-Ausdrücken.

Das Verwenden von Mustern hat den Vorteil, dass man keine extra Namen für die Selektoren braucht. Will man die Implementierung verstecken, dann sind allerdings explizite Selektoren notwendig.

Nun können wir auch den Typ von Tupeln hinschreiben: n -Tupel haben einen impliziten Konstruktor: $(\underbrace{}_n)$. Der Typ wird entsprechend notiert:

```
(1,1) :: (Integer, Integer)
selektiere_3_von_5 :: (t1,t2,t3,t4,t5) -> t3
(1,2.0, selektiere_3_von_5) :: (Integer, Float,t1,t2,t3,t4,t5) -> t3
```

Bemerkung 2.1.3 *In Haskell kann man Typen und Konstruktoren mittels der `data`-Anweisung definieren. Zum Beispiel*

```
data Punkt          = Punktkonstruktor Double Double
                    deriving(Eq,Show)
data Strecke        = Streckenkonstruktor Punkt Punkt
                    deriving(Eq,Show)
data Viertupel a b c d = Viertupelkons a b c d
                    deriving(Eq,Show)
```

Definition 2.1.4 *Ein Muster ist ein Ausdruck, der nach folgender Syntax erzeugt ist:*

$$\begin{aligned} \langle \text{Muster} \rangle ::= & \langle \text{Variable} \rangle \mid (\langle \text{Muster} \rangle) \\ & \mid \langle \text{Konstruktor}_{(n)} \rangle \underbrace{\langle \text{Muster} \rangle \dots \langle \text{Muster} \rangle}_n \\ & \mid (\langle \text{Muster} \rangle, \dots, \langle \text{Muster} \rangle) \end{aligned}$$

Als Kontextbedingung hat man, dass in einem Muster keine Variable doppelt vorkommen darf. Beachte, dass Zahlen und Character als Konstruktoren zählen.

Die erlaubten Transformationen bei Verwendung von Mustern kann man in etwa so beschreiben: wenn das Datenobjekt wie das Muster aussieht und die Konstruktoren übereinstimmen, dann werden die Variablen an die entsprechenden Ausdrücke (Werte) gebunden. Diese *Musteranpassung* kann man mit der Funktion

`anpassen Muster Ausdruck`¹

rekursiv beschreiben, wobei das Ergebnis eine Menge von Bindungen, d.h. von Paaren $x \rightarrow t$, wobei x eine Variable und t ein Ausdruck ist.

¹Das ist keine Haskell-Funktion, sondern eine rekursive Beschreibung. Diese könnte im Interpreter angesiedelt sein.

- `anpassen Kon Kon = \emptyset` (passt; aber keine Bindung notwendig.)
- `anpassen $x\ t = \{x \rightarrow t\}$:` (x wird an t gebunden.)
- `anpassen (Kon $s_1 \dots s_n$) (Kon $t_1 \dots t_n$) =
anpassen $s_1\ t_1 \cup \dots \cup$ anpassen $s_n\ t_n$`
- `anpassen (Kon $s_1 \dots s_n$) (Kon' $t_1 \dots t_n$) = Fail`, wenn $\text{Kon} \neq \text{Kon}'$.

Dies bedeutet auch, dass die Musteranpassung erzwingt, dass die Datenobjekte, die an das Muster angepasst werden sollen, zunächst ausgewertet werden müssen. Muster wirken wie ein `let`-Ausdruck mit Selektoren kombiniert.

Man kann Zwischenstrukturen in Mustern ebenfalls mit Variablen benennen:

Das Muster $(x, y@(z_1, z_2))$ wird bei der Musteranpassung auf $(1, (2, 3))$ folgende Bindungen liefern: $x = 1, y = (2, 3), z_1 = 2, z_2 = 3$.

Datentypen

Mittels der Anweisung `data` kann man eigene Datentypen definieren. Ein Datentyp korrespondiert zu einer Klasse von Datenobjekten. Hierbei wird für den Datentyp ein Name eingeführt, und die neuen Namen der Konstruktoren zu diesem Datentyp werden angegeben und genauer spezifiziert: Stelligkeit und Typ der Argumente. Hierbei darf der Typ auch rekursiv verwendet werden.

Eine sehr flexible Erweiterung ist die Parametrisierung der Datentypen. Z.B. kann man Listen von Paaren von Integer definieren; der Typ ist dann `[(Integer, Integer)]`. Das Paar `(Integer, Integer)` ist dann der Typ-Parameter.

Die Datenkonstruktoren haben den Status eines Ausdrucks in Haskell. Es gibt eine eigene Anweisung, die Datentypen definiert, wobei man auf die definierten Typnamen zurückgreift und auch rekursive Definitionen machen darf. Es genügt, nur die Datenkonstruktoren zu definieren, da die Selektoren durch Musteranpassung (Musterinstanziierung) definierbar sind. In der Typanweisung werden auch evtl. neue Typnamen definiert. Diese Typnamen können mit einem anderen Typ parametrisiert sein (z.B. `[a]`: Liste mit dem dem a).

Jedes Datenobjekt in Haskell muß einen Typ haben. Die eingebauten arithmetischen Datenobjekte haben z.B. die Typen `Int`, `Integer`, `Float`, `Char`, Datenkonstruktoren sind in der entsprechenden Typdefinition mit einem Typ versehen worden. Das Typechecking kann sehr gut mit parametrisierten Typen umgehen, erzwingt aber gewisse Regelmäßigkeiten, wie z.B. die Kohärenz von Listen.

Beispiel 2.1.5 *Punkte und Strecken und ein Polygonzug werden in der Zahlenebene dargestellt durch Koordinaten:*


```

data Punkt(a)      = Punkt a a
data Strecke(a)    = Strecke (Punkt a) (Punkt a)
data Vektor(a)     = Vektor a a
data Polygon a     = Polygon [Punkt a]

```

Zur Erläuterung:

data	Punkt(a)	=	Punkt a a
<i>Schlüsselwort</i>	<i>Neuer Datentypname</i>		<i>Neuer Datenkonstruktorname, zweistellig</i>

Die Argumente des Datenkonstruktors haben gleichen Typ *a*, der mit dem Parameter des Typs übereinstimmen muss. **Strecke** ist ein neuer Datentyp, der aus zwei Punkten besteht. Es ist unproblematisch, Datentyp und Konstruktor gleich zu benennen, da keine Verwechslungsgefahr besteht. Der Parameter *a* kann beliebig belegt werden: z.B. mit **Float**, **Int**, aber auch mit **[(Int,Char)]**.

