Programmieren III
Funktionale Programmierung

Dr. Werner Pohlmann

Wintersemester 97/98

Vorwort

Dieses Skriptum ist während der Vorlesung „Programmieren III, 1. Teil" zum Thema funktionale Programmierung entstanden. Der Inhalt wurde direkt von den Folien des Dozenten Dr. Werner Pohlmann übernommen, wobei kleinere Korrekturen während der Vorlesung berücksichtigt wurden. Getippt wurde dieser Text von den beiden Tutoren Werner Dietl und Ronald Blaschke im Satzprogramm LaTeX.

RCS Information

```
$Revision: 1.7 $
$Date: 1998/04/18 22:25:49 $
```

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Zweck der Vorlesung

- Grundkenntnisse / -fähigkeiten
- am Beispiel einer modernen funktionalen Sprache
- Verständnis für *Zweck, Bedeutung, Probleme*

1.2 Einschränkungen / Randbedingungen

- *sehr* enger zeitlicher Rahmen
- Kenntnisse in imperativer Programmierung, Algorithmen und Datenstrukturen sind vorausgesetzt!

1.3 Plan der Vorlesung

Kapitel 1	= Einführung
Kapitel 2	= Rundreise durch Haskell
Kapitel 2+n	= Vertiefung

1.4 Überblick „Höhere Programmiersprachen“

Man unterscheidet:

- imperative Programmiersprachen

  z.B.: Fortran (1958, Backus), Algol 60, Algol 68, Cobol, PL1, Pascal, Modula, ...

  Wesentlich: *Anweisungen* (→ „imperativ“) mit festgelegter *Ausführungsreihenfolge* (Kontrollstrukturen) ändern *Programmvariable.*

  (*Objektorientierte* Sprachen sind eine Form imperativer Sprachen; spezielle Definitionstechnik für Datenstrukturen ermöglicht evolutionäre / wiederverwendende Programmierung.)

  Vorbild imperativer Sprachen:

  von-Neumann-Rechner = aktive Zentraleinheit führt *Befehle* aus, die Werte in passivem *Speicher* aktualisieren (lesen / schreiben)

  Wegen der Nähe zur klassischen Rechnerstruktur sind imperative Sprachen sehr effizient zu implementieren!

- funktionale Programmiersprachen

  z.B.: Lisp (1960, McCarthy), Scheme, ML, FP, Hope, Miranda, ...

Charakteristisch:

- *keine* Anweisungen
- *keine* Variablen
- sondern: Programmierer definiert *Funktion* (abgestützt auf andere Funktionen, Konstantenvereinbarungen, ...) und spezifiziert Ziele der Berechnung durch damit gebildeten *Ausdruck*

Manche „funktionale Sprachen“ sind *nicht rein* funktional!

Vorbild: Mathematik, speziell λ-Kalkül; Auswertung von Ausdrücken durch *Reduktion*, d. h. schrittweise Transformation in die einfachste Form (= Wert).

Keine direkte Entsprechung zur üblichen Rechnerstruktur;

Grundsätzlich schwieriger zu implementieren / weniger effizient;

Heute allerdings sehr gute (kompilierende) Implementierungen mit (bei vernünftigen funktionalen Programmen) der imperativen Welt vergleichbaren Leistungen.

- logische Programmiersprachen

  insbesondere Prolog (1971, Colmerauer)

  Charakteristisch: gewünschtes Ergebnis wird mit Mitteln der Logik (also relational) definiert und durch ein Beweisverfahren mit Hilfe einer Wissensbasis abgeleitet.

  $\Longrightarrow$ Teil 2 von Programmieren III

  $\Longrightarrow$ funktionale Sprachen heißen auch *applikativ*, logische Sprachen auch *deklarativ*; (wobei letzterer Term auch für funktionale Sprachen benutzt wird!)

1.5 Funktionale Sprachen – Was, Wie, Warum?

- **Ursprung:** Lisp = Überlegungen aus der Theorie der Berechenbarkeit praktisch machen;
  - *Anwendungen* zunächst beschränkt auf künstliche Intelligenz und Hochschulen;
  - seit den 70er Jahren neues und wachsendes Interesse an funktionaler Programmierung;
  - Entwicklung von Sprachen, die durch angebotene Mittel und Implementierung auf *Softwaretechniker* allgemein zielen.

- **Gründe**:

  1. Kritik an imperativen Sprachen:
     Semantik (Wirkung, Verständnis vom Programm) durch *sich in der Zeit ändernden Zustand* (Stand der Bearbeitung, Variablenbelegung) gegeben – und das ist zu schwer für das menschliche Hirn!
     Beispiel: Partitionierung in Quicksort:

     ```
     i:=links; j:=rechts-1;

     loop
       while a(i)<trenn loop i:=i+1; end loop;
       while a(j)>trenn loop j:=j-1; end loop;

       if i>=j then exit; end if;

       temp:=a(i); a(i):=a(j); a(j):=temp;

       i:=i+1; j:=j-1;
     end loop;
     ```

     ?? Was gilt hier ??
     ?? Kommt man überhaupt dahin ??
     imperative Programmierung ist mühsam, logisch anspruchsvoll und fehleranfällig, denn:

– Verhältnis von Zuweisung ":=" und mathematischer Gleichheit "=" ist (nicht nur für Anfänger) problematisch,

– zwar reichliche Verwendung von Ausdrücken in den Anweisungen, aber wegen mangelnder referentieller Transparenz (Kontextabhängigkeit) großer Unterschied zu üblicher (mathematischer) Interpretation, z. B.:

```
i: integer := 1;
a: array(1..10) of integer := (others=>0);

function f return integer is          -- Seiteneffekt
   begin i:=i+1; return i; end f;
```

dann z. B.:

```
i:=i+f;
a(i):=f;
a(f):=a(f);
a(i+f):=i+f;
if 2*(i+f)=(i+f)+(i+f) then ...
...
```

Klar: Ziel nicht, Reihenfolge der Auswertung in Gebilden wie "a(f):=a(f)" zu normieren, sondern das Problem zu vermeiden!

2. positive Beiträge funktionaler Programmierung: andere Formen von *Abstraktion & Modularisierung*

   – Funktionen als „Bürger erster Klasse", also z. B.:
     * Vektor von Funktionen,
     * Funktion als Ergebnis und Parameter anderer Funktionen
       ⇒ „higher order functions"

   vgl. „map", „fold" in Programmieren II
   vgl. Mathematik:
   Differentiation als Transformation von reellen Funktionen, also:

```
diff :: (Float->Float) -> (Float->Float)
diff(sin) = cos
```

   ist in funktionaler Programmierung *numerisch* ohne weiteres machbar!

   – verzögerte Auswertung („lazy evaluation") = rechne nur soviel, wie für das gewünschte Ergebnis nötig.
   ⇒ effiziente Kombination vom Programmteilen, die *ohne* Rücksicht auf spezifische (beschränkte) Anwendung entworfen sind.

Beispiel:	Test ob Zahl n prim ist
Annahme:	es gebe Funktion `divisors :: Int -> [Int]` die *Liste aller Teiler* einer gegebenen Zahl liefert
dann:	`is_prim n = divisors n == [1,n]`
wobei:	falls n keine Primzahl ist wird `divisiors` nur *solange ausgewertet, bis* erstes Gegenbeispiel gefunden wurde.
also:	Abbruchbedingung für `divisors` *implizit*, also `divisors` für ganz verschiedene Zwecke nutzbar (vgl.: wie macht man das in imperativer Programmierung ??)

Ferner: lazy evaluation erlaubt Arbeit mit (potentiell) unendlichen Datenstrukturen (Listen, Bäumen, . . .). !

1.6 Haskell

(Name kommt vom Mathematiker Haskell Curry)

- ist moderne, anspruchsvolle, umfangreiche funktionale Sprache,
- seit Anfang der 90'er von einer Gruppe von führenden Wissenschaftlern (UK, USA, Dk) entwickelt und weiterentwickelt, jetzt Version 1.4,
- mehrere frei zugängliche Implementierungen (Compiler für Unix, Interpreter auch für andere Systeme), die an Universitäten (Chalmers, Glasgow, Yale) entwickelt wurden und „industrial strength" aufweisen,
- 2 für Ausbildungszwecke, aber auch für Entwicklungsarbeiten, angenehme interpretierende Umgebungen (Unix + PCs):
  - Gofer (Spezialvariante)
  - Hugs (große Teilmenge von Haskell)

1.7 Literatur

- R. Bird, P. Waller:
  „Introduction to Functional Programming",
  Prentice Hall, 1988
  (klassisches Lehrbuch von zwei Spitzenleuten, benutzt nicht Haskell, aber ähnliche Notation)
- S. Thompson:
  „Haskell, the craft of Functional Programming",
  Addison Wesley, 1996
  (umfassendes, an Anfängern orientiertes Lehrbuch)
- A. J. T. Davies:
  „Functional Programming using Haskell",
  Cambridge University Press 1992
  (als Einführung zu knapp, aber weiterführende Themen)
- J. Hughes:
  „Why Functional Programming Matters",
  The Computer Journal 32/2, 1989
  (Zeitschriftenartikel, der Ziele der funktionalen Programmierung illustriert)

Letzteres ist im Haus elektronisch verfügbar; dito:

- Haskell-Report,
- Hudak & Fasel: „A Gentle Introduction to Haskell"

  (Obacht: viel lesbarer als Report, für Anfänger trotzdem nicht ganz leicht ...)

Viel Material im WWW verfügbar; gutes Archiv und Startadresse für Weiteres ist Uni Aachen:
`http://www-i2.informatik.rwth-aachen.de/Forschung/FP/Haskell/`

Kapitel 2

Kurze Rundreise durch Haskell

2.1 Sitzungen mit Hugs

Hugs ist ein Interpreter mit dem typischen „read-evaluate-write"-Zyklus;
Hugs wertet aus:

- Haskell Ausdrücke

  abgestützt auf Funktionssammlung im Standard-Prelude und/oder vom Benutzer geladenen Dateien mit Haskell Definitionen.

  Ausdrücke werden durch „Return" abgeschlossen! !

- durch ":" eingeleitete Kommandos, z. B.:

```
:?       -- listet Kommandos auf
:quit    -- beendet die Sitzung
```

Einige Beispiele:

```
1 pohlmann@okapi:hugs
      ___    ___   ___    ___   __________   __________
     /  /   /  /  /  /   /  /  /  _______/  /  _______/       Hugs 1.4
    /  /___/  /  /  /   /  /  /  / _____   /  /______
   /  ____   /  /  /   /  /  /  / /_   /  /______   /  The Nottingham and Yale
  /  /   /  /  /  /___/  /  /  /___/  /  _______/  /    Haskell User's System
 /__/   /__/  /_________/  /_________/  /_________/        Version 970410

   Copyright (c) The University of Nottingham and Yale University, 1994-1997.
    Bug reports: hugs-bugs@haskell.org.   Web: http://www.haskell.org/hugs.

Reading file "/usr/local/lib/hugs/lib/Prelude.hs":

Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
Type :? for help
Prelude> 37-(2+5)
30
Prelude> 27-2+5
30
Prelude> min 3 4
3
Prelude> 3==4
False
```

```
Prelude> sum [1..10]
55
Prelude> reverse "Salzburg"
"grubzlaS"
Prelude>
```

Natürlich gibt's auch Fehlermeldungen — für syntaktisch falsche Ausdrücke, undefinierte Bezeichner oder unanständige Wünsche, ...

1. ```
   Prelude> 37-(2+5

   ERROR: Syntax error in expression (unexpected end of input)
   ```
2. ```
   Prelude> 37-(2+n)
   ERROR: Undefined variable ``n''
   ```
3. ```
   Prelude> let n=5 in 37-(2+n)  --so geht's, und dies ist ein Kommentar!
   30
   ```
4. ```
   Prelude> 37-True
   ERROR: Bool is not an instance of class ``Num''
   ```
5. ```
   Prelude> 37`div`(5-5)

   Program error: primDivInt 37 0
   ```
6. ```
   Prelude> let x=x in x
   ERROR: Cannot find ``show'' function for:
   *** expression : let ... in x
   *** of type    : a
   ```
7. ```
   Prelude> let x::Int; x=x in x
   Interrupted!                         -- mit Control-C
   ```

Richtige Programmierer wollen natürlich ihre eigenen Funktionen, Konstanten, Datentypen, ... definieren. Solche „Script" genannte Sammlungen von Definitionen werden in eine Datei geschrieben und vor Gebrauch geladen:

```
:load bla.bla
```

es ist sinnvoll, das Suffix ".hs" zu verwenden (es gibt Werkzeuge, die das brauchen);
außer solchen unstrukturierten Sammlungen gibt es auch einen *Modulbegriff* mit entsprechenden Bezügen (Export / Import), mehr dazu später!

Beispiel: Datei `bla.bla`:

```
area r=pi*r*r
  where pi=3.1416
```

Datei `error.bla`:

```
area r=pi*r*r
  were pi=3.1416
```

Damit folgende Hugs-Sitzung:
```

```
Prelude> :load bla.bla
Reading file "bla.bla":

Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
bla.bla
Main> area 10
314.16
Main> :load error.bla
Reading file "error.bla":
Parsing
ERROR "error.bla" (line 2): Syntax error in input (unexpected `=')
```

Fehlermeldungen sind manchmal etwas schwierig zu interpretieren, woran oft die ungewohnte Typ-Philosophie von Haskell schuld ist (siehe später).

Im ersten Beispiel gilt:

1. fehlende Klammer
2. n ist keine „Programmvariable“, sondern eine Variable im üblichen *mathematischen* Sinn, d. h. ein *Konstantenbezeichner*, wobei der Wert natürlich willkürlich ist, aber festgelegt werden muß,
3. diese Festlegung kann man mit “let” machen,
4. Typfehler: die Subtraktionsoperation ist (überladen!) für mehrere numerische Typen definiert, aber `Bool` ist kein numerischer Typ
5. Division durch 0 (`` `div` `` ist abgestützt auf eine speziellere elementare Divisionsoperation)
6. `let x=x in x` führt auf Endlos-Schleife (=Wert von x ist undefiniert, wird aber „gesucht“;

   hier: Typ von x unklar, deshalb kann nicht die geeignete Funktion “show” zur Ausgabe des Ergebnisses bestimmt werden,
7. hier wird das Ergebnis als “Int” deklariert, und schon sind wir in der Endlos-Schleife ...

beachte:

- eine Gleichung wie `pi = 3.14` oder `area r = ...` bindet einen *Bezeichner* an einen *Wert* (insbesondere ist also pi Konstant, nicht Variable)
- der Gebrauch einer “where”-Klausel macht `pi` *lokal* zur Definition von `area` (⇒ später);

  grundsätzlich ist ein Skript eine *Menge* von Definitionen, die sich aufeinander beziehen können, aber z. B. auch:

```
pi = 3.14      -- besser umbenennen: name clash mit Prelude!
area r = pi * r * r
circumference r = 2 * pi * r
```

- beim Laden werden die Definitionen *analysiert* und *übersetzt*, aber werden auch bei dieser Gelegenheit *Syntaxfehler* im Skript gemeldet!

2.2 Grundlegende Sprachelemente

2.2.1 Bezeichner

Bezeichner für Funktionen (und Konstante = 0-stellige Funktionen) beginnen mit Kleinbuchstaben, gefolgt von Buchstaben, Ziffern, “_”, “'”.

- Operatoren = 2-stellige Funktionen mit Infix-Gebrauch werden durch Sonderzeichen gekennzeichnet

  ⇒ später,

- es gibt einige reservierte Wörter und Sonderzeichenkombinationen,
  ⇒ Report
- Bezeichner mit Großbuchstaben am Anfang werden für Typen (und zur Konstruktion derselben) verwendet, z. B. Int, Float, Bool

Der Prelude enthält viele in Haskell definierte *Standardfunktionen* (und Operatoren), die sich wiederum auf gewisse *primitive* Funktionen abstützen
⇒ Report !

2.2.2 Kommentare

= "`--`" und Zeilenrest;
alternativ geht auch Einklammern mit "`{-`" und "`-}`".

2.2.3 Klammerung / Klammerersparnis

ist weitgehend an mathematischer Praxis orientiert, mit einer wichtigen Ausnahme:
Funktionsapplikation ist so häufig und fundamental, daß sie *höchste* Präzedenz bekommt (aber keine Klammerung nötig)
⇒ bequem aber tückisch (für Anfänger)

`area 10 + 10` **wird interpretiert als**
`(area 10) + 10` **⇒ 324.16**
Will man `area(10+10)`**, muß man das auch so schreiben !!**
Mehr zum Thema später! !