Haskell sorgt mit der Typüberprüfung dafür, dass z.B. Funktionen, die für Punkte definiert sind, nicht auf rationale Zahlen angewendet werden, die ebenfalls aus zwei Zahlen bestehen.

Einige Funktionen, die man jetzt definieren kann, sind:

```

addiereVektoren::Num a => Vektor a -> Vektor a -> Vektor a
addiereVektoren (Vektor a1 a2) (Vektor b1 b2) =
    Vektor (a1 + b1) (a2 + b2)

streckenLaenge (Strecke (Punkt a1 a2) (Punkt b1 b2)) =
    sqrt (fromInteger ((quadrat (a1 - b1))
                          + (quadrat (a2-b2))))

verschiebeStrecke s v =
    let  (Strecke (Punkt a1 a2) (Punkt b1 b2)) = s
        (Vektor v1 v2) = v
    in  (Strecke (Punkt (a1+v1) (a2+v2))
        (Punkt (b1+v1) (b2+v2)))

teststrecke = (Strecke (Punkt 0 0) (Punkt 3 4))

test_streckenlaenge = streckenLaenge
    (verschiebeStrecke teststrecke
    (Vektor 10 (-10)))

```

```
*Main> streckenLaenge teststrecke ↵
5.0
*Main> test_streckenlaenge ↵
5.0
```

Wenn wir die Typen der Funktionen überprüfen, erhalten wir:

```
*Main> :t addiereVektoren ↵
addiereVektoren :: Num a => Vektor a -> Vektor a -> Vektor a
*Main> streckenlaenge ↵
streckenlaenge :: Num a => Strecke a -> Float
*Main> test_streckenlaenge ↵
test_streckenlaenge :: Float
*Main> verschiebeStrecke ↵
verschiebeStrecke :: Num a => Strecke a -> Vektor a -> Strecke a
```

2.1.2 Summentypen und Fallunterscheidung

Um eine Entsprechung des Booleschen Datentyps selbst zu definieren braucht man Summentypen, die mehrere Konstruktoren haben können: Klassen von Datenobjekten verschiedener Struktur kann man dann in einem Typ vereinigen.

Zum Beispiel sieht die Definition des Booleschen Datentyp so aus:

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

Die Konstruktoren kann man in Patternmatch / Mustern verwenden

Man kann auch ohne Pattern eine Fallunterscheidung nach Konstruktoren programmieren mit dem **case**-Primitiv :

Definition 2.1.6 Die Syntax des **case**-Ausdruck, der zur Fallunterscheidung verwendet werden kann:

$$\text{case } \langle \text{Ausdruck} \rangle \text{ of } \{ \langle \text{Muster} \rangle \rightarrow \langle \text{Ausdruck} \rangle; \dots; \langle \text{Muster} \rangle \rightarrow \langle \text{Ausdruck} \rangle \}$$

Die Kontextbedingung ist, dass die Muster vom Typ her passen. Die Bindungsbereiche der Variablen in den Mustern sind genau die zugehörigen Ausdrücke hinter dem Pfeil (\rightarrow).

Der Gültigkeitsbereich der Variablen in bezug auf das **case**-Konstrukt kann an der Definition der freien Variablen abgelesen werden:

$$\begin{aligned}
& FV(\text{case } s \text{ of } (c_1 \ x_{11} \ \dots \ x_{1n_1} \rightarrow t_1); \dots; \\
& \quad (c_k \ x_{k1} \ \dots \ x_{kn_k} \rightarrow t_k)) \\
& \quad := \\
& \quad FV(s) \cup FV(t_1) \setminus \{x_{11}, \dots, x_{1n_1}\} \dots \\
& \quad \cup FV(t_k) \setminus \{x_{k1}, \dots, x_{kn_k}\} \\
& \quad GV(\text{case } s \text{ of } (c_1 \ x_{11} \ \dots \ x_{1n_1} \rightarrow t_1); \dots; \\
& \quad \quad (c_k \ x_{k1} \ \dots \ x_{kn_k} \rightarrow t_k)) \\
& \quad := \\
& \quad GV(s) \cup GV(t_1) \cup \{x_{11}, \dots, x_{1n_1}\} \cup \dots \\
& \quad \cup FV(t_k) \cup \{x_{k1}, \dots, x_{kn_k}\}
\end{aligned}$$

Beispiel 2.1.7 Folgende Definition ist äquivalent zum in Haskell definierten logischen “und” (und auch zu `und2` und `und3`) : `&&`.

```
und4  x y = case x of True -> y; False -> False
```

Folgende Definition ist äquivalent zum normalen `if . then . else`. D.h. `case`-Ausdrücke sind eine Verallgemeinerung des `if-then-else`.

```
mein_if  x y z = case x of True -> y; False -> z
```

Definition 2.1.8 (Reduktionen zur `case`-Behandlung)

$$\boxed{\text{Case-Reduktion}} \quad \frac{(\text{case } (c \ t_1 \ \dots \ t_n) \text{ of } \dots (c \ x_1 \ \dots \ x_n \rightarrow s) \dots)}{s[t_1/x_1, \dots, t_n/x_n]}$$

2.2 Rekursive Datenobjekte: z.B. Listen

Listen sind eine Datenstruktur für Folgen von gleichartigen (gleichgetypten) Objekten. Da wir beliebig lange Folgen verarbeiten und definieren wollen, nutzen wir die Möglichkeit, rekursive Datentypen zu definieren. Der Typ der Objekte in der Liste ist nicht festgelegt, sondern wird hier als (Typ-) Variable in der Definition verwendet. D.h. aber, dass trotzdem in einer bestimmten Liste nur Elemente eines Typs sein dürfen.

```
-- eine eigene Definition
data Liste a = Leereliste | ListenKons a (Liste a)
```

Dies ergibt Datenobjekte, die entweder leer sind: `Leereliste`, oder deren Folgelemente alle den gleichen Typ a haben, und die aufgebaut sind als `ListenKons b_1 (ListenKons $b_2 \dots$ Leereliste)`.