2.2.4 Layout

Man kann in Haskell das Semikolon als *Trennzeichen* (und geschwungene Klammern zur Gruppierung ähnlich "begin - end") nehmen,
aber es gibt eine bequemere *Layout-Konvention* die *Abseits-Regel* (offside-rule):

- ist eine Zeile *genauso weit* eingerückt wie die vorherige, so bringt sie eine *neue* Definition,
- ist eine Zeile *weiter* eingerückt, so enthält sie eine Fortsetzung der vorher begonnenen Definition,
- ist eine Zeile *weniger* weit eingerückt, so beginnt eine neue Gruppe von Definitionen

Beispiel: Das Programm

```
pi = 3.1416
   area r = pi * r * r
```

gibt einen Syntaxfehler!!

Richtig sind:

```
pi = 3.1416;
   area r = pi * r * r
```

oder:

```
pi = 3.1416
area r = pi * r * r
```

Achtung:

- **Programmdateien besser nicht mit Proportionalschrift-Editor schreiben,**
- **Verstöße gegen offside-rule ergibt manchmal (für den Anfänger) schwer verständliche Fehlermeldungen** !

2.2.5 Lokale Definitionen

Mit `where`-Klauseln kann man *Definitionen* qualifizieren, d. h. auf eine untergeordnete Definition stützen:

```
area r = pi * r * r
  where pi = 3.14
```

Mit `let` kann man was Entsprechendes für *Ausdrücke* tun:

```
let x = 1 in x + 1

let x = 3
    f y = x + y
in f x
```

Damit dann Frage nach *Gültigkeitsbereich / scope:*

1. Gültigkeit einer gewöhnlichen Definition ist das *ganze Skript*, in dem sie enthalten ist;
   Reihenfolge der Aufschreibung ist egal!
2. untergeordnete Definitionen gelten nur lokal, d. h. scope ist durch die übergeordnete Definition gegeben:

   ```
   two_areas x y = area x + area y
      where area x = pi * square x
         where pi = 3.14
               square x = x * x
   ```

   beachte:

   - zweite Schachtelung im Beispiel nicht nötig – man kann alles „flach" und in beliebiger Reihenfolge aufschreiben,
   - Wiederverwendung des Bezeichners x als Parameter *verdeckt* natürlich die frühere Einführung – es gilt immer die „lokalste" Version
   - Layout, *Layout*, ***Layout***

2.2.6 Typen

Ausdrücke bezeichnen *Werte*
(Auswertung eines Ausdrucks versucht, ihn auf seine einfachste äquivalente Form zu *reduzieren*, also z. B.

```
3 + 4     => 7
abs(-2)   => 2
```

)
und jeder Wert hat einen *Typ*, der wie folgt angegeben werden kann:

```
   3 :: Int
True :: Bool
 abs :: Int -> Int
```

Solche *Typdeklarationen* kann der Programmierer explizit anschreiben, insbesondere bei der Einführung von Bezeichnern:

```
  pi   ::  Float
  pi   =   3.14
area   ::  Float -> Float
area x =   pi * x * x
```

Er *muß* es aber *nicht* !

Trotzdem hat Haskell ein *statisches* Typsystem (= Typ steht zur Compile-Zeit, also vor der Auswertung, fest): Haskell betreibt neben Syntaxanalyse noch *Typ-Inferenz*, d. h. erschließt den Typ von Ausdrücken aus Teilen und weist den Ausdruck als *fehlerhaft* zurück, falls kein vernünftiger Typ gefunden werden kann.

Empfehlung: verwende Typ deklarationen

- zur Dokumentation / Lesbarkeit
- gegen Fehler (Haskell vergleicht Typdeklarationen mit erschlossenem Typ)

Es gibt die üblichen *Basistypen* wie

```
Int, Float, Bool, Char
```

mit Elementen wie

```
3, 3.14, True, 'A'
```

und Operationen wie

```
+, *, .. , &&, ||, ..
```

und es gibt *strukturierte* Typen wie z. B. `[Int]` oder `[Char]` = Typen der Listen von Int- oder Char-Elemente mit Elementen wie `[1,2,3]`, `['A', 'B', 'C']`.

`[]` = leere Liste,

Operator "`:`" zum Anfügen eines Elements *vorne*:

```
'A':['B','C']     ist     ['A','B','C']
```

und der Konkatenation "`++`":

```
['A'] ++ ['B','C']
```

ist das selbe wie oben.

Zusatz: `[Char]` ist dasselbe wie der Typ `String`, String-Elemente können wie üblich auch mit Anführungszeichen gebildet werden: `"ABC"`

$\Longrightarrow$ Mehr über

Basistypen	$\Rightarrow$ Report, Literatur
Listen	$\Rightarrow$ späteres Kapitel
benutzerdefinierte Typen	$\Rightarrow$ späteres Kapitel

Mit dem Hugs-Kommando "`:type`" kann man den Typ eines Ausdrucks erfragen:

```
> :type 'A'
'A' :: Char
> :type "hallo"
"hallo" :: String
```

Das Ergebnis sieht aber manchmal anders aus als erwartet:

```
> :type 3
3 :: Num a => a
> :type area
area :: Fractional a => a -> a
```

Ähnliches gilt für Meldungen bei Typfehlern! !

und verweist auf einen weiteren wichtigen Aspekt von Haskells Typ-System:

- Polymorphismus

  = ein Haskell-Typ kann mit anderen Typen *parametrisiert* sein.

  Beispiel: Die Identitätfunktion ist für *jeden* Typ sinnvoll; man darf daher definieren

  ```
  id x = x
  ```

  was den Typ

  ```
  id :: a -> a
  ```

  mit Parameter "a" (Kleinbuchstaben!) ergibt, der für beliebige andere Typen steht, also möglich:

  ```
  id 3, id 'A', id [1,2,3]
  ```

  neben parametrischem oder universellem Polymorphismus gibt es auch noch

- Überladen (overloading, ad-hoc-Polymorphismus)

  = das gewünschte Verhalten ist für mehrere, aber *nicht* alle Typen möglich und erfordert i. A. jeweils verschiedene Implementierungen.

  Beispiele:

  - Gleichheitsoperator "=="
    ("=" ist Definitionszeichen!)
    und erfordert für Listen z. B. ganz andere Definition als für die diversen Basistypen.
  - Zahlenbezeichner wie z. B. "3" stehen für verschiedene Zahlbereiche (Int, Float, Genauigkeit), ähnlich arithmetische Operatoren wie "+".

  Der Typinferenzmechanismus ermittelt deshalb einen parametrisierten Typausdruck mit einer *Einschränkung* des Parameters: der erlaubte Typ muß zu einer Klasse gehören, die die im betrachteten Ausdruck vorkommenden Operationen usw. unterstützt.

  Beispiel:

  ```
  > :type 3
  3 :: Num a => a
           -- lies: 3 hat Typ a, wobei a ein numerischer Typ ist

  > :type area
  area :: Fractional a => a -> a
           -- lies: area ist eine Funktion vom Typ a->a, wobei a ein
           -- Typ für "reelle" Zahlen ist, also die in der Definition
           -- von "area" steckende Multiplikation mit 3.14 verkraftet
  ```

  ⇒ mehr zum Thema später!

  Polymorphismus & Overloading nicht verwechseln mit den ebenso benannten Dingen in Ada! !

2.2.7 Typen von Funktionen

- higher order functions

  = Funktionen mit Funktionen als Argument und Resultat

  z. B. (numerische) Differentiation:

```
diff :: (Float -> Float) -> (Float -> Float)
diff f = g
  where g x = ( f (x + delta) - f x ) / delta
        delta = 0.001
```

  (Obacht: Beispiel ist *numerisch nicht* das Feinste!)

```
> diff sin 0
1.0
> diff cos (pi/2)
-1.00005
>diff (diff cos) 0          -- 2. Ableitung
-1.07288
```

  Beachte Bindekraft und Linksassoziierung von Funktionsanwendung! !

```
diff sin 0      -- ist (diff sin) 0,
                -- und diff sin :: Float -> Float
                -- ( => :type Kommando!)
```

  Sowas wie

```
diff diff cos 0
```

  wird deshalb gegliedert zu

```
(diff diff) cos 0
```

  und ergibt *Typfehler!*

- Currying

  (Konzept aus der Mathematischen Logik & Grundlagenforschung; erfunden von Schönfinkel, popularisiert durch Haskell Curry)

  Funktionen mit *mehreren* Argumenten werden meist aufgefaßt als Funktionen mit *Tupel* als Argument, z. B.:

  Volumen :: $(\mathbf{R} \times \mathbf{R} \times \mathbf{R}) \quad \to \mathbf{R}$
  $\quad$ (länge, breite, höhe) $\mapsto$ länge * breite * höhe

  Haskell benutzt andere und äquivalente Auffassung:

```
volume :: Float->Float->Float->Float
volume l b h = l*b*h
```

  deren besonderer Nutzen darin liegt, daß die Argumente sukzessive einzeln kommen können d. h. *partielle* Funktionsanwendung ist möglich (und ergibt natürlich eine Funktion der *restlichen* Argumente!):

```
> volume 1 1
<< Function >>
> :type volume 1 1
volume 1 1 :: Float -> Float
```

Die durch partielle Anwendung resultierende Funktion läßt sich wie üblich verwenden, z. B.:

```
> diff (volume 1 1) 47.11
0.949451
```

und zur Bequemlichkeit auch mit einem Namen versehen, z. B.:

```
volume_per_unitsquare = volume 1 1
```

anderes Beispiel: Differentiation mit variierbarer Genauigkeit:

```
multi_diff :: Float -> (Float -> Float) -> (Float -> Float)
multi_diff d f x = (f (x+d)- f x) / d
```

dann kann man definieren:

```
coarse_diff = multi_diff 10.0
fine_diff   = multi_diff 1.0e-3    -- nicht zu klein machen!!
```

und rechnen:

```
> coarse_diff cos (pi/2)
0.0544021
> fine_diff cos (pi/2)
-1.00005
```

Damit partielle Anwendung funktioniert, müssen Funktionstypen *nach rechts* assoziieren:

```
        volume :: Float -> Float -> Float -> Float
-- ist:           Float -> (Float -> (Float -> Float))

    multi_diff :: Float -> (Float -> Float) -> (Float -> Float)
-- ist            Float -> ((Float -> Float) -> (Float -> Float))
```

Hier noch mal ausführliche Beispiele:

```
-------------------------------POLYMORPHISM, list-type examples

Prelude> :type []
[] :: [a]
Prelude> :type (:)
(:) :: a -> [a] -> [a]
Prelude> :type 'a':[]
'a' : [] :: [Char]
Prelude> 'a':[]
"a"
Prelude> :type ['a','b','c']:[]
['a','b','c'] : [] :: [[Char]]
Prelude> ['a','b','c']:[]
["abc"]
Prelude> :type ['a','b','c']:['a','b','c']
ERROR: Type error in application
```

```
*** expression     : ['a','b','c'] : ['a','b','c']
*** term           : ['a','b','c']
*** type           : [Char]
*** does not match : Char
Prelude> :type ['a','b','c']:[['a','b','c']]
['a','b','c'] : [['a','b','c']] :: [[Char]]
Prelude> ['a','b','c']:[['a','b','c']]
["abc", "abc"]
Prelude> :type length
length :: [a] -> Int
Prelude> length []
0
Prelude> length ['a']
1
Prelude> length [1000, 10000]
2
Prelude> length ( ['a','b','c']:[['a','b','c']])
2


-------------------HIGHER ORDER FUNCTION EXAMPLE

tabulate :: Float->Float->Int->(Float->Float)->[Float]
tabulate start step times f
 |times <=0 = []
 |otherwise = (f start):(tabulate (start+step) step (times-1) f)



Main> tabulate (-1) 0.2 11 sin
[-0.841471, -0.717356, -0.564642, -0.389418, -0.198669, -2.98023e-08,
0.198669, 0.389418, 0.564642, 0.717356, 0.841471]


Main> :type diff
diff :: Fractional a => (a -> a) -> a -> a
Main> :type (diff cos)
diff cos :: Floating a => a -> a
Main> tabulate (-1) 0.2 11 (diff cos)
[0.8412, 0.716984, 0.564218, 0.38898, 0.198126, -0.000476837, -0.199199,
-0.389874, -0.565052, -0.717699, -0.841737]


Main> let around_zero = tabulate (-1) 0.2 11 in around_zero sin
[-0.841471, -0.717356, -0.564642, -0.389418, -0.198669, -2.98023e-08,
0.198669, 0.389418, 0.564642, 0.717356, 0.841471]
```

2.2.8 Funktionsdefinitionen

sind natürlich die Seele vom Geschäft und auf verschiedene Weise möglich, wobei man immer auf *bereits vorhandenes Material* aufbaut;

formell besteht eine Funktionsdefinition aus dem Definitionszeichen "=",
einer *linken* Seite, die den *Namen* der Funktion und (wenn zur Definition nötig) formalen Parametern besteht,
einer *rechten* Seite, die (evtl. unter Benutzung der formalen Parameter) ein *Ausdruck* von entsprechendem Typ ist.

Beispiele:

```
inc :: Int -> Int      -- Typdeklaration
```

```
inc x = x + 1          -- Funktionsdefinition
        -----          -- Ausdruck, der benutzt:
                       -- formalen Parameter x,
                       -- vordefinierte Funktionen x und 1

k_aus_n :: Int -> Int -> Int
k_aus_n k n = fac n `div` (fac k * fac (n-k))

negative :: Num(a) => a -> Bool
negative x = x < 0

is_zero :: Num(a) => a -> Bool
is_zero x = x==0
```

Keine formalen Parameter benötigt man, wenn die neue Funktion eigentlich eine alte ist ...

```
circle_area :: Float -> Float
circle_area = area

coarse_diff :: (Float -> Float) -> (Float -> Float)
coarse_diff = multi_diff 10.0        -- partielle Anwendung, vgl. vorher
```

Im allgemeinen enthält die rechte Seite einer Definition eine

- Fallunterscheidung
  = die einzelnen Definitionsfälle werden von *„Wächtern"* (guards, Boolesche Ausdrücke) gestützt;
  es wird der (im Sinne der Aufschreibungsreihenfolge) erste Fall genommen, dessen Wächter sich zu True errechnet.

  Beispiele:

  ```
  max2 :: Int -> Int -> Int
  max2 x y | x >  y = x
           | x <= y = y
  ```

  dafür geht auch:

  ```
  max2 x y | x > y     = x
           | otherwise = y

  max3 :: Int -> Int -> Int -> Int
  max3 x y z | x>=y && x>=z = x
             | y>=x && y>=z = y
             | otherwise    = z
  ```

  usw.

- pattern matching
  Beim Aufruf einer Funktion werden *aktuelle* Parameter (=Ausdrücke, Werte) an die Stelle der *formalen* Parameter (bloße Bezeichner) gesetzt:

  max2 17 (2*3)
  $\downarrow$ $\downarrow$
  x y

  Man kann dieses übliche „matchen" (auf einander beziehen) von formalen und aktuellen Parametern auch noch *verallgemeinern:*

  Lasse zu, daß ein formaler Parameter ein *Muster* (pattern) ist, d. h. etwas *Strukturiertes.*

  Typische Beispiele bieten Listen-Datentyp:
  eine Liste ist entweder die *leere Liste* `[]`, *oder* sie hat ein Listenelement `x` gefolgt von einer *Restliste* `xs`, d. h. hat den Aufbau `x:xs`;
  das nimmt man jetzt zur Definition;

Beispiel:

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

Beachte:

- für jeden pattern-Fall eigene Gleichung;
- bei Anfang wird erst passende Gleichung (d. h. passendes Muster) genommen; gibt es keine Übereinstimmung, resultiert ein Laufzeitfehler;
- Witz an der Sache ist, daß in Verallgemeinerung der formal/aktuell-Beziehung ein *Zugriff auf Teile* (Konstituenten) des aktuellen Parameterwerts ermöglicht wird:
  im Beispiel werden Kopf und Schwanz der Liste an `x` und `xs` gebunden:

$$\begin{array}{cccccl} [& 1 & , & \underbrace{2,3} &] & \text{aktuelle Parameter} \\ & \downarrow & & \downarrow & & \\ & \text{x} & & \text{xs} & & \text{pattern-Teile} \end{array}$$

Was ist für patterns erlaubt?

- Variable (=formale Parameter) und Konstante (=numerische Literale, Char- , Bool-Elemente),
- Listen, von denen die Elemente wieder Patterns sind, also sowas wie `[True, b1, b2]`,
- mit ":" gebildete Listenausdrücke (siehe oben),
- mit "+" gebildete arithmetische Ausdrücke der Form `n+k`, wobei `k` konstant und `n-k>=0` ist,
- ähnlich: Ausdrücke der Form `k*n`,
- das wild-card-pattern "_" auf das alles paßt

Beispiel:

```
vorgaenger :: Int -> Int
vorgaenger 0     = 0              -- naja
vorgaenger (n+1) = n
```

dann folgt:

```
> vorgaenger 77
76
> vorgaenger (2-3)
Programm Error: vorgaenger(-1)
```

denn: (-1) läßt sich nicht in der verlangten Weise als n+1 darstellen, und einen Fall für negative Werte gibt es nicht.