Listen sind in Haskell eingebaut und werden syntaktisch bevorzugt behandelt. Aber: man könnte sie völlig funktionsgleich auch selbst definieren. Im folgenden werden wir die Haskell-Notation verwenden. Die Definition würde man so hinschreiben: (allerdings entspricht diese nicht der Syntax)

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

Wir wiederholen einige rekursive Funktionen auf Listen:

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

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

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

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

Allgemeine Funktionen auf Listen

Zwei allgemeine Funktionen (Methoden), die Listen verarbeiten sind `foldl` und `foldr` und z.B. „die Summe aller Elemente einer Liste“ verallgemeinern.

Die Argumente sind:

- eine zweistellige Operation,
- ein Anfangselement (Einheitselement) und
- die Liste.

```

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

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

```

Für einen Operator \otimes wie $*$, $+$, **append** und ein Anfangselement (Einheits-element) e , wie zB. $1, 0, []$ ist der Ausdruck

$$\text{foldl } \otimes e [a_1, \dots, a_n]$$

äquivalent zu

$$((\dots ((e \otimes a_1) \otimes a_2) \dots) \otimes a_n).$$

Analog entspricht

$$\text{foldr } \otimes e [a_1, \dots, a_n]$$

der umgekehrten Klammerung:

$$a_1 \otimes (a_2 \otimes (\dots (a_n \otimes e))).$$

Für einen assoziativen Operator \otimes und wenn e Rechts- und Linkseins zu \otimes ist, ergibt sich derselbe Wert.

Die Operatoren **foldl** und **foldr** unterscheiden sich bzgl. des Ressourcenbedarfs in Abhängigkeit vom Operator und vom Typ des Arguments.

Beispiele für die Verwendung, wobei die jeweils bessere Variante definiert wurde.

```

sum :: (Num a) => [a] -> a
sum xs      = foldl (+) 0 xs

produkt :: (Num a) => [a] -> a
produkt xs = foldl (*) 1 xs      -- bzw. (foldl' (*) 1 xs)

concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs

```

Eine Funktionen zum Umdrehen einer Liste:

```

reverse :: [a] -> [a]
reverse xs = foldl (\x y -> y:x) [] xs

```

Eine Liste von Listen wird um eine Ebene reduziert, d.h. flachgemacht zu einer einzigen Liste:

```
concat xs = foldr append [] xs
```

Folgende Funktion liefert eine Liste von (Pseudo-) Zufallszahlen.²

```
randomInts a b
```

Weitere Listen-Funktionen sind:

```
-- Restliste nach n-tem Element
drop :: Int -> [a] -> [a]
drop 0 xs      = xs
drop _ []      = []
drop n (_:xs) | n>0 = drop (n-1) xs
drop _ _       = error "Prelude.drop: negative argument"
```

Bildet Liste der Paare

```
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _ _          = []
```

Bildet aus Liste von Paaren ein Paar von Listen

```
unzip :: [(a,b)] -> ([a],[b])
unzip = foldr (\(a,b) (as,bs) -> (a:as, b:bs)) ([], [])
```

Beispielauswertungen sind:

```
*Main> drop 10 [1..100]
[11,12,...]
*Main> zip "abcde" [1..]
[( 'a',1),('b',2),('c',3),('d',4),('e',5)]
*Main> zip ['a'..'z'] [1..]
[( 'a',1),('b',2),('c',3),('d',4),('e',5),('f',6),('g',7),('h',8),
('i',9),('j',10),('k',11),('l',12),('m',13),('n',14),('o',15),
('p',16),('q',17),('r',18),('s',19),('t',20),('u',21),('v',22),
('w',23),('x',24),('y',25),('z',26)]
*Main> unzip (zip "abcdefg" [1..])
("abcdefg",[1,2,3,4,5,6,7])
```

Für weitere Funktionen auf Listen siehe Prelude der Implementierung von Haskell bzw. Dokumentation in Handbüchern oder in www.haskell.org.

²wenn `import System.Random` benutzt wird und `prg2.hs`.

2.2.1 Ressourcenbedarf von Listenfunktionen

Beim Abschätzen des Ressourcenbedarfs von Listenfunktionen muss man sich darauf festlegen, welche Auswertung in Haskell man meint. Es gibt drei Möglichkeiten:

Komplette Auswertung: Auswertung der gesamten Ergebnis-Liste und auch der Elemente. Das entspricht der Auswertung, die man im Interpreter macht, da hier die Anforderung ist, die gesamte Liste zu drucken.

Rückgrat-Auswertung: Auswertung nur der Listen-Konstruktoren, des *Rückgrats der Liste*, aber nicht der Elemente, d.h. gerade so viel, wie die Funktion `length` von der Liste benötigt.

Kopf-Auswertung : Auswertung bis der oberste Listenkonstruktor sichtbar ist. D.h. werte nur soviel aus, dass die Frage „Ist die Liste leer oder nicht leer“ beantwortet werden kann.

Beispiel 2.2.1 *Länge einer Liste:*

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

length_lin xs      = length_linr 0 xs
length_linr s []    = s
length_linr s (x:xs) = (length_linr $(s+1)) xs
```

Der Ausdruck `length` ist für eine bereits ausgewertete Liste der Länge n benötigt $O(n)$ Reduktionsschritte. Der benötigte Zwischenspeicher ist für die nicht-iterative Version ebenfalls $O(n)$, da die Berechnung des Endergebnisses erst erfolgt, wenn die Liste zu Ende abgearbeitet wurde. Für die iterative Version der Funktion `length` ist der Bedarf an Zwischenspeicher konstant, d.h. $O(1)$.

Beispiel 2.2.2 Der Ausdruck `xs ++ ys` für zwei ausgewertete Listen der Länge $|xs|$ und $|ys|$ benötigt $O(|xs|+|ys|)$ Reduktionsschritte, wenn man nur das Rückgrat der Ergebnis-Liste auswertet. Der benötigte Zwischenspeicher ist in diesem Fall $O(|xs|)$, da nur das Rückgrat von `xs` kopiert werden muss.

Beispiel 2.2.3 Der Ausdruck `map f xs` für eine ausgewertete Liste der Länge n benötigt $O(n)$ Reduktionsschritte, wenn man nur das Rückgrat der Liste auswertet.

Wenn man auch alle Elemente der Ergebnisliste auswertet, hängt der Aufwand von f ab. Der Speicherbedarf ist im Fall der Rückgratauswertung $O(n)$, da das Rückgrat der Liste kopiert wird, im Fall der Auswertung aller Elemente hängt der Aufwand ebenfalls von f ab.

Beispiel 2.2.4 Der Ausdruck `concat xs` für eine ausgewertete Liste von Listen xs , wobei das i -te Element eine Liste mit n_i Elementen ist, benötigt folgende Anzahl Reduktionsschritte: $n_1 + 1 + n_2 + \dots + n_m + 1$, d.h. Anzahl der Elemente in der Eingabeliste + Anzahl der Elemente in der Ergebnisliste, wenn man nur das Rückgrat der Liste auswertet.

Der Speicherbedarf ist im Fall der Rückgratauswertung fast genauso hoch, wobei die letzte Liste nicht kopiert wird, und somit nicht berücksichtigt werden muss.

Beispiel 2.2.5 Eine Liste kann man so umdrehen:

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

Der Ressourcenbedarf ist quadratisch, d.h. $O(n^2)$, wenn n Länge der Liste und bei Rückgratauswertung. Die Begründung ist wie folgt:

Eine Liste $[a_1, \dots, a_n]$ wird nach einigen Reduktionsschritten zu

$([a_n] ++ [a_{n-1}]) ++ \dots ++ [a_1]$

Beim Auswerten der geschachtelten Append-Ausdrücke braucht man für den Teilschritt der $[a_n, a_{n-1}, \dots, a_{k-1}] ++ [a_k]$ auswertet, $k - 1$ Reduktionsschritte.

Zählt man das für alle k zusammen, so ergibt sich $1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}$, was insgesamt $O(n^2)$ ergibt.

Effizienter umdrehen kann man mit

```
reverse xs = foldl (\x y -> y:x) [] xs
```

oder mit der Funktion

```
reverse xs = reverse_lin xs []
reverse_lin [] ys          = ys
reverse_lin (x:xs) ys      = (reverse_lin xs (x:ys))
```

Beide ergeben einen iterativen Prozess für Listen (bei applikativer Reihenfolge der Auswertung).

```
reverse [1,3,5,7]
reverse_lin (1:3:5:7:[]) []
reverse_lin (3:5:7:[]) (1:[])
reverse_lin (5:7:[]) (3:1:[])
reverse_lin (7:[]) (5:3:1:[])
reverse_lin [] (7:5:3:1:[])
(7:5:3:1:[])
```


2.2.2 Listenausdrücke, List-Comprehensions

Dies ist eine Spezialität von Haskell und erleichtert die Handhabung von Listen. Die Syntax ist analog zu Mengenausdrücken, nur dass die Reihenfolge der Abarbeitung der Listenelemente in den Argumentlisten eine Rolle spielt, und die Reihenfolge der Listenelemente in der Ergebnisliste dadurch festgelegt ist.

Syntax:

$[(\langle \text{Ausdruck} \rangle \mid \{ \langle \text{Generator} \rangle \mid \langle \text{Filter} \rangle \}) \{ \{ \langle \text{Generator} \rangle \mid \langle \text{Filter} \rangle \} \}^*]$

Terminalzeichen sind '[' ']' und das Zeichen '|' als '|' geschrieben.

Vor dem senkrechten Strich „|“ kommt ein Ausdruck, danach eine mit Komma getrennte Folge von Generatoren der Form `v <- liste` oder von Prädikaten. Wirkungsweise: die Generatoren liefern nach und nach die Elemente der Listen. Wenn alle Prädikate zutreffen, wird ein Element entsprechend dem Ausdruck links von | in die Liste aufgenommen. Hierbei können auch neue lokale Variablen eingeführt werden, deren Geltungsbereich im Resultatausdruck ist und rechts von der Einführung liegt, aber noch in der Klammer [...].

Beispiel 2.2.6

<code>[x x <- xs]</code>	ergibt die Liste selbst
<code>[f x x <- xs]</code>	ist dasselbe wie <code>map f xs</code>
<code>[x x <- xs, p x]</code>	ist dasselbe wie <code>filter p xs</code>
<code>[(x,y) x <- xs, y <- ys]</code>	kartesisches Produkt der endlichen Listen <code>xs</code> und <code>ys</code>
<code>[y x <- xs, y <- x]</code>	entspricht der Funktion <code>concat xs</code>

Programmierhinweise: Wann kann man Listenkomprehensionen verwenden, wann nicht?

Generelle Beobachtung ist, dass man Eingabelisten als Erzeuger braucht, und dort auch Filterfunktionen platzieren kann. Man kann eine Liste erzeugen.

Die Ausgabe, d.h. die erzeugte Liste wird elementweise aus Elementen der Eingabelisten erzeugt. D.h. eine Listenkomprehension gibt einen Takt vor, wobei die erste Eingabeliste einmal durchgelesen wird, die zweite Eingabeliste wird pro Element der ersten Eingabelisten einmal durchgelesen, usw. In diesem Takt erhält man die Element der Ergebnislisten.

Was geht nicht mit List-Komprehensionen, bzw. nur sehr umständlich? :

- Paralleles Abarbeiten von mehreren Listen: das macht man besser mit den `zip`-Funktionen.

- Umsortieren von Listen: Dazu muss man eigene Funktionen schreiben: `reverse`, `quicksort`,
- Mehrere Elemente einer Liste werden gleichzeitig gebraucht, z.B. beim Entfernen von doppelten aus einer Liste: z.B. die Funktion `elimdub`, oder die Funktionalitäten der Faltungsfunktionen `foldr`, `foldl`. Auch beim Sortieren braucht man mehr als ein Element aus der gleichen Liste.

Hier die Funktion `elimdub`, die benachbarte doppelte Elemente eliminiert. (Das war die Funktionalität der ehemaligen Funktion `nub`, die aktuell in Haskell anders definiert ist. und zwar als Elimination aller doppelten.)