Beispiel: für beliebige Listentypen:

```
head (x:_)  = x           -- Rest der Liste interessiert nicht
                          -- Keine Definition für leere Liste!!

tail (_:xs) = xs          -- Kopf interessiert nicht,
                          -- nicht definiert für leere Liste!!
```

Manchmal will man eine Funktion konstruieren, ohne sie einem Bezeichner zuzuweisen, d. h. die Funktion bleibt anonym.

Beispiel: partielle Anwendung einer Funktion, also sowas wie `volume 1 1`.

Weitere wichtige Möglichkeiten sind:

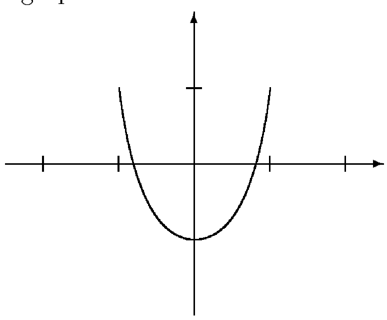
- Lambda-Abstraktion
  gegeben: ein Ausdruck `exp` und ein Variablenbezeichner `x`

  dann: `\x -> exp` ist diejenige Funktion, die einem Wert v den Wert exp $\frac{v}{x}$ (d.h. v ersetzt x in exp) zuweist.

  Beispiel: gegeben: $2.0 * x^2 - 1$
  dann: `\x -> (2.0 * x^2 - 1.0)`
  Das ist die Funktion mit Werten:

x	...	-1	0	1	...
Wert	...	1	-1	1	...

  also graphisch:



  Natürlich kann man einen Bezeichner einführen wie z.B.

```
f = \x -> 2.0 * x^2 - 1.0
```

  aber es ist oft bequem, die unbenannte Funktion als Argument in „higher order functions" zu benutzen:

```
> diff (\x -> 2.0 * x^2 - 1.0) 0
0.00202656
> diff (\x -> 2.0 * x^2 - 1.0) 1
4.00209
> diff (\x -> 2.0 * x^2 - 1.0) (-1)
-3.99792
```

  usw.

- Komposition von Funktionen
  Mathematik: gof heißt: erst f, dann darauf g anwenden,

  also sowas wie „2 Verarbeitungsschritte" hintereinander:

  Ergebnis $\leftarrow \boxed{\text{g}} \leftarrow \boxed{\text{f}} \leftarrow$ Input

  Die Haskell-Schreibweise für dieselbe Sache ist `g.f`

  ***Wertebereich* von f muß zu *Argumentbereich* von g passen!** !

  Beispiel: 2. Ableitung:

```
> (diff.diff) cos 0
-1.07288
```

  denn: (-cos) $\leftarrow \boxed{\text{diff}} \leftarrow$ (-sin) $\leftarrow \boxed{\text{diff}} \leftarrow$ cos
  und - cos 0 = -1

`diff.diff cos 0` **wird aufgefaßt als**
`diff.((diff cos) 0)` **wegen höchster Priorität der Funktionsapplikation!** !

Funktions*applikation* (bei higher order functions) und Funktions*komposition nicht verwechseln!* !

```
(diff.diff) cos 0
     |     |
     |     |--- Applikation: diff hat cos als Argument transformiert cos
     |
     |--- Komposition: zweites diff wird nach dem ersten diff, d.h.
          auf Ergebnis vom ersten diff angewendet.
```

Natürlich kann man dem Kind einen Namen geben, z. B.:

```
> let diff2 = diff.diff in diff2 cos 0
-1.07288
```

Beispiel: Komposition von *Funktion* und *inverser* Funktion gibt Identität, z. B.:

```
sqrt . (\x -> x^2) = id  in R
 |      |
 |      |--- Quadratfunktion
 |
 |--- vordefinierte Wurzelfunktion
```

```
> (sqrt.(\x -> x^2)) 5
5.0
> (sqrt.(\x -> x^2)) (-5)  -- stimmt nur für x>0
5.0
> diff (sqrt.(\x -> x^2)) (-5)
-0.999927
```

2.2.9 Operatoren

= Funktionen mit *2 Argumenten* in *Infix*-Schreibweise
Operator-Bezeichner werden durch Zusammenstellung von 1 oder mehr Sonderzeichen gebildet, klar: einige Sonderzeichen (-kombinationen) sind reserviert oder schon belegt z. B.: "`::`", "`=`", "`->`" usw.
Man kann zwischen Operator-/Funktions-Schreibweise wechseln:

- Operator in Klammern

  = Funktionsbezeichner für Präfix-Schreibweise, z. B.:

```
> (+) 3 4
7
```

- Funktionsbezeichner in backquotes "`"

  = Operatorbezeichner für Infixschreibweise, z. B.:

```
> 3 `div` 4
0
> 3 `max` 4
4
```

Klammerungen: es gibt 9 *Präzedenzstufen*, z. B.:

"."	Funktionskomposition	9
"+"	Addition	6
"&&"	logisches und	3
"||"	logisches oder	2

⇒ Literatur

innerhalb einer Stufe gibt es *Assoziativitätsregeln*:

- Assoziation nach *links*:

  z. B.: 8-5-1=(8-5)-1=2 und nicht 8-(5-1)=4

- Assoziation nach *rechts*:

  z. B.: 2^2^3 = 2^(2^3) = 256 und nicht (2^2)^3 = 64

- Operation *ist assoziativ*, d. h. beliebig zu klammern:

  z. B.: 3+2+1 = (3+2)+1 = 3+(2+1)

  (Linksklammerung ist hier üblich)

⇒ Literatur

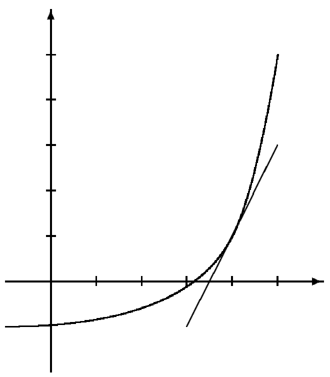
Benutzer kann selber Operatoren definieren und ihre Priorität und Assoziationsart in einer besonderen Deklaration festlegen, mehr dazu beim Thema „Module". !

2.2.10 Beispiel: Newton-Approximation

Das Verfahren zur Ermittlung einer Nullstelle einer Funktion f: R→R ist uns aus dem 1. Semester bekannt:

x_0 `:= geeigneter Startwert`

x_{i+1} `:=` x_i `- f(`x_i`)/f´(`x_i`)`



es gilt: $f'(x) \approx \frac{0-f(x_i)}{x_{i+1}-x_i}$

Das Verfahren ist wie geschaffen für *imperative* Programmierung:

- *Iteration* = Serie von Schritten, bis akzeptable Näherung gefunden
- *Programmvariable* = nehmen sich ändernde Werte von x auf

also interessant, ob man sowas — und zwar gut! — *funktional* erledigen kann!

Wir erinnern uns:

- Iteration läßt sich durch Rekursion ersetzen
- wobei Zwischenergebnisse als Resultate der Funktionsergebnisse „unterwegs" anfallen.

Daher die offensichtliche Lösung:

```
newton :: (Float->Float)->Float->Float
newton f x
   |abs(f x) < 0.001 = x  -- Feierabend wenn gut genug
   |otherwise        = newton f (x - f x / diff f x) -- weiter
```

Anwendung z. B. so:

```
Main> newton (\x->x^2-2) 10  --loest x^2-2 = 0, d.h. Wurzel 2, Start bei 10
1.41454
```

Hat man erst eine Programmieraufgabe gelöst, so sollte man gleich fragen, ob sich die Lösung *verallgemeinern* läßt, d. h. gibt es hier eine *Struktur*, ein *Muster*, das *mehrfach* angewendet werden kann?

Bestandteile der Lösung sind:

- ein Ausgangswert: `x`
- eine den aktuellen Wert verbessernde Berechnung: `(x - f x / diff f x)`
- ein Wiederholungsmechanismus: rekursiver Aufruf
- ein Abbruchkriterium

Idee:
Mache aus dem Wiederholungsmechanismus eine Funktion, die die drei anderen Bestandteile als Parameter hat!
Daher definiere:

```
until :: (a->Bool)->(a->a)->a->a
           |          |     |  |--- Ergebnis
           |          |     |--- Startwert
           |          |--- Verbesserung
           |--- Abbruchkriterium
```

Und natürlich a = *Typvariable*, d. h. egal, in welchem Wertebereich wir arbeiten, also *polymorphe* Funktion.

```
repeat_until::(a->Bool)->(a->a)->a->a
repeat_until crit change x
   |crit x    = x                                  --zufrieden
   |otherwise = repeat_until crit change (change x) --weiter
```

Einfaches Anwendungsbeispiel:

```
Main> repeat_until even pred 10
10
Main> repeat_until even pred 11
10
```

Wobei die Standardfunktionen

```
even :: Int -> Bool     -- "ist gerade"
pred :: Int -> Int      -- "Vorgänger"
```

vorhanden sind.

Jetzt damit zurück zu Newton:

```
newton :: (Float->Float)->Float->Float
newton f x = repeat_until n_crit n_change x
   where n_crit x = abs(f x) < 0.001
         n_change x = x - f x / diff f x
```

Anwendung für Quadratwurzelberechnung:

```
Main> newton (\x->x^2-2) 10
1.41454
Main> newton (\x->x^2-25) 10
5.0
Main> newton (\x->x^2-80) 10
8.94427
```

usw.

- Man kann die Suche noch flexibler machen, insbesondere *bessere Abbruchkriterien* einführen, die auf *Vergleich* von 2 (oder mehr) der *aufeinanderfolgenden Näherungswerte* beruhen;

  ⇒ Kapitel 3: Listen

- Man kann Nullstellenberechnung auch benutzen, um die *inverse Funktion* zu einer Funktion f numerisch zu ermitteln:

  $$f^{-1}(y) = x \qquad \text{gdw.} \qquad f(x) - y = 0$$

  Das leistet die folgende Definition:

  ```
  inverse:: (Float->Float)->Float->Float->Float
  inverse f y x = newton g x
     where g x = f x - y
  ```

  Einfache Anwendungen sind wieder Wurzelprobleme:

  ```
  Main> inverse (\x->x) 2 10
  2.0
  Main> inverse (\x->x^2) 2 10
  1.41454
  Main> inverse (\x->x^3) 2 10
  1.25993
  Main> inverse (\x->x^4) 2 10
  1.18925
  Main> inverse (\x->x^10) 2 10
  1.07179
  ```

  usw.

Beachte:

Anwendung der Newton-Iteration hat ihre mathematischen Grenzen, *und* man muß sich für wirklich gute Ergebnisse auch noch in der numerischen Mathematik auskennen und Sorgfalt hinsichtlich Startwerten, Abbruchkriterien, genaue Formulierungen des Iterationsschritts ... walten lassen!

2.2.11 Auswertung von Ausdrücken

= Reduktion auf die einfachste äquivalente Form als Antwort
dazu: verwende „eingebaute“ Semantik für primitive Operationen *und* die vom Programmierer gelieferten Definitionen, nämlich: ersetze linke Seite einer definierenden Gleichung durch rechte Seite.

Beispiel:

gegeben:

```
pi = 3.14
area x = pi * x * x
```

dann:

```
   area (9+1)
=> area 10        Semantik "+"
=> pi * 10 * 10   Definition "area"
=> 3.14 * 10 * 10 Definition "pi"
=> 31.4 * 10      Semantik "*"
=> 314            Semantik "*"
```

(dabei angenommen, was stimmt, daß Folgen von “*” unter Assoziation nach *links* ausgewertet werden)

Das ist *nicht die einzig* mögliche Reduktionsfolge! !

Alternative:

```
   area (9+1)
=> pi   * (9+1) * (9+1)  Definition "area"
=> 3.14 * (9+1) * (9+1)  Definition "pi"
=> 3.14 * 10 * (9+1)     "+"
=> 31.4 * (9+1)          "*"
=> 31.4 * 10             "+"
=> 314                   "*"
```

Im ersten Beispiel wird zuerst der *innere* Teilausdruck “(9+1)” ausgewertet, während im zweiten Beispiel die Auswertung *von außen* nach *innen* geschieht; man spricht (bei konsequenter Anwendung der Vorgehensweisen) von „innermost“ und „outermost“ Strategie.

***Haskell* verwendet *„outermost“* mit dem wichtigen Zusatz, daß *Mehrfachvorkommen desselben* Teilausdrucks nur *einmal* ausgewertet werden.** !

also im Beispiel:

```
   area (9+1)
=> pi * (9+1) * ( )
              >------^
=> 3.14 * (9+1) * ( )
                >------^
=> 3.14 * 10 * 10
=> ...
```

Outermost-Reduktion mit dieser Optimierung heißt auch *„lazy evaluation“*: Ein Teilausdruck wird erst ausgewertet, wenn es wirklich nicht mehr anders geht, und dann auch nur einmal.
⇒ möglich, daß Teilausdruck gar nicht ausgewertet,
⇒ möglich, daß das Scheitern einer Berechnung vermieden wird!

Beispiel:

definiere: `first_of_two x y = x`
dann: `first_of_two 1 (1 ‘div‘ 0)`
`=> 1 -- outermost!`

Bei innermost gäbe es hier einen Laufzeitfehler!
ähnlich: nicht-terminierende Teil-Berechnung.

Welche Strategie ist besser?

Keine eindeutige Antwort; verschiedene Programmiersprachen verhalten sich verschieden:

innermost (auch: „eager evaluation“):

Lisp-Dialekte, ML (funktionale Sprachen) imperative Sprachen!

mit kleinen Ausnahmen, z. B.:

```
if ... then ... else ...
... and then ...
... or else ...
```

Theorie sagt:

Wenn innermost ein Ergebnis liefert, dann liefert outermost *dasselbe* Ergebnis – aber es kann sein (siehe oben), daß *nur* outermost ein Ergebnis liefert!

- outermost Vorteile:

  Erlaubt gewisse Programmstrukturen, die mit unendlichen Datenstrukturen ($\approx$ nicht terminierende Berechnungen) arbeiten.

- outermost Nachteile:

  Implementierung schwieriger und aufwendiger

noch ein damit zusammenhängender Begriff:

weil Auswertung eines Ausdrucks scheitern kann, ist es bequem (für theoretische Überlegungen), alle Wertebereiche durch einen besonderen Wert zu ergänzen:

$\perp$ = „undefiniert“ = „bottom“
steht für Laufzeitfehler oder Nichtterminierung

dann:

Funktion f heißt *strikt* gdw. $\texttt{f}\ \perp = \perp$.

arithmetische Operationen sind immer strikt — klar, 3 mal undefiniert is undefiniert —, aber bei lazy evaluation gibt es auch nicht strikte Funktionen:

Beispiel: konstante Funktionen wie z. B.:

```
three x = 3
```

dann:

```
three (1/0)
=> 3         -- outermost!


all_ints_from :: Int -> [Int]
all_ints_from x = x : all_ints_from(x+1)

first_even :: [Int] -> Int
first_even x:xs
        | even x    = x
        | otherwise = first_even xs
```

2.2.12 Input / Output

lassen sich nur schwer dem Grundprinzip „Funktionsauswertung“ unterordnen.

Beispiel:

Interaktion mit Benutzer durch Ausgabe von Ergebnissen und Aufforderung zu weiterer Eingabe.

Eine saubere Lösung benötigt erhebliche theoretische/konzeptuelle Vorbereitung; hier Beschränkung auf Rezept für Teilproblem:

Wie kriegt man Ergebnis einer Berechnung in sinnvoll formatierter Weise ausgedruckt?

1. die üblichen Datentypen gehören zu einer Typklasse `Show` und bieten eine Funktion `show::(Show a) => a -> String`, die Wert (Element des Datentyps) in einen String umwandelt;

   Diese Funktion wird von Hugs zur Ausgabe von Ergebnissen benutzt. !

   Es gibt Typen, die nicht zur Klasse `Show` gehören, insbesondere Funktionstypen. !

   Führt man selber neue Datentypen ein (⇒ später), so sollte man sich um eine `show`-Funktion kümmern. !

2. Hugs benutzt `show`-Funktionen, gibt aber dann Antworten üblicherweise nicht in Form von Haskell-String-Bezeichnern, sondern eben die Zeichenreihe selbst:

   ```
   > fac 3
   6
   ```

   Ausnahme:

   Werte vom Typ String werden so ausgegeben, wie sie auch in der Programmiersprache auftreten, nämlich in Anführungszeichen:

   ```
   > "ABC"
   "ABC"
   ```

   Differenz zu älteren Hugs-Versionen oder Gofer (⇒ Vorsicht bei gebrauch von Literatur) !