```
elimdub []      = []
elimdub [x]     = [x]
elimdub (x:(y:r)) = if x == y then  elimdub (y:r)
                      else x : elimdub (y:r)
```

Beispiel 2.2.7

`[quadrat x | x <- [1..20]]` ergibt Liste der Quadrate.

`[(x,y) | x <- [1..10], even x, y <- [2..6], x < y]`

Resultat: `[(2,3),(2,4),(2,5),(2,6),(4,5),(4,6)]`

Die Erzeugungsreihenfolge tabellarisch aufgelistet ergibt:

<i>x</i>	1	2	2	2	2	2	3	4	4	4	4	4	5	6	...
<i>y</i>		2	3	4	5	6		2	3	4	5	6		2	...
?	N	N	Y	Y	Y	Y	N	N	N	N	Y	Y	N	N	...

Ein weiteres Beispiel, das zeigt, dass die Elementvariable auch in der Listenerzeugung weiter rechts verwendet werden kann:

```
[(x,y) | x <- [1..10], y <- [1..10], y<x]
```


Das ergibt die Liste aller Paare (x, y) , so dass $1 \leq x, y \leq 10$ und $x < y$ ist. Der Nachteil der Comprehension oben ist, dass zuerst alle Paare erzeugt werden und dann die falschen wieder gefiltert werden. Besser ist es, die unerwünschten Paare nicht zu erzeugen. Das kann man mit Listen-Komprehensionen auch leicht programmieren:

```
*Main> [(x,y) | x <- [1..10], y <- [1..x]]
```


Die Ausgabe, editiert:

```
[(1,1),
 (2,1),(2,2),
 (3,1),(3,2),(3,3),
 (4,1),(4,2),(4,3),(4,4),
 (5,1),(5,2),(5,3),(5,4),(5,5),
 (6,1),(6,2),(6,3),(6,4),(6,5),(6,6),
 (7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),
 (8,1),(8,2),(8,3),(8,4),(8,5),(8,6),(8,7),(8,8),
 (9,1),(9,2),(9,3),(9,4),(9,5),(9,6),(9,7),(9,8),(9,9),
 (10,1),(10,2),(10,3),(10,4),(10,5),(10,6),(10,7),(10,8),(10,9),(10,10)]
```

Ein weiteres Beispiel, wenn die Laufvariable nicht im Term vorkommt:

```
*Main> [1 x j- [1..11]]— 
[1,1,1,1,1,1,1,1,1,1,1,1]
```

Liste der nicht durch 2,3,5 teilbaren Zahlen. Die erste Nicht-Primzahl darin ist 49:

```
*Main> [x|xj-[2..], x'rem'2/=0, x'rem'3/=0, x'rem'5/=0]— 
[7,11,13,17,19,23,29,31,37,41,43,47,49,53,59,61,67,71,73,77,...]
```

Beispiel 2.2.8 Man kann die Primzahlen mit einer rekursiv definierten Listenkompensation definieren:

```
primes = 2: [x | x <- [3,5..],
               and (map (\t-> x 'mod' t /= 0)
                       (takeWhile (\y -> y^2 <= x) primes))]
```

Man gibt die 2 als erste Primzahl vor, und kann dann die Liste wiederverwenden beim Test auf Teilbarkeit.

Übungsaufgabe 2.2.9 Ist es möglich, eine Sortierfunktion oder die Funktion `reverse` nur mittels Listkompensationen zu schreiben?

2.3 Felder, Arrays

Zuerst der Datentyp Assoziationsliste: Das ist eine Liste von Index-Wert Paaren. Hier kann man zu Schlüsseln die zugehörigen Werte verwalten. Typische Funktionen dazu sind: Eintragen, Löschen, Ändern, Abfragen.

, Z.B. kann man zu den Zahlzeichen die zugehörigen Werte sich in einer Assoziationsliste speichern: [('0',0), ('1',1), ... , ('9',9)].

```

mkassocl = []
ascfinde x [] = Nothing
ascfinde x ((xk,xw):rest) =
    if x == xk
    then Just xw
    else ascfinde x rest
asceinf x w [] = [(x,w)]
asceinf x w ((xk,xw):rest) =
    if x == xk
    then ((xk,w):rest)
    else (xk,xw) : (asceinf x w rest)
ascremove x [] = []
ascremove x ((xk,xw):rest) =
    if x == xk
    then rest
    else (xk,xw):(ascremove x rest)

```

Diese wird benutzt bei der Erstellung von Array in Haskell.

Felder (arrays, Vektoren) sind eine weitverbreitete Datenstruktur in Programmiersprachen, die normalerweise im einfachsten Fall eine Folge $a_i, i = 0, \dots, n$ modellieren.

Als Semantik eines Feldes A mit den Grenzen (a, b) und Elementen des Typs α kann man eine Funktion nehmen: $f_A : [a, b] \rightarrow \alpha$. In Haskell sind Felder als Zusatzmodul verfügbar. Die Elemente müssen gleichen Typ haben.

In Haskell gibt es als eingebaute Datenstruktur (als Modul) Arrays. Dies sind Felder von Elementen des gleichen Typs. Die Implementierung eines Feldes als Liste ist möglich, hätte aber als Nachteil, dass der Zugriff auf ein bestimmtes Element, wenn der Index bekannt ist, die Größe $O(\text{länge}(\text{liste}))$ hat.

Als Spezialität kann man beim Erzeugen verschiedene Typen des Index wählen, so dass nicht nur Zahlen, sondern auch Tupel (auch geschachtelte Tupel) von ganzen Zahlen möglich sind, womit man (mehrdimensionale) Matrizen modellieren kann. Einige Zugriffsfunktionen sind:

- **array** $x\ y$: erzeugt ein Feld (array) mit den Grenzen $x = (start, ende)$, initialisiert anhand der Liste y , die eine Liste von Index-Wert Paaren (eine **Assoziationsliste**) sein muss
- **listArray** $x\ y$: erzeugt ein array mit den Grenzen $x = (start, ende)$, initialisiert sequentiell anhand der Liste y .
- **(!)**: Infix funktion: **ar**!**i** ergibt das i -te Element des Feldes **ar**.
- **(//)**: Infix-funktion **a** **//** **xs** ergibt ein neues Feld, bei dem die Elemente entsprechend der Assoziationsliste xs abgeändert sind.
- **bounds**: Erlaubt es, die Indexgrenzen des Feldes zu ermitteln.

Beispiel 2.3.1

```
umdrehen_array ar =
  let (n,m) = bounds ar
      mplusn = m+n
  in ar // [(i,ar!(mplusn -i)) | i <- [n..m] ]
```

Beispiel 2.3.2 *Transponieren einer Matrix:*

```
transpose_matrix ar =
  let ((n1,m1),(n2,m2)) = bounds ar
      assoc1 = [(i,j), ar!(j,i))
                | j <- [n1..n2], i <- [m1..m2]]
  in array ((m1,n1), (m2,n2)) assoc1
```

2.4 Kontrollstrukturen, Iteration in Haskell

Wir können auch Kontrollstrukturen wie `while`, `until` und `for` definieren, wobei das Typsystem bestimmte Einschränkungen für den Rumpf macht.

Hier zeigt sich ein Vorteil von Haskell: man kann Kontrollstrukturen als Funktionen definieren, es ist unnötig, diese in die Sprache einzubauen.

Generell ist Rekursion allgemeiner als Iteration, allerdings sind vom theoretischen Standpunkt aus beide gleichwertig: wenn man den Speicher erweitern kann (d.h. beliebig große Datenobjekte verwenden und aufbauen kann), dann kann Iteration die Rekursion simulieren.

Die Definition dieser Kontrollstrukturen benutzt einen Datentyp „Umgebung“, so dass jeder Iterationsschritt die alte Umgebung als Eingabe hat, und die neue Umgebung als Ausgabe, die dann wieder als Eingabe für den nächsten Iterationsschritt dient. Diesen Effekt kann man auch beschreiben als: die Umgebung wird in jedem Iterationsschritt geändert. Der Rumpf ist eine Funktion, die genau diesen Schritt beschreibt.

Beispiel 2.4.1

```

while:: (a -> Bool) -> (a -> a) -> a -> a
--   while test f init
--   a: Typ der Umgebung
--   test: Test, ob While-Bedingung erfuehlt
--   f:: a -> a   Rumpf, der die Umgebung abaendert
--   init:   Anfangsumgebung

while test f init =
    if test init
    then while test f (f init)
    else init

untill :: (a -> Bool) -> (a -> a) -> a -> a
untill test f init =
    if test init
    then init
    else untill test f (f init)

for :: (Ord a, Num a) => a -> a -> a -> (b -> a -> b) -> b -> b
--   For laeuft von start bis end in Schritten von schritt
--   Umgebung:: b, Zaehler:: a
--   f:  Umgebung , aktueller Zaehler  -> neue Umgebung
--   f : init start -> init'
for start end schritt f init =
    if start > end
    then init
    else let startneu = start + schritt
         in for startneu end schritt f (f init start)

```

Die Funktion *f*, die als Argument mit übergeben wird, erzeugt ein neues Date-
nobjekt.

Verwendung im Beispiel:

```

dreinwhile n = while (> 1) dreinschritt n
dreinschritt x = if x == 1 then 1
                else if geradeq x then x 'div' 2
                else 3*x+1

--   berechnet 1 + 2+ ... + n:
summebis n = for 1 n 1 (+) 0

--   berechnet fibonacci (n)
fib_for n = fst (for 1 n 1 fib_schritt  (1,1))
fib_schritt (a,b) _ = (a+b,a)

```

Ein Ausdruck mit `foldl` lässt sich als `while`-Ausdruck schreiben:

```
foldl f e xs
```

ist äquivalent zu:

```
fst (while (\(res,list) -> list /= [])
      (\(res,list) -> (f res (head list), tail list))
      (e,xs))
```

Ein Ausdruck mit `foldr` lässt sich nicht so ohne weiteres als `while`-Ausdruck schreiben. Für endliche Listen ist das (unter Effizienzverlust) möglich:

```
f oldr f e xs
```

ist äquivalent (für endliche Listen) zu:

```
fst (while (\(res,list) -> list /= [])
      (\(res,list) -> (f (head list) res, tail list))
      (e,reverse xs))
```

2.5 Ein-Ausgabe in Haskell

Wir geben eine Kurzeinführung:

Haskell-Programme haben die Möglichkeit, über Aktionen mit der Außenwelt zu kommunizieren. Diese Aktionen sind nur über einen speziellen, vorgegebenen, aber parametrisierbaren Typ `IO a` möglich. Die Typen dieser Aktionen haben alle die Form `IO a` oder `a1 -> a2 -> ... -> IO a`, wobei `ai` und `a` nicht-IO-Typen sind.

Die Aktionen vermitteln zwischen der Außenwelt und der Auswertung von Ausdrücken. Die eigentliche funktionale Berechnung hat keine Ein/Ausgabe-Möglichkeit.

Ein/Ausgabe wird durchgeführt, indem man einen Ausdruck vom Typ `IO a` auswertet. Hierbei ist `a` der Typ des eingegebenen Objekts. Falls eine Ausgabe gemacht wird, ist das zugehörige Objekt und der Typ nur an den Argumenten der von Haskell vorgegebenen und verwendeten IO-Aktionen erkennbar.

Einen Ausdruck vom Typ `IO a` kann man z.B. dadurch konstruieren, dass man eine IO-Aktion mit Typ `b -> IO a` auf einen Ausdruck vom Typ `b` anwendet.

Hierbei sind die Basis-IO-Aktionen vordefiniert, und `a, b` sind Typen ohne IO-Untertypen. Im Typ `IO a` bezieht sich `a` als Argument des `IO` immer auf eine Eingabe. Wenn man kein Resultat einer Aktion hat, d.h. keine Eingabe ins Programm, dann ist der Typ `IO ()`.

Man kann eigene IO-Aktionen definieren, aber nur indem man einen Satz von bereits vordefinierten IO-Aktionen kombiniert. Z.B. sind vordefiniert:

```

putStr:: String -> IO ()
putStrLn:: String -> IO ()
getLine:: IO String
print:: (Show a) => a -> IO ()
readLn: (Read a) => IO a

type FilePath = String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath          -> IO String

```

`putStr`, `putStrLn` und `print` sind so definiert, dass diese ihr Argument gerade ausgeben. Dass gilt aber i.a. nicht bei beliebigen IO-Aktionen.

Die Aktionen `print` und `readLn` sind polymorph. Sie sollten im Programm so eingebettet sein, dass der Typ `a` vom Typ-Checker ermittelt werden kann. Der Typ muss dabei ein `Show`- bzw. `Read`-Typ sein. Ist der Typ bekannt, dann kann Haskell intern die richtige Methode, d.h. die dem Typ entsprechende, verwenden (siehe z.B. `getNumber` unten).

Beispiel 2.5.1 *Das Hello-World-Programm sieht so aus:*

```

*Main> putStr "Hello World" ↵
Hello World
*Main>

```

Kombinieren kann man Aktionen mit der `do`-Notation, die analog zu einer Listenkomprehension notiert wird, aber eine andere Wirkung hat.

Beispiel 2.5.2

```
do {input <- getLine; putStr input}
```

In Programm kann man das so schreiben:

```

do input <- getLine
  putStr input

```

Im Interpreter:

```

*Main> do {input <- getLine; putStrLn input} ↵
Hello ↵
Hello
*Main>

```


Die Kombination `do ... arg <- ...` wirkt wie ein Selektor, der aus einer IO-Aktion die Werte extrahiert.

Die Bedingung, dass alle Ausdrücke getypt sein müssen und dass man den „Inhalt“ der IO-Aktionen, nämlich den Eingabewert, nur mittels spezieller Selektoren im Programm explizit verwenden kann, bewirkt eine starke, aber gewollte Beschränkung der Programmierung der Ein-Ausgabe: Die referentielle Transparenz wird erhalten. d.h. Funktionen ergeben bei gleichen Argumenten auch gleiche Werte. Eine spürbare Beschränkung ist der durch das Typsystem und die Konstruktion und Verwendung des Typs `IO` erzwungene Programmierstil: Der Programmierer wird gezwungen, die IO-Aktionen zu sequentialisieren. Selbst definieren kann man zB die Aktion `echoLine`

```
echoLine:: IO String
echoLine = do
    input <- getLine
    putStr input
```

Eine Int-Zahl kann man einlesen mittels folgender Aktion

```
getNumber:: IO Int
getNumber = do
    putStr "Bitte eine Zahl eingeben:"
    readLn
```

Beispiel 2.5.3 *Mit folgender Definition kann man variable Listen ausgeben:*

```
main = do a <- getNumber
          b <- getNumber
          print (take a (repeat b))
```

```
*Main> main ↵
Bitte eine Zahl eingeben:4 ↵
Bitte eine Zahl eingeben:6 ↵
[6,6,6,6]
```

Beispiel 2.5.4 *Folgende Aktion bewirkt, dass ein File gelesen und ausgegeben wird.*

```
fileLesen = do
    putStr "File-Name:?"
    fname <- getLine
    contents <- readFile fname
    putStr contents
```

2.6 Modularisierung in Haskell

Module dienen zur

Strukturierung / Hierarchisierung: Einzelne Programmteile können innerhalb verschiedener Module definiert werden; eine (z. B. inhaltliche) Unterteilung des gesamten Programms ist somit möglich. Hierarchisierung ist möglich, indem kleinere Programmteile mittels Modulimport zu größeren Programmen zusammen gesetzt werden.

Kapselung: Nur über Schnittstellen kann auf bestimmte Funktionalitäten zugegriffen werden, die Implementierung bleibt verdeckt. Sie kann somit unabhängig von anderen Programmteilen geändert werden, solange die Funktionalität (bzgl. einer vorher festgelegten Spezifikation) erhalten bleibt.

Wiederverwendbarkeit: Ein Modul kann für verschiedene Programme benutzt (d.h. importiert) werden.

2.6.1 Module in Haskell

In einem Modul werden Funktionen, Datentypen, Typsynonyme, usw. definiert. Durch die Moduldefinition können diese exportiert Konstrukte werden, die dann von anderen Modulen importiert werden können.

Ein Modul wird mittels

```
module Modulname(Exportliste) where
    Modulimporte,
    Datentypdefinitionen,
    Funktionsdefinitionen, ... } Modulrumpf
```

definiert. Hierbei ist `module` das Schlüsselwort zur Moduldefinition, *Modulname* der Name des Moduls, der mit einem Großbuchstaben anfangen muss. In der *Exportliste* werden diejenigen Funktionen, Datentypen usw. definiert, die durch das Modul exportiert werden, d.h. von außen sichtbar sind.

Für jedes Modul muss eine separate Datei angelegt werden, wobei der Modulname dem Dateinamen ohne Dateiendung entsprechen muss.

Ein Haskell-Programm besteht aus einer Menge von Modulen, wobei eines der Module ausgezeichnet ist, es muss laut Konvention den Namen `Main` haben und eine Funktion namens `main` definieren und exportieren. Der Typ von `main` ist auch per Konvention festgelegt, er muss `IO ()` sein, d.h. eine Ein-/Ausgabe-Aktion, die nichts (dieses „Nichts“ wird durch das Nulltupel `()` dargestellt) zurück liefert. Der Wert des Programms ist dann der Wert, der durch `main` definiert wird. Das Grundgerüst eines Haskell-Programms ist somit von der Form:

```
module Main(main) where
    ...
    main = ...
    ...
```

Im folgenden werden wir den Modulexport und -import anhand folgendes Beispiels verdeutlichen:

Beispiel 2.6.1

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
                        else if a < b then Niederlage
                        else Unentschieden

  istSieg Sieg = True
  istSieg _    = False
  istNiederlage Niederlage = True
  istNiederlage _          = False
```

Modulexport

Durch die *Exportliste* bei der Moduldefinition kann festgelegt werden, was exportiert wird. Wird die Exportliste einschließlich der Klammern weggelassen, so werden alle definierten, bis auf von anderen Modulen importierte, Namen exportiert. Für Beispiel 2.6.1 bedeutet dies, dass sowohl die Funktionen `berechneErgebnis`, `istSieg`, `istNiederlage` als auch der Datentyp `Ergebnis` samt aller seiner Konstruktoren `Sieg`, `Niederlage` und `Unentschieden` exportiert werden. Die Exportliste kann folgende Einträge enthalten:

- Ein Funktionsname, der im Modulrumpf definiert oder von einem anderem Modul importiert wird. Operatoren, wie z.B. `+` müssen in der Präfixnotation, d.h. geklammert `(+)` in die Exportliste eingetragen werden.

Würde in Beispiel 2.6.1 der Modulkopf

```
module Spiel(berechneErgebnis) where
```

lauten, so würde nur die Funktion `berechneErgebnis` durch das Modul `Spiel` exportiert.

- Datentypen die mittels `data` oder `newtype` definiert wurden. Hierbei gibt es drei unterschiedliche Möglichkeiten, die wir anhand des Beispiels 2.6.1 zeigen:

- Wird nur `Ergebnis` in die Exportliste eingetragen, d.h. der Modulkopf würde lauten

```
module Spiel(Ergebnis) where
```

so wird der Typ `Ergebnis` exportiert, nicht jedoch die Datenkonstrukturen, d.h. `Sieg`, `Niederlage`, `Unentschieden` sind von außen nicht sichtbar bzw. verwendbar.

- Lautet der Modulkopf

```
module Spiel(Ergebnis(Sieg, Niederlage))
```

so werden der Typ `Ergebnis` und die Konstrukturen `Sieg` und `Niederlage` exportiert, nicht jedoch der Konstruktor `Unentschieden`.