3. zum Zeichenvorrat Char gehören neben druckbaren Zeichen auch *Steuerzeichen*, insbesondere *newline* Symbol: `'\n'`

   (Bemerkung: backslash \ dient als Fluchtsymbol, andere Verwendung:

`'\\'`	$\widehat{=}$	backslash
`'\´'`	$\widehat{=}$	single quote
`'\"´`	$\widehat{=}$	double quote)

   Mann kann dan Text *formatieren* wie z. B.:

   ```
   "dies ist \neine neue Zeile "
   ```

   was man mit Hilfe des *Konkatenationsoperators* `++` vielleicht sinnfälliger schreiben kann als

   ```
   "dies ist \n" ++ "eine neue Zeile"
   ```

   Aber: wegen vorherigem Punkt hilft das noch nicht viel:

   Hugs druckt gnadenlos den String:

```
> "dies ist \n" ++ "eine neue Zeile"
"dies ist \neine neue Zeile"
```

d. h. interpretiert \n *nicht*! !

4. Vernünftig wird das Resultat durch Vorschalten der Funktion

```
putStr :: String -> IO()
```

nämlich:

```
> putStr("dies ist \n" ++ "eine neue Zeile")
dies ist
eine neue Zeile
```

Sorry: Genaue Diskussion des „Wertebereichs“ dieser „Funktion“ ist hier nicht möglich. !

- approximative Vorstellung:
  putStr liefert als Ergebnis (Wert) eine *Aktion*, d. h. `IO()` und verwandte Typen beinhalten Aktionen, und Wiedergabe von sowas durch Hugs ist gleich Ausführung der Aktion.
- Anwendung von putStr und ähnlichen Funktionen nur auf dem *top level* von Ausdrücken möglich, also *nicht* innerhalb von normalen Ausdrücken mit normalen Werten!
  (d. h. man kann anders als in Ada nicht *während* der Auswertung einer Funktion wie `fac :: Int->Int` mal ´was ausdrucken!)

Beispiel:

Wir machen die Fakultätsfunktion „fehlertolerant“ dadurch, daß sie für negative Argumente eine entsprechende Meldung erzeugt.

Dazu: ändere Fakultätsfunktion so, daß sie Zeichenreihe als Ergebnis hat:

```
fac:: Int->Int
fac 0 = 1
fac n = n*fac (n-1)

fault_tolerant_fac::Int->String
fault_tolerant_fac n
   |n >= 0     = show(fac n)
   |otherwise  = ">>>: fac nicht definiert \n"++
                 "     fuer "++ (show n)


Main> fault_tolerant_fac 3
"6"
Main> fault_tolerant_fac (-3)
">>>: fac nicht definiert \n     fuer -3"
Main> putStr (fault_tolerant_fac (-3))
>>>: fac nicht definiert
     fuer -3
```

Für dieses Beispiel gibt es eine bessere Lösung: Haskell hat Vorkehrungen für Ausnahmebehandlung, z. B.:

```
error :: String->a
```

Dies ist auch wieder eine Funktion, die auf eine Aktion hinausläuft: *Abbruch* der Auswertung, aber: vorher wird die als Argument gegebene Zeichenreihe ausgedruckt.

(error hat den polymorphen Ergebnistyp a, damit man die Funktion in jeden Zusammenhang einfügen kann) !

```
fault_tolerant_fac::Int->Int
fault_tolerant_fac n
   |n >= 0     = fac n
   |otherwise  = error (" fac nicht definiert fuer " ++ (show n))


Main> fault_tolerant_fac (-3)

Program error:  fac nicht definiert fuer -3
```

Wirkung von error:

1. Abbruch (= Program error)
2. Ausgabe des vom Programmierer vorgesehenen String

Kapitel 3

Listen

3.1 Elementares

- Listen sind *linear* angeordnete *Kollektionen* von Werten *desselben* Typs.

  Unterschied zum Mengenbegriff:

  ```
  [1] /= [1,1],  [1,2] /= [2,1]
  ```

  !

- Listen können über *jeden* Typ gebildet werden:

  ```
  [] :: [a]   -- leere Liste, gehört zu jedem Listentyp (polymorph)

  [1,2,3] :: Num a => [a]             -- Liste von Zahlen

  [[1], [2,3]] :: Num a => [[a]]      -- Liste von Listen von Zahlen

  [[], [2,3]] :: Num a => [[a]]       -- [] als leere Zahlenliste

  [sin, cos] :: Floating a => [a->a]  -- Liste von reelen Funktionen

  [coarse_diff, fine_diff] :: [(Float->Float)->Float->Float]
                                      -- Liste von higher order functions
  ```

  ```
  1 /= [1]
  ```

  Element ungleich einelementige Liste (singleton list)!

  !

- Listen sind wichtige Datenstruktur in gesamter Informatik, aber in funktionaler Programmierung noch größere Bedeutung als in imperativer Programmierung;

  insbesondere: lazy evaluation erlaubt Arbeit mit *unendlichen* Listen!

- Listen sind gut verstandener mathematischer Gegenstand mit vielen nützlichen Gesetzen für einschlägige Operationen und Funktionen.
  zB:

  > R. Bird:
  > „An introduction to the theory of lists“,
  > Oxford Programming Research Group report PRG-56, 1986

- Listen können *aufgebaut* und in *eindeutiger* Weise *zerlegt* werden durch:

```
[] :: [a]  -- leere Liste, Basis aller Konstruktionen

(:) :: a->[a]->[a]  -- Anfügen eines Elements am Kopf der Liste
                    -- ("cons" = "construct"-Operation)
```

zB:

```
3:[]
2:(3:[])
1:(2:(3:[]))
```

(:) assoziiert *rechts*, also:

```
1:2:3:[]  ==  1:(2:(3:[]))
```

!

Konstruktion / Zerlegung mit (:) ist Grundlage aller Definitionen einschlägiger Operationen / Funktionen. !

- Es gibt mehrere *notationelle Kurzformen:*
  - **Aufzählung:**

    ```
    [1,2,3] == 1:2:3:[]
    ```

    Sonderfall für Strings:

    ```
    "abc" == ['a', 'b', 'c']
          == 'a':'b':'c':[]
    ```

  - **Intervalle:**

    ```
    [2..5]   == [2,3,4,5]  -- Schritte von +1
    [5..2]   == []         -- Schritte von +1
    [5,4..2] == [5,4,3,2]  -- Schritte von -1
    ```

    ähnlich:

    ```
    ['a','c'..'f'] == "ace"
    ```

    Beachte:

    ```
    [0.0,0.2..1.4] == [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2]
           -- Feierabend, wenn Obergrenze nicht exakt getroffen wird
           -- (Float-Arithmetik!)

    [0.2,0.2..1.4] == [0.2,0.2,0.2,0.2......
           -- Schrittweite von 0!
    ```

    !

  - **Listen-Komprehension:** (analog: Mengenkomprehension)
    Sei `l = [1,2,3]`
    dann `[ 2*x | x<-l ] == [2,4,6]`
    ⇒ genauere Betrachtung später...

- viele *Standardfunktionen / -operationen*

  Definition abgestützt auf [] und (:) und pattern matching.

  zB kennen wir schon:

```
length :: [a] -> Int    -- Länge der Liste
head   :: [a] -> a      -- Kopfelement
tail   :: [a] -> [a]    -- Rest
                        -- head und tail sind partielle Funktionen,
                        -- Fehler falls Liste leer!
```

  andere Beispiele:

  - **Konkatenation** (join, append, Vereinigung)

    zB:

    ```
    [1,2] ++ [1,3] == [1,2,1,3]
    [] ++ [1,2] == [1,2]
    [1,2] ++ [] == [1,2]
    ```

    also:

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

    Beispiel: Listenumkehr:

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

    (++) ist *assoziative* Operation mit *neutralem* Element [], also:

    ```
    ([1]++[2])++[3] == [1,2,3] == [1]++([2]++[3])
    ```

    In Haskell konventionell Assoziirung nach rechts:

    ```
    [1]++[2]++[3] == [1]++([2]++[3])
    ```

    (:) und (++) nicht verwechseln; Typfehler bei:

    ```
    [1]:[2,3]
    1++[2,3]
    ```

    !

  - **Konkation, Form 2** („flatten“)

    wirkt auf *Listen* von *Listen*:

    ```
    concat [[1,2,3],[4,5],[],[6]] == [1,2,3,4,5,6]
    ```

    also:

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

    Beachte Typen in dieser Definition:

```
[] :: [[a]]
[] :: [a]
x  :: [a]
xs :: [[a]]
```

!

– **Teil-Listen:**

```
take, drop :: Int->[a]->[a]
take 0 _  = []
take _ [] = []
take (n+1) (x:xs) = x:take n xs

drop 0 xs = xs
drop _ [] = []
drop (n+1) (x:xs) = drop n xs


-- Anwendung zum Beispiel:
> take 2 "abc"
"ab"
> take 5 "abc"
"abc"

> drop 2 "abc"
"c"
> drop 5 "abc"
""
```

– **Enthalten sein:**

```
elem :: a->[a]->Bool
elem _ [] = False
elem e (x:xs) = x==e || elem e xs

-- Anwendung zum Beispiel:
> elem 'a' "abc"
True
> elem 'x' "abc''
False
```

⇒ *Standard Prelude* A1, Report Seite 96 ff. enthält viele weitere Definitionen, einiges daraus später!

- Dieser Teil ist auch ohne „höhere“ Haskell-Kenntnisse fast 100% verständlich ⇒ lesen!
- allgemeiner Aufbau als Haskell-Modul:

```
module PreludeList      -- Modulkopf
   (head, tail, ...)    -- Verzeichnis der nach aussen angebotenen Dienste
                        -- (Exportliste vgl. Spezifikationsteil von Ada Paketen)
  where                 -- hier geht's los

import ...              -- Verzeichnis der von anderswo importierten Dienste
                        -- (vgl. Ada "with clause")

infix ...               -- "fixity declarations" d.h. Deklaration von
                        -- Präzedenz und Assoz. der im Modul definierten
                        -- Operatoren

...                     -- jetzt die Definitionen ("Implementierung")
```

Beispiel PreludeList aus dem Standard-Prelude:

```
--  A.1  Prelude PreludeList

-- Standard list functions
module PreludeList (
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3)
  where

import qualified Char(isSpace)

infixl 9  !!
infix  4  `elem`, `notElem`

-- head and tail extract the first element and remaining elements,
-- respectively, of a list, which must be non-empty.  last and init
-- are the dual functions working from the end of a finite list,
-- rather than the beginning.
head             :: [a] -> a
head (x:_)       =  x
head []          =  error "PreludeList.head: empty list"

last             :: [a] -> a
last [x]         =  x
last (_:xs)      =  last xs
last []          =  error "PreludeList.last: empty list"

tail             :: [a] -> [a]
tail (_:xs)      =  xs
tail []          =  error "PreludeList.tail: empty list"

init             :: [a] -> [a]
init [x]         =  []
init (x:xs)      =  x : init xs
init []          =  error "PreludeList.init: empty list"

null             :: [a] -> Bool
null []          =  True
null (_:_)       =  False

-- length returns the length of a finite list as an Int.
length           :: [a] -> Int
length []        =  0
length (_:l)     =  1 + length l
...
```

3.2 Beispiel: Sortieren durch Einfügen

das Verfahren ist uns wohlbekannt:

> sortiere Liste durch wiederholtes ordnungsgerechtes Einfügen.

hier die Haskell Definitionen:

```
into :: Ord a => a->[a]->[a]     -- Sortieren braucht Ordnungsrelation auf
                                 -- Elementtyp!
into x [] = [x]
into x (y:ys)
       |x <= y    = x:y:ys       -- aufsteigend!
       |otherwise = y:into x ys

iSort :: Ord a => [a]->[a]
iSort [] = []
iSort (y:ys) = into y (iSort ys)

-- Anwendung zum Beispiel:
> into 3 [1,5,7,12]
[1,3,5,7,12]
> into 'a' "Computer"
"Caomputer"
> into "xx" ["aa", "zz"]
["aa","xx","zz"]           -- Strings und allgemein Listen von Elementen mit
                           -- Ordnungsrelation, sind in Haskell
                           -- lexikographisch geordnet!
> iSort [2,-9,67,-3,0,11]
[-9,-3,0,2,11,67]
> iSort [cos 0, cos 1, cos 2, cos 3]
[-0.989992, -0.416147, 0.540302, 1.0]
```

Man beachte die Segnungen von overloading und polymorphism! !

Kontrollfrage: was gibt

```
> iSort [cos,sin]  ?
> iSort []         ?
```

!

3.3 Beispiel: „Datenbank“

Zur Verwaltung der Hardware des Institus wird eine Liste geführt, die aus Paaren des Formats (<Rechnertyp>, <Raum>) besteht.

Lösung hier ist sehr primitiv, soll nur Arbeit mit Listen illustrieren.

(natürlich kennt Haskell auch Aufzählungstypen, natürlich kann man auch einen Ordnungsbegriff für die Listenelemente einführen usw. usf. ...)

```
-------------------------------------Liste von Rechnern mit Aufstellungsort

type Room = String                           --Typ-Synonyme fuer Lesbarkeit
type Hardware = String
type HwList =[(Hardware,Room)]

example_list =                                                    --Beispiel
               [("PC","T01"),
                ("PC","Lab1"),("PC","Lab1"),("PC","Lab1"),
                ("PC","Lab2"),("Sun","Lab2"),("Sun","Lab2")]

                                                              --Aufbau/Abbau
insert, delete:: Hardware->Room->HwList->HwList
```

```
insert h r hwl = (h,r):hwl
delete h r [] = []              --keine Wirkung wenn (h,r) nicht vorhanden
delete h r ((h1,r1) : rest)
       |h1 == h && r1 == r = rest
       |otherwise          = (h1,r1) : delete h r rest

                                --Funktionen fuer Anfragen, Aenderungen etc
locations:: Hardware->HwList->[Room]  --Ergebnis enthaelt Mehrfachvorkommen
locations h [] = []
locations h ((h1,r1) : rest)
       |h1 == h    = r1 : (locations h rest)
       |otherwise  = locations h rest

how_many:: Room->HwList->Int
how_many r [] = 0
how_many r ((h1,r1) : rest)
       |r1 == r   = 1 + how_many r rest
       |otherwise = how_many r rest

replace:: Hardware->Hardware->HwList->HwList
replace h h' [] = []
replace h h' ((h1,r1) : rest)  = entry : replace h h' rest
                          where entry |h1 == h   = (h',r1)
                                      |otherwise = (h1,r1)


-- Beispielanwendungen der definierten Funktionen
> insert "Sun" "T01" []
[("Sun", "T01")]
> locations "Sun" example_list
["Lab2","Lab2"]
> locations "Sun" (replace "Sun" "Moon" example_list)
[]
> how_many "Lab2" (delete "PC" "Lab2" example_list)
2
> how_many "PC" (delete "PC" "Lab2" example_list)
0       -- Quatsch, das kommt davon, wenn man mit Typsynonymen arbeitet!!
```

3.4 Higher order functions (HOFs) für Listen

Ein gut sortierter Werkzeugkasten voll HOFs erlaubt, typische Aufgaben *ohne* explizite Programmierung des Listendurchlaufs (Rekursion, pattern matching) zu lösen. !

Natürlich ist alles polymorph um universelle Anwendung zu ermöglichen! !

- *map* wendet eine Funktion an auf jedes Listenelement:

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

> map even [1..10]
[False,True,False,True,False,True,False,True,False,True]
```

- *fold*-Funktionen fassen Listenelemente zusammen, indem sie eine geeignete Operation „dazwischen setzen“; es gibt mehrere Arten:

```
foldr:: (a->b->b)->b->[a]->b
foldr op start []      = start
foldr op start (x:xs)  = x ‘op‘ (foldr op start xs)

foldl:: (a->b->a)->a->[b]->a
foldl op start []      = start
foldl op start (x:xs)  = foldl op (start ‘op‘ x) xs
```

also:

foldr op start [$x_1 \ldots x_n$] = x_1 ‘op‘ (x_2 ‘op‘ (...(x_n ‘op‘ start)...)

foldl op start [$x_1 \ldots x_n$] = (...(start ‘op‘ x_1) ‘op‘ x_2)...) ‘op‘ x_n

```
> foldr (+) 0 [1..10]
55
> foldl (+) 0 [1..10]
55
                        -- jetzt Unterschied!

> foldr (-) 0 [1..10]
-5                      -- 1-(2-(...10-0)...)
> foldl (-) 0 [1..10]
-55                     -- (...(0-1)-2)...)-10
```

Will man *nur* die Elemente der Liste zusammenfassen, so benötigt man für foldr und foldl ein *neutrales* Element der anzuwendenden Operation als Startwert (Basiswert):

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

es geht auch ohne:

```
foldr1 :: (a->a->a)->[a]->a
foldr1 op [x]    = x
foldr1 op (x:xs) = x ‘op (foldr1 op xs)
```

also: foldr op start [$x_1 \ldots x_n$] = x_1 ‘op‘ (x_2 ‘op‘ (...(x_{n-1} ‘op‘ x_n)...)

Liste darf nicht leer sein! !

analog: foldl1

```
> foldl1 max [1..10]
10      -- 0 nicht neutrales Element für max, da Liste auch negative
        -- Werte enthalten könnte.
```

- filter, takeWhile, dropWhile

  liefern einen *Teil* der Liste

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

```
> filter even [1..11]
[2,4,6,8,10]

takeWhile:: (a->Bool)->[a]->[a]
takeWhile p []   = []
takeWhile p (x:xs)
   |p x          = x:takeWhile p xs
   |otherwise    = []         -- also Anfangsstück, solange p gilt

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

- mehr in Prelude

  (einige Definitionen dort aus Gründen der Systematik anders als bei uns, aber äquivalent)
- mit solchen HOFs leicht, komplexere Listenverarbeitungsbeispiele ökonomisch zu formulieren:

  z.B.:

```
> ((foldl (+) 0).(map (^2)).(filter (>0)) [-10..10]
385
```

  also Stufenverarbeitung:

```
[-10..10]
     >-------|
             v
          filter (>0)
             |
             |           [1..10]
             v
          map (^2)
             |
             |           [1,4,9,16,25,36,49,64,81,100]
             v
          foldl (+) 0
             |
             >--> 385
```

3.5 Wiederaufnahme der Beispiele 3.2 und 3.3

Jetzt dasselbe mit HOFs!

- insertion sort:

```
iSort2:: Ord a => [a]->[a]
iSort2 = foldr into []
```

  also: `iSort2` $[x_1 \ldots x_n]$ = x_1 `'into'` (x_2 `'into'` ($\ldots x_n$ `'into'` `[]`)...)
- Datenbank
  - *replace* ist Kandidat für *map*, weil Elemente der Liste verändert werden;
  - *locations* ist Kandidat für:

  * erst *filter* zum Herauslösen der Einträge mit gesuchter Hardware
  * dann *map* für Projektion der verbleibenden Elemente auf Raum-Komponente
- *how_many*: ähnlich

```
--------------Alternativformulierung mit higher order functions
------------------------------------------------------------------

is_wanted:: Hardware->(Hardware,Room)->Bool
is_wanted h (h',r) = h == h'

locations2:: Hardware->HwList->[Room]
locations2 h = map snd . filter (is_wanted h)

------------------------------------------------------------------
                                  --dasselbe mit operator section

locations3:: Hardware->HwList->[Room]
locations3 h = map snd . filter ((== h).fst)

------------------------------------------------------------------
------------------------------------------------------------------

exchange:: Hardware->Hardware->(Hardware,Room)->(Hardware,Room)
exchange h h' (h'',r)
   |h == h''  = (h',r)
   |otherwise = (h'',r)

replace2:: Hardware->Hardware->HwList->HwList
replac2 h h' = map (exchange h h')
------------------------------------------------------------------

-- how_many: Hausaufgabe!
```

3.6 Beispiel: Text-Verarbeitung

Texte sind zunächst einach *Zeichnreihen*:

```
type Text = String
```

die aber zur Verwaltung / Verarbeitung verschiedenen *Sichtweisen* unterworfen werden, z. B.:

```
type Line = String    -- Zeile, ohne '\n'
type Word = String    -- Wort, ohne ' ', '\n'
type Para = [Line]    -- Absatz = Zeilenfolge
```

Wir betrachten die Umwandlung:

```
asLines:: Text->[Line]
```

zu deren Definition einige Entscheidungen nötig sind (natürlich sind solche Spezifikationen nicht zwingend):

1. Text ohne `'\n'` gibt Zeile:

   ```
   asLines "Das ist ein Text" = ["Das ist ein Text"]
   ```

   dann aber auch:

```
asLines "" = [""] = [[]]
```

d. h. leerer Text gibt eine Leerzeile.

2. `'\n'` ist *Trennzeichen* (nicht: Anfangs-/Endezeichen):

```
asLines "Das ist \n ein Text" = ["Das ist", "ein Text"]

asLines "Das ist ein Text\n" = ["Das ist ein Text", []]
```

d. h. #Zeilen = #newline + 1

3. Es gibt *keinen* Umbruch gemäß einer maximalen Zeilenlänge und *keine* Normalisierung durch Weglassen oder Einfügen von Leerzeichen:

```
asLines "Das ist    ein Text    " = ["Das ist    ein Text    "]
```

Lösungsansatz:
wir betrachten versuchsweise die Definition

```
asLines = foldr op start
```

und überlegen uns geeignete Definitionen für `start` und `op`.
man erinnere sich:

α `foldr ::  (a->b->b) -> b -> [a] -> b`

β `foldr op start [] = start`

γ `foldr op start (x:xs) = x 'op' (foldr op start xs)`

wir schließen:

- `asLines ::  Text->[Line]`, also `foldr start op ::  Text->[Line]`
  also wegen α:

  ```
  b = [Line]
  a = Char
  ```

  also:

  ```
  op    :: Char -> [Line] -> [Line]
  start :: [Line]
  ```

- `asLines  = [[]]` (1.)
  andererseits wegen β:

  ```
  asLines "" = foldr op start [] = start
  ```

  d. h.:

  ```
  start = [[]]   -- Liste, die nur 1 Leerzeile enthält
  ```

- `asLines ¨ein \nText'' = [¨ein", ¨Text"]` (1., 2.)
  andererseits wegen γ:

```
asLines "ein \nText"
  = 'e' `op` (asLines "in \nText")
  = 'e' `op` ["in", "Text"]
```

(wegen 1., 2.)

also:

```
op c ls = (c:head ls) : (tail ls)
                      falls c/='\n'
```

- `asLines "\n ein \nText" = [[], "ein", "Text"]` (1., 2.)

  andererseits wegen γ:

```
asLines "\n ein \nText"
  = '\n' `op` (asLines "ein \nText")
  = '\n' `op` ["ein", "Text"]
```

(wegen 1., 2.)

also:

```
op c ls = [] : ls          falls c=='\n'
```

also insgesamt:

```
asLines:: Text->[Line]
asLines  = foldr break_on_newline [[]]
  where break_on_newline:: Char->[Line]->[Line]
        break_on_newline c ls
          |c == '\n' = []:ls                    --new group
          |otherwise = (c:(head ls)) : (tail ls) --add to group
```

Die in dieser Definition verwendete Funktion "break_on_newline" ist es wert, *verallgemeinert* zu werden:

offensichtlich analoge Aufgabe, wenn wir einen *Text* nach *Wörtern* gruppieren: dann definiert `' '` die Bruchstelle,

wenn wir eine Folge von Zeilen nach Absätzen gruppieren: dann definiert die Leerzeile `[]` die Bruchstelle.

$\Rightarrow$ aus der Bruchstellenmarkierung einen *Parameter* machen *und* die Funktion gleich für *beliebige* Listen formulieren.

```
------------------------------allgemeines break_on fuer Listen
                         --behandelt break-elem als Separator

break_on:: Eq a => a->a->[[a]]->[[a]]
break_on b c ls
   |c == b    = []:ls                    --new group
   |otherwise = (c:(head ls)) : (tail ls) --add to group

-----------------------------------------------text as lines

asLines:: Text->[Line]
asLines  = foldr (break_on '\n') [[]]
```

Anwendung z. B. so:

```
Main> break_on '\n' 'd' [‘‘ef’’]
[‘‘def’’]
Main> break_on '\n' '\n' [‘‘def’’]
[", ‘‘def’’]
Main> break_on '\n' '\n' [[]]
[", "]
Main> asLines "
["]
Main> asLines ‘‘abc def’’
[‘‘abc def’’]
Main> asLines ‘‘\nabc \n\ndef\n’’
[", ‘‘abc ‘‘, ", ‘‘def’’, "]
```

jetzt machen wir in *derselben* Weise:

- aus einer Zeile eine Liste von *Wörtern*:

  Trennzeichen = `' '`

- aus einer Liste von Zeilen eine Liste von *Paragraphen* Absätzen:

  Trennzeichen = `[]` = leere Zeile

```
-----------------------------------------------lines as words
                                              --no empty words

asWords:: Line->[Word]
asWords = filter (/= []) . foldr (break_on ' ') [[]]

-------------------------------------line lists as paragraphs
                                         --no empty paragraphs

asParas:: [Line]->[Para]
asParas = filter (/= []) . foldr (break_on []) [[]]
```

Wesentliche Zutat: Wir filtern jetzt *leere Wörter* und leere *Paragraphen* heraus (sind funktionslos in der angestrebten Strukturierung von Texten) !

Beispiel (`microText` ist in einem Skript als der unten gezeigte String vereinbart):

```
Main> microText
"Stille Nacht, \n heilige Nacht, \n\n alles
schlaeft,\n einsam wacht...\n"
Main> asLines microText
["Stille Nacht, ", " heilige Nacht, ", "", " alles schlaeft,", " einsam wacht...", ""]
Main> asParas (asLines microText)
[["Stille Nacht, ", " heilige Nacht, "], [" alles schlaeft,", " einsam wacht..."]]
Main> map (map asWords) (asParas (asLines microText))
[[["Stille", "Nacht,"], ["heilige", "Nacht,"]], [["alles", "schlaeft,"],
["einsam", "wacht..."]]]
                     |||-- Liste von Paras
                     ||
                     ||--- Liste von Zeilen
                     |
                     |---- Liste von Worten
```

Unsere drei Funktionen lassen sich jtzt zusammensetzen, um die *Struktur* eines Texts zu analysieren (to parse = zerteilen):

```
parse:: Text->[[[Word]]]
parse = map(map asWords) . asParas . asLines
```

also z. B.:

```
Main> microText
"Stille Nacht, \n heilige Nacht, \n\n alles schlaeft,\n einsam wacht...\n"
Main> parse microText
[[["Stille", "Nacht,"], ["heilige", "Nacht,"]], [["alles", "schlaeft,"],
["einsam", "wacht..."]]]
```

also:

```
<Text> ::= [<Para>]
<Para> ::= [<Line>]
<Line> ::= [<Word>]
<Word> ::= <print_char> {<print_char>}
```

(diese syntaktischen Variablen sind natürlich nicht identisch mit den Haskell Typen)
jetzt neue Aufgabe:
mache Strukturanalyse wieder *rückgängig* (d. h. mache das Resultat von parse wieder „flach“)
dazu: füge alle nötigen *Separatoren* wieder ein!
Dieses Einfügen formulieren wir gleich allgemein, für beliebige Listen ⇒ `concWith`
und wenden diese Operation an mit `foldr1` (=kein Startelement!)

```
-------------------------------allgemeines concWith fuer Listen

concWith:: a->[a]->[a]->[a]
concWith x ys zs = ys ++ [x] ++ zs

------------------------------------------------Umkehrfunktionen
                                         --re-insert Separatoren

unLines:: [Line]->Text
unLines = foldr1 (concWith '\n')

unWords:: [Word]->Line
unWords = foldr1 (concWith ' ')

unParas:: [Para]->[Line]
unParas = foldr1 (concWith [])
```

beachte:

- ***keine* exakten *Umkehrfunktionen*, da wir überflüssige Separatoren entfernt haben;**
- **funktioniert wegen foldr1 *nicht* für *leere* Listen!** !

```
Main> (unLines.asLines) "abc \ndef\n"
"abc \ndef\n"
Main> (asLines.unLines)["abc ", "def", ""]
["abc ", "def", ""]
Main> (unWords.asWords) "abc  def "
"abc def"
Main> (asWords.unWords) ["abc", "def"]
["abc", "def"]
Main> (unWords.asWords) ""
"
Program error: foldr1 (concWith ' ') []
```

```
Main> (unWords.asWords) " "
"
Program error: foldr1 (concWith ' ') []
```

damit dann die gewünschte Umkehrung von parse:

```
unparse:: [[[Word]]]->Text
unparse = unLines . unParas . map(map unWords)
```

Beispiel:

```
Main> (unparse.parse) microText
"Stille Nacht,\nheilige Nacht,\n\nalles schlaeft,\neinsam wacht..."

             = vereinfachte (nicht redundante) Form des Ausgangstexts


simplify:: Text->Text        --entferne redundante Separatoren
simplify = unparse . parse
```

Beispiel:

```
Main> simplify microText
"Stille Nacht,\nheilige Nacht,\n\nalles schlaeft,\neinsam wacht..."
Main> putStr (simplify microText)
Stille Nacht,
heilige Nacht,

alles schlaeft,
einsam wacht...
```

Analyse der Struktur eines Texts durch parse kann als Ausgangspunkt dienen für viele weitere nützliche Arbeiten, z. B.:

- Eigenschaften eines Texts ermitteln,
- Änderungen des Texts vornehmen,
- Druckbild gestalten, ...

Beispiel: Texteigenschaften:

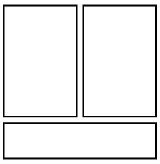
```
countLines = length . asLines
countWords = length . concat. map asWords . asLines
countParas = length . asParas . asLines


Main> countLines microText
6
Main> countWords microText
8
Main> countParas microText
2
```

Beispiel: Layout
wir betrachten einige einfache Möglichkeiten, aus einem Text ein gewünschtes *Druckbild* zu erzeugen;
dabei *Einschränkungen*:

- benutze *Zeilenstruktur*, die der Hersteller des Texts vorgegeben hat (also kein Zeilenumbruch, aber auch kein Umbruch größerer Einheiten wie z. B. Seiten)
- und normiere Zeilen lediglich durch Einfügen von Füllmaterial (oder Abschneiden) am *rechten* Rand!

unser *Layout* arbeitet mit *Textblöcken* = Folgen von Zeilen gleicher Länge
und arrangiert diese durch vertikale und horizontale *Komposition*, z. B.:



Textblöcke sind bei uns einfach Listen von Zeichen, und die fällige Normierung der Länge (Einfügen / Abschneiden rechts) formulieren wir gleich in voller Allgemeinheit für beliebige Listen (weil wir damit dann auch die Höhe von Textblöcken normieren können):

```
--Textblock = Char-Rechteck, d.h. Folge von gleichlangen Zeilen

type Block = [Line]                              --type synonym

------------------------------------------------Hilfsfunktionen

copy:: Int->a->[a]               --Liste mit n-mal elem
copy (n+1) elem = elem : copy n elem
copy _ _ = []

ljustify:: Int->a->[a]->[a]    --normiert Liste auf Laenge n
                               --rechts Abschneiden/Auffuellen
ljustify n filler list
      = take n list ++ copy (n - (length list)) filler
```

Beispiel:

```
Main> copy 9 'a'
"aaaaaaaaa"
Main> copy 9 "a"
["a", "a", "a", "a", "a", "a", "a", "a", "a"]
Main> copy (-2) "a"
[]
Main> ljustify 10 '*' "abc"
"abc*******"
Main> ljustify 10 0 [1,2,3,4,5]
[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]
Main> ljustify 0 ' ' "abc"
""
```

Ein *Textblock* entsteht durch Normierung von *Zeilenlänge* (=Breite des Blocks) und *Zeilenanzahl* (=Höhe des Blocks)

```
----------------------------line lists as height*width blocks

asBlock:: Int->Int->[Line]->Block
asBlock height width
       = ljustify height (copy width ' ')           --Hoehe
               . map (ljustify width ' ')           --Breite
```

Beispiel:

```
Main> asBlock 3 10 ["erste Zeile", "zweite Zeile"]
["erste Zeil", "zweite Zei", "          "]
Main> asBlock 3 15 ["erste Zeile", "zweite Zeile"]
["erste Zeile    ", "zweite Zeile   ", "               "]
Main> putStr (unLines (asBlock 3 15 ["erste Zeile", "zweite Zeile"]))
erste Zeile
zweite Zeile

Main> putStr (unLines (asBlock 2 10 ["Zeile", "Zeile", "keine Zeile"]))
Zeile
Zeile
Main> ((asBlock 5 30).asLines.simplify) microText
["Stille Nacht,                 ", "heilige Nacht,                ", "
                  ", "alles schlaeft,               ", "einsam wacht...
        "]
Main> (putStr.unLines.(asBlock 5 30).asLines.simplify) microText
Stille Nacht,
heilige Nacht,

alles schlaeft,
einsam wacht...
```

Textblöcke (geeigneter Dimension) lassen sich *horizontal* und *vertikal* kombinieren:

```
----------------------------------Komposition von Textbloecken

combHori, combVerti:: Block->Block->Block

combHori = zipWith (++)          --requires equal heights!!!
combVerti = (++)                 --requires equal widths!!!
```

Dabei ist zipWith eine higher-order Standardfunktion:

```
zipWith :: (a->b->c)->[a]->[b]->[c]

zipWith op (x:xs) (y:ys) = x `op` y : zipWith op xs ys
zipWith _ _ _            = []      -- eine Liste erschöpft
```

Es folgt ein Layout-Beispiel für

```
microText = "Stille Nacht ..."     -- wie gehabt
```

und

```
headline = "Weihnachten in Salzburg"
```

Layout-Beispiel:

```
layout1 =
  (putStr.unLines)
    (combVerti
       (asBlock 1 30 [])
       (combVerti
          (((asBlock 2 30).asLines.simplify) headline)
          (((asBlock 6 30).asLines.simplify) microText)))


layout2 =
```

```
  (putStr.unLines)
   (combVerti
    (asBlock 1 34 [])
    (combVerti
     (((asBlock 2 34).asLines.simplify) headline)
     (combHori
      (((asBlock 3 17).asLines.simplify) microText)
      (((asBlock 3 17).(drop 3).asLines.simplify) microText))))
```

```
Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
text_proc
Main> layout1

Weihnachten in Salzburg

Stille Nacht,
heilige Nacht,

alles schlaeft,
einsam wacht...

Main> layout2

Weihnachten in Salzburg

Stille Nacht,    alles schlaeft,
heilige Nacht,   einsam wacht...
```

3.7 Effizienz

Die oft große *Ökonomie* in der *Formulierung* einer Lösung darf nicht darüber hinwegtäuschen, daß es erhebliche *Kosten* (*Laufzeit*, *Speicherplatz*) bei der *Ausführung* geben kann;

deshalb immer nötig:

- Anwendung allgemeiner Strategien (z. B. divide & conquer)
- Verständnis des Ausführungs- (Reduktions-) Prozesses
- Modellierung/Tests zur Lokalisierung von Problemen

Beispiel:
die Funktion *„aslines"* in 3.6 wurde mit *foldr* definiert, kann aber in analoger Weise auch mit *foldl* abgestützt werden!