- Durch den Eintrag `Ergebnis(...)`, wird der Typ mit sämtlichen Konstrukturen exportiert.
- Typsynonyme, die mit `type` definiert wurden, können exportiert werden, indem sie in die Exportliste eingetragen werden, z.B. würde bei folgender Moduldeklaration

```
module Spiel(Result) where
  ... wie vorher ...
  type Result = Ergebnis
```

der mittels `type` erzeugte Typ `Result` exportiert.

- Schließlich können auch alle exportierten Namen eines importierten Moduls wiederum durch das Modul exportiert werden, indem man `module Modulname` in die Exportliste aufnimmt, z.B. seien das Modul `Spiel` wie in Beispiel 2.6.1 definiert und das Modul `Game` als:

```
module Game(module Spiel, Result) where
  import Spiel
  type Result = Ergebnis
```

Das Modul `Game` exportiert alle Funktionen, Datentypen und Konstrukturen, die auch `Spiel` exportiert sowie zusätzlich noch den Typ `Result`.

Modulimport

Die exportierten Definitionen eines Moduls können mittels der `import` Anweisung in ein anderes Modul importiert werden. Diese steht am Anfang des Modulrumpfs. In einfacher Form geschieht dies durch

```
import Modulname
```

Durch diese Anweisung werden sämtliche Einträge der Exportliste vom Modul mit dem Namen *Modulname* importiert, d.h. sichtbar und verwendbar.

Will man nicht alle exportierten Namen in ein anderes Modul importieren, so ist dies auf folgende Weisen möglich:

Explizites Auflisten der zu importierenden Einträge: Die importierten Namen werden in Klammern geschrieben aufgelistet. Die Einträge werden hier genauso geschrieben wie in der Exportliste.

Z.B. importiert das Modul

```
module Game where
  import Spiel(berechneErgebnis, Ergebnis(..))
  ...
```

nur die Funktion `berechneErgebnis` und den Datentyp `Ergebnis` mit seinen Konstruktoren, nicht jedoch die Funktionen `istSieg` und `istNiederlage`.

Explizites Ausschließen einzelner Einträge: Einträge können vom Import ausgeschlossen werden, indem man das Schlüsselwort `hiding` gefolgt von einer Liste der ausgeschlossen Einträge benutzt.

Den gleichen Effekt wie beim expliziten Auflisten können wir auch im Beispiel durch Ausschließen der Funktionen `istSieg` und `istNiederlage` erzielen:

```
module Game where
  import Spiel hiding(istSieg,istNiederlage)
  ...
```

Die importierten Funktionen sind sowohl mit ihrem (unqualifizierten) Namen ansprechbar, als auch mit ihrem qualifizierten Namen: *Modulname.unqualifizierter Name*, manchmal ist es notwendig den qualifizierten Namen zu verwenden, z.B.

```
module A(f) where
  f a b = a + b

module B(f) where
  f a b = a * b

module C where
  import A
  import B
  g = f 1 2 + f 3 4 -- funktioniert nicht
```

führt zu einem Namenskonflikt, da `f` mehrfach (in Modul A und B) definiert wird.

```
Prelude> :l C.hs
ERROR C.hs:4 - Ambiguous variable occurrence "f"
*** Could refer to: B.f A.f
```

Werden qualifizierte Namen benutzt, wird die Definition von `g` eindeutig:

```
module C where
  import A
  import B
  g = A.f 1 2 + B.f 3 4
```

Durch das Schlüsselwort `qualified` sind nur die qualifizierten Namen sichtbar:

```
module C where
  import qualified A
  g = f 1 2    -- f ist nicht sichtbar
```

```
Prelude> :l C.hs
ERROR C.hs:3 - Undefined variable "f"
```

Man kann auch *lokale Aliase* für die zu importierenden Modulnamen angeben, hierfür gibt es das Schlüsselwort `as`, z.B.

```
import LangerModulName as C
```

Eine durch `LangerModulName` exportierte Funktion `f` kann dann mit `C.f` aufgerufen werden.

Abschließend eine Übersicht: Angenommen das Modul `M` exportiert `f` und `g`, dann zeigt die folgende Tabelle, welche Namen durch die angegebene `import`-Anweisung sichtbar sind:

Import-Deklaration	definierte Namen
<code>import M</code>	<code>f, g, M.f, M.g</code>
<code>import M()</code>	keine
<code>import M(f)</code>	<code>f, M.f</code>
<code>import qualified M</code>	<code>M.f, M.g</code>
<code>import qualified M()</code>	keine
<code>import qualified M(f)</code>	<code>M.f</code>
<code>import M hiding ()</code>	<code>f, g, M.f, M.g</code>
<code>import M hiding (f)</code>	<code>g, M.g</code>
<code>import qualified M hiding ()</code>	<code>M.f, M.g</code>
<code>import qualified M hiding (f)</code>	<code>M.g</code>
<code>import M as N</code>	<code>f, g, N.f, N.g</code>
<code>import M as N(f)</code>	<code>f, N.f</code>
<code>import qualified M as N</code>	<code>N.f, N.g</code>

Hierarchische Modulstruktur

Diese Erweiterung ist nicht durch den Haskell-Report festgelegt, wird jedoch von GHC und Hugs unterstützt³. Sie erlaubt es Modulnamen mit Punkten zu versehen. So kann z.B. ein Modul `A.B.C` definiert werden. Allerdings ist dies eine rein syntaktische Erweiterung des Namens und es besteht nicht notwendigerweise eine Verbindung zwischen einem Modul mit dem Namen `A.B` und `A.B.C`.

Die Verwendung dieser Syntax hat lediglich Auswirkungen wie der Interpreter nach der zu importierenden Datei im Dateisystem sucht: Wird `import A.B.C` ausgeführt, so wird das Modul `A/B/C.hs` geladen, wobei `A` und `B` Verzeichnisse sind.

Die „Haskell Hierarchical Libraries“⁴ sind mithilfe der hierarchischen Modulstruktur aufgebaut, z.B. sind Funktionen, die auf Listen operieren, im Modul `Data.List` definiert.

³An der Standardisierung der hierarchischen Modulstruktur wird gearbeitet, siehe <http://www.haskell.org/hierarchical-modules>

⁴siehe <http://www.haskell.org/ghc/docs/latest/html/libraries>