Unterschiede bei Definition:
Anhängen *von rechts* (ans Ende) des jeweils bereits gewonnen Teilergebnisses.

```
------------------------asLines: realisiert mit foldr und foldl

copy:: Int->a->[a]                --Liste mit n-mal elem
copy (n+1) elem = elem : copy n elem
copy _ _ = []

break1:: Char->[[Char]]->[[Char]]
break1 c ls
```

```
   |c == '\n' = [[]] ++ ls                      --new group
   |otherwise = [[c]++(head ls)] ++ (tail ls)  --add to group

asLines1:: [Char]->[[Char]]
asLines1  = foldr break1 [[]]


break2:: [[Char]]->Char->[[Char]]
break2 ls c
   |c == '\n' = ls ++ [[]]                      --new group
   |otherwise = (init ls) ++ [(last ls)++[c]]  --add to group

asLines2:: [Char]->[[Char]]
asLines2  = foldl break2 [[]]

{----------------------------Beispielanwendungen asLines1/2:

Main> length (asLines1 (copy 1000 'a'))
1
(15023 reductions, 32026 cells)
Main> length (asLines2 (copy 1000 'a'))
1
(14023 reductions, 30026 cells, 1 garbage collection)

Main> length (asLines1 (copy 1000 '\n'))
1001
(19024 reductions, 32033 cells)
Main> length (asLines2 (copy 1000 '\n'))
1001
(518524 reductions, 1531533 cells, 17 garbage collections)

-----------------------------------------------------------}
```

also: im Falle von '\n' = Beginn neuer Zeile *gewaltige* Verschlechterung bei *foldl*-Version!

genauere Analyse *sehr mühsam,*

hier nur:

- Laufzeit (Reduktionen)
- einfacheres (aber ähnliches) Beispiel.

zur *Erinnerung*:
foldr $\otimes$ a $[x_1,\ldots,x_n]$
$\Rightarrow x_1 \otimes$ (foldr $\otimes$ a $[x_2,\ldots,x_n]$
$\vdots$
$\Rightarrow x_1 \otimes (x_2 \otimes (\ldots(x_n \otimes \text{a})\ldots))$

foldl $\otimes$ a $[x_1,\ldots,x_n]$
$\Rightarrow$ foldl $\otimes$ (a $\otimes\, x_1$) $[x_2,\ldots,x_n]$
$\vdots$
$\Rightarrow (\ldots((\text{a} \otimes x_1)\otimes x_2)\otimes \ldots)\ldots)\otimes x_n$

dann: wichtig, ob Operator $\otimes$ strikt ist oder nicht!
Beispiele:

Addition (+)	= strikt
log. Und (&&)	= nicht strikt in 2.Argument, falls 1.Argument False: False && ? == False

```
------------------------------------------fold with strict op
Main> foldr (+) 0 [1..1000]
500500
(13020 reductions, 17032 cells)
Main> foldl (+) 0 [1..1000]
500500
(13020 reductions, 18032 cells)

---------------------------------------fold with non-strict op

Main> foldr (&&) True [False,True,True,True,True,True,True,True,True,True]
False
(15 reductions, 31 cells)
Main> foldr (&&) True [True,True,True,True,True,False,True,True,True,True]
False
(25 reductions, 41 cells)
Main> foldr (&&) True [True,True,True,True,True,True,True,True,True,False]
False
(33 reductions, 49 cells)

Main> foldl (&&) True [False,True,True,True,True,True,True,True,True,True]
False
(34 reductions, 59 cells)
Main> foldl (&&) True [True,True,True,True,True,False,True,True,True,True]
False
(34 reductions, 59 cells)
Main> foldl (&&) True [True,True,True,True,True,True,True,True,True,False]
False
(34 reductions, 59 cells)

--------------------------------------------------------------
```

Analyse:

Fall (+): $\approx$ *gleiche* Effizienz!

foldr (+) 0 [1,…,1000]
… ⇒ 1+(2+ …+(1000+0)…)

foldl (+) 0 [1,…,1000]
… ⇒ (…((0+1)+2)…)+1000

in *beiden* Fällen Aufbau des *ganzen* Ausdrucks, bevor erste Addition …

(! lazy evaluation: Bei foldl werden die Teilsummen unausgewertet mitgeschleppt; es gibt eine Variante von foldl, die das nicht macht und in manchen Situationen besser ist)
Fall (&&): unterschiedliche Effizienz!

foldr (&&) True [True, …, False, …, True]
… ⇒ True && (…(False && (foldr (&&) True [True, …, True])…)
aber: False && „irgendwas“ = False
d. h.: *fertig* sobald False angetroffen

foldl (&&) True [True, …, False, …, True]
… ⇒ (…(…(True && True) && …) && False) && …)
d. h.: erst ganzen Ausdruck aufbauen

```
-------------------------------------------------Konkatenation
concTwo:: Int->Int
```

```
concTwo m = length ((copy m 'a') ++ "0123456789")

{----------------------------------------------Anwendungen:

Main> concTwo 1
11
(84 reductions, 121 cells)
Main> concTwo 10
20
(219 reductions, 346 cells)
Main> concTwo 100
110
(1569 reductions, 2597 cells)
Main> concTwo 1000
1010
(15069 reductions, 25098 cells, 1 garbage collection)
Main> concTwo 10000
10010
(150069 reductions, 250099 cells, 2 garbage collections)

-----------------------------------------------------------}
```

Zahlen lassen vermuten, daß Aufwand für (++) *linear* abhängt von *Länge* des *linken* Arguments;
in der Tat:

$$\left.\begin{array}{l} [\,] + +ys = ys \\ (x : xs) + +ys = x : (xs + +ys) \end{array}\right\} Def.\ (++)$$

d. h.

(++) braucht *linkes* (*nicht* rechtes) Argument und durchläuft dies *elementweise*, also Aufwand bestimmt durch *Länge* des *linken* Arguments!

jetzt: fold zusammen mit *(++)*:

```
---------------------------einfacheres foldr/foldl (++) Beispiel

right, left:: Int->Int
right m = length (foldr (++) [] (copy m "0123456789"))
left m  = length (foldl (++) [] (copy m "0123456789"))

{--------------------------------------Beispielanwendungen

Main> right 1
10
(90 reductions, 145 cells)
Main> right 10
100
(729 reductions, 1118 cells)
Main> right 100
1000
(7119 reductions, 10839 cells)
Main> right 1000
10000
(71019 reductions, 108040 cells, 2 garbage collections)

Main> left 1
10
(80 reductions, 116 cells)
Main> left 10
100
(1079 reductions, 2178 cells)
```

```
Main> left 100
1000
(55619 reductions, 156439 cells, 1 garbage collection)
Main> left 1000
10000
(5056019 reductions, 15064040 cells, 163 garbage collections)

------------------------------------------------------------}
```

Vermutung:	right	*linear*
	left	*quadratisch*

Analyse:
(grob, unter Vernachlässigung von copy)

sei $l_1 = l_2 = \ldots = l_m =$ „0123456789"

dann:
foldr (++) [] $[l_1, \ldots, l_m]$
$\Rightarrow l_1$ ++ (foldr (++) [] $[l_2, \ldots, l_m]$)
$\vdots$
$\Rightarrow$ '0': ('1': …('9': (foldr (++) [] $[l_2, \ldots, l_m]$)…)
$\Rightarrow$ usw.

also (++) wird *n-mal* angewendet auf ein linkes Argument der *Konstanten* Länge 10, d.h. Schrittzahl $\hat{=}$ *O(n)*

foldl (++) [] $[l_1, \ldots, l_m]$
$\Rightarrow$ foldl (++) ([] ++ l_1) $[l_2, \ldots, l_m]$
$\Rightarrow$ foldl (++) (([] ++ l_1) ++ l_2) $[l_3, \ldots, l_m]$
$\vdots$
$\Rightarrow$ (…(([] ++ l_1) ++ l_2) ++ …) ++ l_m

also (++) wird *n-mal* angewendet auf ein linkes Argument der *wachsenden* Länge 0, 10, 20, …, (m-1)*10, d.h. Schrittzahl $\hat{=}$ O(m^2)

3.8 Unendliche Listen

- entstehen aus *nicht-terminierenden* Berechnungen (die Listen erzeugen),
- sind nützliches Zwischenergebnis zur *Komposition* von Teilberechnungen,
- sind *praktikabel* wegen lazy evaluation: unendliche Liste wird *nicht* komplett erzeugt, sondern nur soweit *Bedarf* besteht (es ist nützlich, sich *vorzustellen*, eine solche Liste sei tatsächlich vorhanden – vgl. Mathematik: z.B. {x < 100 | x $\in N$} obwohl man immer nur *endliche* Anfangsstücke braucht!)

Beispiel: Wiederholung ohne Ende:

```
iterate :: (a->a)->a->[a]
iterate f x = x : (iterate f (f x))
```

also: [x, f x, f(f x), …]

```
Main> iterate (+2) 1
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, ...
1537, 1539, 1541, 1543, 1545, 1547, 1549, Interrupted!     --Control C
Main> take 5 (iterate (+2) 1)
[1, 3, 5, 7, 9]
(120 reductions, 209 cells)
Main> take 1 (drop 99 (iterate (+2) 1))
[199]
```

```
(1324 reductions, 1933 cells)
Main> take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
(239 reductions, 437 cells)
Main> take 10 [1,3..]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
(224 reductions, 415 cells)
```

dabei *Kurzformen* wie:
[1..] für iterate (+1) 1
[1,3..] für iterate (+2) 1

Analyse:
take 2 (iterate (+2) 1)
⇒ take 2 (1 : iterate (+2) (1+2))
⇒ 1 : (take 1 (iterate (+2) (1+2)))
⇒ 1 : (take 1 (iterate (+2) 3))
⇒ 1 : (take 1 (3 : (iterate (+2) (3+2)))
⇒ 1 : 3 : (take 0 (iterate (+2) (3+2)))
⇒ 1 : 3 : []

3.9 Beispiel: Erzeugung von Primzahlen

Eratosthenes erfand die *Siebmethode*:
starte mit L=[2,3,4,5,...]
loop
entferne alle Vielfachen von head L aus L
end loop

wobei man als *Invariante* hat:
head L ist Primzahl, da von keiner kleineren Primzahl geteilt!
Also muß man unterwegs bloß alle head L ausgeben!
Die Schleife für das wiederholte Sieben läßt sich durch iterate realisieren, für das Sieben selbst definieren wir eine Funktion „crossout", die eine geeignete Anwendung von „filter" ist:

```
--------------------------------------------------prime numbers

crossout::[Int]->[Int]         --elim multiples of head element
crossout (x:xs) = filter (\y->y`rem`x /= 0) xs

primeNumbers :: [Int]
primeNumbers = map head (iterate crossout [2..])

{-
Main> take 10 (crossout [2..])
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
Main> take 10 primeNumbers
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
Main> take 1 (drop 9 primeNumbers)  -- 10. Primzahl
[29]
Main> primeNumbers !! 9   --Indexierung mit (!!) beginnt mit 0
29
Main> primeNumbers !! 99
541
Main> primeNumbers !! 999


ERROR: Garbage collection fails to reclaim sufficient space
```

```
-}
```

Erklärung: iterate crossout [2..] liefert *unendliche Liste* von *unendlichen Listen:*

$$
\begin{array}{llllll}
[[& 2, & 3, & 4, & 5, & \ldots\ ], \\
[& 3, & 5, & 7, & 9, & \ldots\ ], \\
[& 5, & 7, & 11, & 13, & \ldots\ ], \\
 & \vdots & & & & \\
 & & & & & \ ]
\end{array}
$$

die Primzahlen selbst kriegt man durch map head:

$$
\begin{array}{l|l|llll}
\cline{2-2}
[[& 2, & 3, & 4, & 5, & \ldots\ ], \\
[& 3, & 5, & 7, & 9, & \ldots\ ], \\
[& 5, & 7, & 11, & 13, & \ldots\ ], \\
 & \vdots & & & & \\
 & & & & & \ ] \\
\cline{2-2}
\end{array}
$$

3.10 Beispiel: Numerische Differentation und Integration

Wir betrachten zunächst nochmal numerische Differentation:

die mathematische Definition

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

führt für beliebige feste Wahl der Intervallbreite h auf

```
simpleDiff:: (Double->Double)->Double->Double->Double
simpleDiff f x h = (f(x+h)-f(x))/h
```

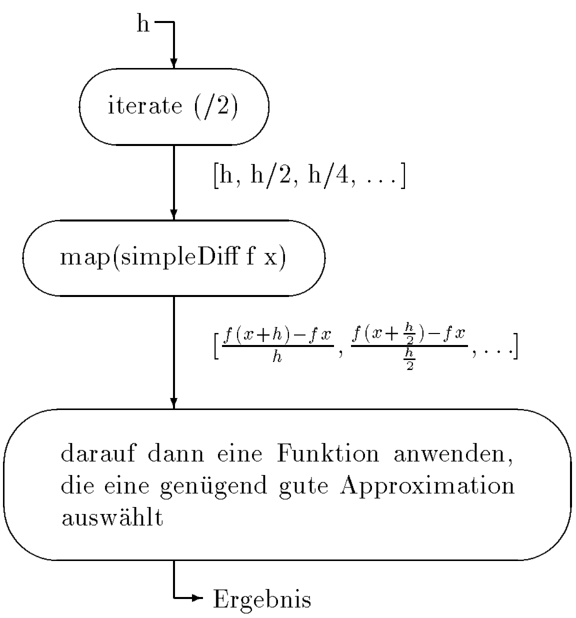
Wir erzeugen jetzt eine *Folge* von Näherungswerten, indem wir simpleDiff auf eine *Folge* von *abnehmenden Intervallbreiten* anwenden:

iterate (/2) liefert Folge von Intervallbreiten durch fortgesetzte Halbierung

damit dann:

```
seqDiff:: (Double->Double)->Double->Double->[Double]
seqDiff f x = map (simpleDiff f x) . iterate (/2)
```

also:



⇒ vgl. Seite 53

jetzt Problem:
läßt sich die „Konvergenzgeschwindigkeit“ verbessern?

plausibel:
Fehler eines Näherungswertes *hängt ab* von *Intervallbreite* h, also im einfachsten Fall *lineare* Abhängigkeit:
A_n = W + K*h
wobei:
A_n = n-te Approximation
W = wahrer Wert
K = Koeffizient

also:
A_n = W + K*h
A_{n+1} = W + K*h/2
⇒ W = 2*A_{n+1} - A_n

so definierte W-Werte sind natürlich auch noch ein bißchen falsch, d. h. Näherung
⇒ mache daraus wieder eine Folge!

$$\text{seqDiff:} \quad \begin{array}{ccccccccc} [& A_1 & & A_2, & & A_3, & \dots &] \\ & \downarrow & \swarrow & \downarrow & \swarrow & & & \\ [& W_1 & & W_2, & & & \dots &] \end{array}$$

diese zweite Folge wird erzeugt durch:

```
betterSeqDiff:: (Double->Double)->Double->Double->[Double]
betterSeqDiff f x = elimFstOrdError . seqDiff f x
```

wobei:

```
elimFstOrdError:: [Double]->[Double]
elimFstOrdError (x1:x2:xs)       --assumes halving of distances
     = (2*x2-x1):(elimFstOrdError (x2:xs))
--      -------unsere Formel
```

⇒ Seite 53

```
--------------------------------------infinite lists: examples

-------------------------------------------------------numerics
--------------------------------------------auxiliary functions


withinEps:: Double->[Double]->Double
withinEps eps (x1:x2:xs)
   |abs(x1 - x2) <= eps = x2
   |otherwise           = withinEps eps (x2:xs)


elimFstOrdError:: [Double]->[Double]
elimFstOrdError (x1:x2:xs)       --assumes halving of distances
     = (2*x2-x1):(elimFstOrdError (x2:xs))


-------------------------------------------------differentiation
simpleDiff:: (Double->Double)->Double->Double->Double
simpleDiff f x h = (f(x+h)-f(x))/h
```

```
seqDiff:: (Double->Double)->Double->Double->[Double]
seqDiff f x = map (simpleDiff f x) . iterate (/2)

betterSeqDiff:: (Double->Double)->Double->Double->[Double]
betterSeqDiff f x = elimFstOrdError . seqDiff f x

{-
Main> take 5 (seqDiff cos 0 0.1)
[-0.0499582, -0.0249946, -0.0125003, -0.00625134, -0.00312805]
Main> take 5 (betterSeqDiff cos 0 0.1)
[-3.09944e-05, -5.96046e-06, -2.38419e-06, -4.76837e-06, 0.0]
Main> withinEps 0.001 (seqDiff cos 0 0.1)
-0.000762939
Main> withinEps 0.001 (betterSeqDiff cos 0 0.1)
-5.96046e-06
-}
-----------------------------------------------------integration
simpleIntegrate:: (Double->Double)->Double->Double->Double
simpleIntegrate f a b = (f a + f b) * (b-a)/2

seqIntegrate:: (Double->Double)->Double->Double->[Double]
seqIntegrate f a b = (simpleIntegrate f a b)
                      : (zipWith (+) (seqIntegrate f a m)
                                     (seqIntegrate f m b))
                      where m = (a+b)/2
                      --inefficient: recomputations of fa,fb,fm


{-
Main> take 10 (seqIntegrate sin 0 pi)
[-1.37323e-07, 1.5708, 1.89612, 1.97423, 1.99357, 1.99839, 1.9996, 1.9999, 1.99997, 1.99999]
Main> withinEps 0.001 (seqIntegrate sin 0 pi)
1.9999
-}
```

mit der *Integration* verfahren wir ähnlich:

ein grobes Verfahren ist offenbar:

```
simpleIntegrate:: (Double->Double)->Double->Double->Double
simpleIntegrate f a b = (f a + f b) * (b-a)/2
```

Wir gewinnen eine *Folge* von *Approximationswerten*, indem wir (wiederholt)

- das *Intervall* [a,b] *halbieren*
- die *Ergebnisse* für beide Hälften *addieren*

```
seqIntegrate:: (Double->Double)->Double->Double->[Double]
seqIntegrate f a b = (simpleIntegrate f a b)
                      : (zipWith (+) (seqIntegrate f a m)
                                     (seqIntegrate f m b))
                      where m = (a+b)/2
                      --inefficient: recomputations of fa,fb,fm
```

beachte: Ergebnis wieder *Folge* von Näherungswerten, aber *komplexere Struktur* der Rekursion: Baum, (Kaskade) !

3.11 Beispiel: Modellierung von Schaltungen

Wir basteln uns ein *Flipflop!*

Zustand des Flipflops	$\widehat{=}$	Belegung von q, q' (q= ¬ q')
Zustand *unverändert*	$\widehat{=}$	(set, reset) = (1, 1)
Veränderung	$\widehat{=}$	set = 0 ⇒ q = 0
		reset = 0 ⇒ q' = 0

Modellierung:

- set, reset, q, q' $\widehat{=}$ unendliche Listen von Werten, wobei jede Position in Liste einem Zeit-Takt entspricht
- *Verzögerung* (3 Zeiteinheiten) damit Rückkopplung möglich
- Sonderfall: *undefinierter* Wert

```
------------------------------------------------ flip-flop model

data ThreeVal = U | L | H          -- undefined, low, high
     deriving Show

nand :: ThreeVal->ThreeVal->ThreeVal
nand H H = L
nand L _ = H
nand _ L = H
nand _ _ = U

nandSeq :: [ThreeVal]->[ThreeVal]->[ThreeVal]
nandSeq = zipWith nand


q, q', set, reset :: [ThreeVal]

q  = U : U : U : nandSeq set q'
q' = U : U : U : nandSeq reset q

set   = [L,L,L,L,L,L,L,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,H]
reset = [H,H,H,H,H,H,H,H,H,H,H,H,H,H,H,L,L,L,L,L,L,L,H,H,H,H,H]

{-
Main> q
[U, U, U, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, L,
L, L, L, L, L, L, L, L]
Main> q'
[U, U, U, U, U, U, L, L, L, L, L, L, L, L, L, L, L, H, H, H, H,
H, H, H, H, H, H, H, H]
-}
```

Kapitel 4

Benutzerdefinierte Datentypen und Typklassen

4.1 Algebraische Datentypen

= „konkrete“ Datentypen,
Konstruktion der Elemente des Types liegt offen und ist Basis für

- Benennung einzelner Elemente
- Definition von Funktionen über dem Typ

Beispiel: ⇒ Seite 56
Beachte:

- linke Seite: Schlüsselwort “data”, dann Typname
- rechte Seite: *Konstruktionsschema* für Elemente des Typs, unter Verwendung von:
  - Alternativsymbol “|”
  - andere Typen
  - Konstruktoren, d. h. 0,1,2 ...stellige Funktionen
- Interpretation: zum Typ gehört alles, was sich aus wiederholter Anwendung des Konstruktionschemas erzeugen läßt, vgl. Mathematik:

$$0 \in N$$

$$n \in N \Rightarrow succ(n) \in N$$

  in Haskell-Stil:

```
data N = Zero | Succ N
```

```
----------------------------------------- algebraic data types
---------------------------------------------- simple examples

data PrimaryColour = Yellow | Red | Blue   -- enumeration type

data Points = Point Float Float  -- typle type, based on Float

type Name = String
type MatrNr = Int
```

```
data StudId = Stud Name MatrNr
                        -- record type, based on type synonyms

data Pairs a = Pair a a
                -- polymorphic type, based on arbitrary type a

data Union a b = Fst a | Snd b
               -- union type, based on arbitrary types a and b

data Tree a = Inner a (Tree a) (Tree a) | Leaf a
                                 -- recursive polymorphic type

{-
Main> :t Stud "Franz Gans" 4711
Stud "Franz Gans" 4711 :: StudId
Main> :t Point 0.0 0.0
Point 0.0 0.0 :: Points
Main> :t Leaf "Franz Ganz"
Leaf "Franz Ganz" :: Tree [Char]
Main> :t Fst 3.14
Fst 3.14 :: Fractional a => Union a b
Main> :t Inner 1 (Inner 2 (Leaf 3) (Leaf 4)) (Leaf 5)
Inner 1 (Inner 2 (Leaf 3) (Leaf 4)) (Leaf 5) :: Num a => Tree a
Main> Inner 1 (Inner 2 (Leaf 3) (Leaf 4)) (Leaf 5)

ERROR: Cannot find "show" function for:
*** expression : Inner 1 (Inner 2 (Leaf 3) (Leaf 4)) (Leaf 5)
*** of type    : Tree Int
-}
```

beachte:

- Großbuchstaben;
- Typname darf als Konstruktorbezeichner verwendet werden, also zB:

  ```
  data Point a = Point a a
  ```

Unterscheide:
Konstruktorfunktionen und „Programm"-Funktionen:

1. Konstruktorfunktionen erlauben *keine Auswertung* im Sinne einer Reduktion auf einen einfacheren Wert; sowas wie

   ```
   Inner 1 (Inner 1 (Leaf 1) (Leaf 2)) (Leaf 3)
   ```

   ist Darstellung von

   ```
        4
       / \
      1   3
     / \
    1   2
   ```

   Situation ist völlig analog zu *eingebauten* Datentypen, außer daß die keine Wortsymbole verwenden, zB: : und [] für Listen.

2. Konstruktoren dürfen in *patterns* auftreten und erlauben so die Definition von Funktionen über dem Datentyp.

```
-------------------------------------------some Tree functions

data Tree a = Inner a (Tree a) (Tree a) | Leaf a
                                 -- recursive polymorphic type

miniTree = Inner 1 (Inner 2 (Leaf 3) (Leaf 4)) (Leaf 5)

listInorderTree :: (Show a) => Tree a -> String
listInorderTree (Leaf x) = show x
listInorderTree (Inner x yt zt)
      = "<" ++ (listInorderTree yt) ++
                  "<" ++ (show x) ++ ">" ++
                           (listInorderTree zt) ++ ">"

{-
Main> listInorderTree miniTree
"<<3<2>4><1>5>"
-}

sizeOfTree :: Tree a -> Int
sizeOfTree (Leaf x) = 1
sizeOfTree (Inner x yt zt) = 1 + (sizeOfTree yt) + (sizeOfTree zt)


mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Inner x yt zt) = (Inner (f x) (mapTree f yt) (mapTree f zt))

{-
Main> listInorderTree miniTree
"<<3<2>4><1>5>"
Main> sizeOfTree miniTree
5
Main> listInorderTree (mapTree (*100) miniTree)
"<<300<200>400><100>500>"
-}
```

Beispiel:

```
--------------------------------------SuchBaeme

data Ord a => STree a = Node a (STree a) (STree a) | Nil
        -- polymorphic tree type with constrained on base type
        -- note: empty leaves

insertSTree :: Ord a => a -> STree a -> STree a
insertSTree x Nil = Node x Nil Nil
insertSTree x (Node y left right)
   | x<=y       = Node y (insertSTree x left) right
   | otherwise  = Node y left (insertSTree x right)

elemSTree :: Ord a => a -> STree a -> Bool
elemSTree x Nil = False
elemSTree x (Node y left right)
```

```
   | x==y       = True
   | x<y        = elemSTree x left
   | otherwise  = elemSTree x right

listToSTree :: Ord a => [a] -> STree a
listToSTree = foldr insertSTree Nil

sTreeToList :: Ord a => STree a -> [a]
sTreeToList Nil = []
sTreeToList (Node x left right)
   = (sTreeToList left) ++ [x] ++ (sTreeToList right)

miniList = [1,5,-3,6,0,99,4,4,55,1000,-1000]

{-
Main> :t Node 3.14 Nil Nil
Node 3.14 Nil Nil :: (Ord a, Fractional a) => STree a
Main> :t Node cos Nil Nil

ERROR: a -> a is not an instance of class ``Ord''
Main> elemSTree 0 (listToSTree miniList)
True
Main> elemSTree 77 (listToSTree miniList)
False
Main> (sTreeToList.listToSTree) miniList  -- sort!
[-1000, -3, 0, 1, 4, 4, 5, 6, 55, 99, 1000]
Main> (sTreeToList.listToSTree) (map abs miniList)
[0, 1, 3, 4, 4, 5, 6, 55, 99, 1000, 1000]
-}
```

4.2 Abstrakte Datentypen

= *verberge* (verbiete Benutzung von) *Konstruktion* der Typelemente
Lösung in Haskell: Konventionell, nämlich:

- verpacke Typdefinition + einschlägige Operationsdefinitionen in *Modul*
- *exportiere nur* das, was der Anwender benutzen soll, also zB Typ- und Operationsbezeichner in „Exportliste“.

⇒ Seite 59

Haskells Modulkonzept wird von verschiedenen Hugs-Versionen nicht (vollständig) unterstützt; umgekehrt bieten Gofer/Hugs Systemeigene (nicht-Haskell) Syntax für abstrakte Datentypen. !

```
---------------------------------------------------------ADT module
module Searchtree(STree, listToSTree, sTreeToList) where

data Ord a => STree a = Node a (STree a) (STree a) | Nil

insertSTree :: Ord a => a -> STree a -> STree a
insertSTree x Nil = Node x Nil Nil
insertSTree x (Node y left right)
   | x<=y       = Node y (insertSTree x left) right
   | otherwise  = Node y left (insertSTree x right)

listToSTree :: Ord a => [a] -> STree a
```

```
listToSTree = foldr insertSTree Nil

sTreeToList :: Ord a => STree a -> [a]
sTreeToList Nil = []
sTreeToList (Node x left right)
   = (sTreeToList left) ++ [x] ++ (sTreeToList right)



--------------------------------------client module (separate file)

module Main where
import Searchtree           -- improt complete export of Searchtree

-----------------------------------------------------------use e.g.:

{-
Prelude> :l Main.hs
Reading file "Main.hs":
Reading file "Searchtree.hs":
Reading file "Main.hs":

Hugs session for:
/usr/local/share/hugs/lib/Prelude.hs
Searchtree.hs
Main.hs
Main> (sTreeToList.listToSTree) [1,-1,0,2,-2]
[-2, -1, 0, 1, 2]
Main> sTreeToList (insertSTree 2 Nil)  --insertSTree is hidden!
ERROR: Undefined variable "insertSTree"
Main> sTreeToList Nil                   -- constructors are hidden!
ERROR: Undefined constructor function "Nil"
-}
```

4.3 Klassen

sind in Haskell *Typklassen*, dh Kollektionen von Typen, die durch einen *gemeinsamen* Satz von *Operationen* ausgezeichnet sind.
Idee aus der *OO-Welt*, aber in Haskell:

- sowieso keine Objekte (mit änderbarem Zustand), sondern nur Werte
- nicht dynamische („runtime“) Bindung einer Funktion, sondern Typüberprüfung zur compile-time
- viele OO-Sprachen verwenden „Klasse“ wie „Typ“, in Ada ähnlich wie in Haskell:

  „Klasse“ = Familie von Typen, nämlich Definitionshierarchie
- Klassendefinitionen in Haskell sind normalerweise *abstrakt* (nur Interfaces, allerdings defaults für Funktionen möglich) und enthalten jedenfalls keine „Daten“-Definitionen
- *Klassen* können haben
  - „instances“ (Exemplare), dh Typen, die zur Klasse gehören
  - „subclasses“ (abgeleitete Klassen), dh Klassen, die die Operationen der Oberklassse „erben“.

instance	=	**Typ**
subclass	=	**class, dh Menge von Typen**

!

- es gibt Sortiment von vordefinierten Klassen wie zB Eq, Ord, Num, Show, Read u.a., die man typisch verwendet um

  1. bei *überladenen* Funktionen den Polymorphismus sinnvoll einzuschränken, dh die zulässige Auswahl von Typen anzugeben, zB:

     ```
     quicksort :: Ord a => [a]->[a]
                  \---/
                  Kontextklausel: a muß Instanz von Ord sein (mittelbar)
     ```

  2. bei der Neueinführung von *Datentypen* eine „*Grundausstattung*" mit *Operationen* herzustellen, zB: Datentyp zur Instanz von „Eq" machen.

- zu vordefinierten Klassen siehe Report, zB:

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

  Beispiel für *Unterklasse* von Eq ist Ord, die zu "==" noch die Operationen "<, <=, >=, >" sowie die Funktionen "min, max" (und noch eine Funktion "compare" die ein symbolisches Ergebnis liefert) definiert.

  Beispiel von *Instanzen* von Eq gibt's massenhaft, zB Int.

Beispiel: benutzerdefinierter Datentyp als Instanz von Eq:

```
--------------------------------------old defs do not support "=":

data PrimaryColour = Yellow | Red | Blue

---------------------------------------------------as is shown by:
{-
Main> Yellow == Blue

ERROR: PrimaryColour is not an instance of class "Eq"
-}

-------------------------so make new type an instance of class Eq:

data PrimaryColour = Yellow | Red | Blue
instance Eq PrimaryColour
   where Yellow == Yellow = True
         Red == Red       = True
         Blue == Blue     = True
         _ == _           = True

-----------------------------------------------------which allows:
{-
Main> Yellow == Blue
False
-}


-------------------------------------this can be done more simply:

data PrimaryColour = Yellow | Red | Blue deriving Eq

---------------------------------------------which gives the same:
```

```
{-
Main> Yellow == Blue
False
Main> Yellow /= Blue
True
-}

---------but other defs of "==" could be useful for colour blinds:

data PrimaryColour = Yellow | Red | Blue
instance Eq PrimaryColour
   where _ == _ = True

-----------------------------------------------------------giving:
{-
Main> Yellow == Blue
True
-}
```

Beispiel: Definition einer Klasse und zweier Instanzen

```
--------------------------------------now define your own classes:

class Mail m
   where giveAdr:: m->String
         setAdr:: String->m->m
         existsAdr:: m->Bool
         existsAdr x = (giveAdr x /= "")        -- default

data SimpleLetter a = SL String a        -- hat eine Adresse
instance Mail (SimpleLetter a)
   where giveAdr (SL s _) = s
         setAdr s (SL s' x) = (SL s x)

data BroadcastLetter a = BC [String] a   -- hat Liste von Adressen
clearAdr:: (BroadcastLetter a)->(BroadcastLetter a)
clearAdr (BC ss x) = BC [] x

instance Mail (BroadcastLetter a)
   where giveAdr (BC [] _)     = ""
         giveAdr (BC (s:ss) _) = s
         setAdr s (BC ss x) = BC (s:ss) x

---------------------------------------------------------examples:
{-
Main> existsAdr (SL "Grete" (25, "Now", 1997))
True
Main> giveAdr(setAdr "Gustav" (setAdr "Franz"
(SL "Grete" (25, "Nov", 1997))))
"Gustav"
Main> giveAdr (clearAdr (BC ["Grete"] (25, "Nov", 1997)))
""
Main> existsAdr (clearAdr (BC ["Grete"] (25, "Nov", 1997)))
False
-}
```

Anhang A

Übungen

Im Proseminar Programmieren III wurden im Wintersemester 1997/98 folgende Hausübungen aufgegeben.

A.1 Blatt 1

1. Machen Sie sich mit der Bedienung von Hugs und den begleitenden Dokumenten vertraut. Wenn gewünscht, kopieren Sie das Material und installieren Sie das System daheim.

2. Geben sie die folgenden Ausdrücke ein und korrigieren Sie im Falle von Fehlermeldungen die Syntax:

```
max 3 4
negate 3
3 `div` 4
div 3 4
div 3 -4
3/4
2**3 - 4
2 ** -3
2 **-3
fromIntegral 3 + 3.999
max 3.0 4
2 <= 3 || true
False ==  (2 <= 3)
"ABC" == ['A','B','C']
"A" = 'A'
"A" =='A'
let x = 'A' in x:"BC"
let x = "A" in x == x
let x = "A" in x /= x
let x = "A" in not x == x
let x = "A" in not (x == x)
not (let x = "A" in x == x)
let y = not (let x = "A" in x == x) in not y
```

   Informieren Sie sich in Anhang A, Standard Prelude, des Haskell Reports über Standardfunktionen und ihren Gebrauch und bilden Sie ähnliche Beispiele.

3. Schreiben Sie ein Haskell-Skript, das Funktionen zur Berechnug der Fläche einiger geometrischer Figuren enthält.

A.2 Blatt 2

1. Experimentieren Sie mit arithmetischen Ausdrücken und machen Sie sich die dabei zur Anwendung kommenden Regeln klar (⇒ Literatur); befassen Sie sich insbesondere mit Klammerung, Typfragen (Mischung von Float und Int), Grenzen des Zahlbereichs.

2. Geben sie eigene Haskell Definitionen (nebst Typ) für übliche Funktionen wie Fakultät, Fibonacci, ggT, max, k-aus-n und dergleichen, und probieren Sie die Eigenprodukte aus.

3. Geben sie eigene Haskell Definitionen für „und“ und „oder“ durch Fallunterscheidung (ohne Gebrauch der vorgegebenen Operationen) . Prüfen Sie anhand von Beispielen, ob Ihre Eigenprodukte mit “`&&`” und “`||`” übereinstimmen hinsichtlich a) des Verhaltens im Falle undefinierter/fehlerhafter Teilausdrücke (sowas wie `True || (1/0) == 0.001`) und b) der Bindekraft-/Assoziationsregeln. Wenn nicht: Kann man da was machen?

4. Schreiben Sie eine Funktion `integrate :: (Float->Float)->Float->Float`, die bei Aufruf mit `integrate f x` den Wert des Integrals von f im Intervall von 0 bis x liefert. Benutzen Sie zur Berechnung ein einfaches Näherungsverfahren (Unterteilung in eine feste oder auch von der Intervallbreite abhängige Zahl von Streifen).

5. Verändern Sie das in der Vorlesung gegebene Verfahren zur Nullstellenberechnug in ein Verfahren zur Berechnung der n-ten Wurzel einer Zahl x (n :: Int, n>1, x:: Float). Das neue Verfahren soll im Unterschied zum alten n und nicht eine reelle Funktion als Parameter haben, und es soll bei der Formulierung des Verbesserungsschritts nicht von numerischer Differentiation (also der Funktion diff) Gebrauch machen, sondern die jedermann bekannte Ableitung der Funktion x̂n benutzen.

A.3 Blatt 3

1. Das Hugs-System informiert bei entsprechender Einstellung (Kommando ":set +s") über den bei Auswertung eines Ausdrucks anfallenden Aufwand an Berechnungsschritten („reductions“) und Speicherplatz („cells“). Eine genaue und vollständige Interpretation der gelieferten Zahlenwerte ist nicht ganz einfach; sie geben aber auch ohne detailliertes Verständnis der Arbeit der Haskell-Maschine eine brauchbare Vorstellung von der Effizienz von Funktionsdefinitionen etc. — Informieren Sie sich über diese Option und probieren Sie sie anhand der fogenden Aufgaben und auch anderer Beispiele aus!

2. Die n-te Potenz einer Zahl k ist zu berechnen nach den beiden folgenden (in ihrer Effizienz sehr verschiedenen, vgl. Prog 1) Methoden:

   (a) wiederholtes Multiplizieren von k

   (b) wiederholtes Quadrieren (unter Halbieren des Exponenten).

   Hinweis: Beim Testen verlässt man leicht den implementierten Zahlbereich, was zu einem falschen Ergebnis (ohne Fehlermeldung) führen kann; wenn Sie große Exponenten ausprobieren, sollten Sie deshalb die triviale Basis 1 verwenden. Dabei gibt es dann noch die Möglichkeit, daß das System die Menge der unfertigen rekursiven Aufrufe nicht mehr bewältigen kann. Welche Fehlermeldung ergibt sich dann?

3. Ein Geldbetrag b sei zu wechseln, d.h. in gebräuchlichen Münzeinheiten (gegeben in einer Liste [m1, m2, ..mn] von Zahlenwerten) darzustellen.

   (a) Definieren Sie eine Funktion, die Betrag und Münzwertliste als Parameter hat und eine Darstellung (in Form einer Liste mit Münzanzahlen) des Betrags mit möglichst wenig Münzen liefert. (Nehmen Sie an, daß die Münzwertliste absteigend sortiert sei.)

   (b) Definieren Sie eine Funktion, die Betrag und Münzwertliste als Parameter hat und die Zahl aller möglichen Darstellungen des Betrags errechnet. Verwenden sie dazu die offensichtliche (?) Rekursion:
   Anzahl für b und Münzwertliste [m1, m2, ..mn] =
   Anzahl für b und Münzwertliste [m2, ..mn] + Anzahl für b-m1 und Münzwertliste [m1, m2, ..mn].

A.4 Blatt 4

1. Tupel (z.B. geordnete Paare, Tripel, Quadrupel, ...) sind wie Listen für alle möglichen Elementtypen definiert, haben aber jeweils feste Länge und können auch Elementtypen mischen, also die Aufgabe von records in anderen Sprachen übernehmen. Notation (für Typ und Elemente): runde Klammern, also z.B.:

```
()              -- nix drin !!
(4711)          -- wird mit Zahl 4711 identifiziert
(-1.0,1.0)
(sin,cos)
(sin 0,cos 0)
("Franz Gans", (30,"Okt",1950), "Salzburg")
```

(a) Stellen Sie den Typ dieser Beispiele fest.

(b) Für Paare gibt es vordefinierte Projektionsfunktionen fst und snd auf die beiden Komponenten; definieren Sie entsprechende Funktionen für Tripel.

(c) Die explizite Angabe von Tupeltypen ist oft mühsam (vgl. a)). Haskell erlaubt die Einführung von Typsynonymen, d.h. benutzerdefinierte Namen für vorhandene Typkonstruktionen. (Das ist wohlgemerkt die Einführung einer bequemeren Bezeichnung und nicht eines neuen Typs!!)

Beispiel: `type Point = (Float,Float) ; type Rectangle = Point,Point,Point,Point)`.

Geben Sie Definitionen für typische Datensätze wie z.B. für Personen oder Bibliotheksbücher zusammen mit einem Typsynonym, Projektionsfunktionen auf Komponenten und ein paar weitere sinnvolle Funktionen (z.B. aktuelles Alter einer Person, Anschriftänderung).

2. Sind zu einer Funktion f zwei Argumentwerte a und b mit $f(a) < 0 < f(b)$ bekannt, so kann man eine Nullstelle durch wiederholte Halbierung des Intervalls suchen. Definieren Sie eine entsprechende Funktion unter Verwendung der "until"-Funktion (Vorlesung oder Prelude) sowie von Float-Paaren für Intervalle.

3. Brüche lassen sich als Int-Paare darstellen. Geben Sie Definitionen von Funktionen für die vier Grundrechenarten für Brüche sowie eine Normalisierungsfunktion, die einen Bruch auf die übliche gekürzte Form mit positivem Nenner bringt.

A.5 Blatt 5

Bei den folgenden Aufgaben sollen Sie Gebrauch machen von Standardfunktionen zur Listenverarbeitung wie take, map, fold, filter, ...

1. Geben Sie so eine neue Definition der "how_many" Funktion des Beispiels aus 3.3.

2. Eine Liste wird sortiert, indem man sie in zwei (im Idealfall gleich große) Teillisten zerlegt, diese (nach demselben Verfahren) sortiert und dann wieder in geeigneter Weise vereinigt, z.B.:

   (a) merge sort:
   Teilung = Halbierung der Liste,
   Vereinigung = ordnungstreue Verzahnung ("merge") der beiden sortierten Listen.

   (b) quick sort:
   Teilung durch Vergleich mit Trennelement (z.B. erstes Element der Liste),
   Vereinigung durch Konkatenation der sortierten Listen.

   (Hinweis: man kann die Aufspaltung mit zwei "filter" Anwendungen, aber vielleicht auch in einem Durchlauf machen).

3. Um die Liste aller Permutationen einer Liste x:xs zu berechnen (das ist dann eine Liste von Listen), kann man wie folgt vorgehen:

   erstens, finde alle Permutationen von xs,
   zweitens, setze x auf alle denkbaren Weisen in jede der Permutationen ein.

A.6 Blatt 6

1. Verwenden Sie die in der Vorlesung definierten Funktionen zur Lösung eines nichttrivialen Layout-Beispiels.

2. Definieren Sie eine Funktion `asCBlock:: Int->Int->[Line]->Block`, in deren Resultat Textzeilen zentriert vorliegen, und geben Sie eine Beispielanwendung.

3. Bei einem Text soll für jedes vorkommende Wort die Häufigkeit seines Vorkommens ermittelt und in einer lexikographisch geordneten (Wort, Häufigkeit)-Liste ausgegeben werden. Verwenden Sie die Textanalyse-Funktionen der Vorlesung, aber lassen Sie dann als zu zählendes Wort nur eine Zeichenreihe gelten, die mit einem Alphabetzeichen beginnt. Damit die Zählung sinnvoll ist, entfernen Sie am Ende eines Wortes auch noch allfällige Satzzeichen. (Also: "4711" oder ":=" gilt nicht, "DIN-A4" gilt, "Kaffee?" gilt als "Kaffee").

Anhang B

Proseminar-Tests

B.1 Test 1

Die folgenden Aufgaben präsentieren Haskell-Ausdrücke, für die Sie jeweils den Wert angegeben sollen oder aber ob ein Syntax-, Typ- oder Laufzeitfehler vorliegt. (Vermerken Sie Ihre Antwort bitte auf diesem Blatt hinter dem Ausdruck und verwenden Sie für die drei Fehlerfälle die Abkürzungen SF, TF, LF !) Es wird der übliche Kontext (standard prelude) angenommen.

1.
```
max 3 4                        negate 3
3 `div` 4                      div 3 4
div 3 -4                       2**3 - 4
True && (1/0 == 0)             False ==  (2 <= 3)
"A" = 'A'                      "A" == 'A'
"ABC" == ['A','B','C']

let x = 'A' in x:"BC"          let x = "A" in x == x
let x = "A" in x /= x
let x = "A" in not x == x
let x = "A" in not (x == x)
```

2. Zur Erinnerung: Die Funktion max liefert den größeren zweier Zahlenwerte, und die polymorphe Identitätsfunktion id ist definiert durch `id:: a->a` und `id x = x`.

   Untersuchen Sie:

```
(id.max)3 2          (id.max 3) 2         id.(max 3) 2
id max 3 2           id (max 3) 2         (id max) 3 2
(id max 3) 2         id (max 3 2)         max id 3 2
```

3. Die Funktion f sei definiert als:

```
f:: (Int->Int->Int)->Int->[Int]->Int
f g e []     = e
f g e (x:xs) = g x (f g e xs)
```

   Untersuchen Sie:

```
f max 5 [4,3,2,1]
f div 1 [5,4,3,2]
```

B.2 Test 2

Die Aufgaben 1 und 2 präsentieren Haskell-Ausdrücke, für die Sie jeweils den Wert angegeben sollen oder aber ob ein Syntax-, Typ- oder Laufzeitfehler vorliegt. (Vermerken Sie Ihre Antwort bitte auf diesem Blatt hinter dem Ausdruck und verwenden Sie für die drei Fehlerfälle die Abkürzungen SF, TF, LF !) Es wird der übliche Kontext (standard prelude) angenommen.

1.
```
True || (1/0 == 0)          False == True && (1/0 /= 0)
True && 0 * (1/0)           False && 1/0
False || (True || (1/0 > 0))
let x = 1/(x - 1) in x == 1
```

2. Gegeben:

```
fix:: (Float->Float)->Float->Float
fix f = repeat_until pred f
          where pred x = (abs(f x - x) <= 0.01)
square:: Float->Float
square x = x*x
```

Was ist (ungefähr):

```
fix square 0.01
fix square 0.1
fix square 1.0
```

3. Die folgende Funktion soll feststellen, ob eine Liste den Anfang einer anderen bildet. Komplettieren Sie den dritten Fall der Definition (ohne weitere Fallunterscheidung!!).

```
initial_part:: [a]->[a]->Bool
initial_part [] _ = True
initial_part _ [] = False
initial_part (x:xs) (y:ys) =
```

4. Die Funktion `double:: Float->Float` sei definiert als `double x = 2 * x`. Geben Sie die Auswertungsschritte unter lazy evaluation für den Ausdruck

```
double (double (1+1)) =>
```

B.3 Test 3

Die Aufgaben 1 und 2 präsentieren Haskell-Ausdrücke, für die Sie jeweils den Wert angeben sollen oder aber ob ein Syntax-, Typ- oder Laufzeitfehler vorliegt. (Vermerken Sie Ihre Antwort bitte auf diesem Blatt hinter dem Ausdruck und verwenden Sie für die drei Fehlerfälle die Abkürzungen SF, TF, LF !) Es wird der übliche Kontext (standard prelude) angenommen.

1.
```
[[]] ++ [2] == [2]                [[]] ++ [2] == [[2]]
[[]] ++ [2] == [[],[2]]           [[]] ++ [[2]] == [[],[2]]
[[2]] ++ [] == [[2]]              [[2]] ++ [[2]] == [[2]]
[[2]] ++ [[2]] == [[2],[2]]       [[2]] ++ [] == [[2],[]]
```

2. Gegeben sind die Funktionen aus der Vorlesung: `asLines:: Text->[Line]`, `asWords:: Line->[Word]`, `asParas:: [Line]->[Para]` sowie die Definition `beisp = "ab\nx\n"`. Was ist:

```
(map(map asWords).asParas.asLines) beisp
(asParas.(map asWords).asLines) beisp
(length.concat.map asWords) (asLines beisp)
map (length.concat.map asWords) (asParas (asLines beisp))
```

3. Im folgenden soll eine alternative Definition der Konkatenation gegeben werden. Ergänzen Sie die fehlenden Teile (ohne Verwendung von `(++)`, d.h. ohne explizite Rekursion):

```
(++):: [a]->[a]->[a]
xs ++ ys = foldr op a b
           where op =
                  a =
                  b =
```

4. Die Funktion `subs` sei definiert als

```
subs [] = [[]]
subs (x:xs) = subs xs ++ map (x:) (subs xs)
```

   (a) Welche der folgenden Typdeklarationen paßt dazu:

`subs:: [a]->[a]`	`subs:: [a]->[[a]]`
`subs:: [[a]]->[a]`	`subs:: [[a]]->[[a]]`

   (b) Was ist: `subs [[1,2],[9]]`