

Einführung in die funktionale Programmierung

PD Stefan Bosse
Universität Koblenz-Landau, Fakultät Informatik
SS 2019
Revision 2019-07-19
sbosse@uni-koblenz.de

1. Inhalt

1. Inhalt	3
2. Überblick	3
2.1. Schwerpunkte in diesem Kurs	3
2.2. Literatur	4
2.3. Software	6
2.4. Ziele	8
2.5. Inhalte	9
2.6. Übung	9
2.7. Geschichte	10
2.8. Übersicht der Programmiersprachen	11
2.9. Funktionale Programmierung - Warum?	11
2.10. Kenntnis verschiedener Sprachen	12
2.11. Funktionale Programmiersprachen	13
2.12. Funktionale Programmierung - Geschichte	13
2.13. Funktionale Programmierung - Haskell	15
2.14. Interpreter	16
2.15. Just-in-Time Compiler	17
2.16. Haskell WEB Labor	17
3. Einführung in die Funktionale Programmierung	18
3.1. Mathematik	19
3.2. Terme	19
3.3. Zusammengesetzte Werte	19
3.4. Typen	20
3.5. Funktionen	21
3.6. Funktionen und Datenfluss	22
3.7. Funktionsrekursion	23
3.8. Funktionale Komposition	26
3.9. Funktionale Programmierung	27

4. Einführung in Haskell	28
4.1. Haskell	29
4.2. Haskell - Basistypen	29
4.3. Haskell - Terme	30
4.4. Haskell - Symbolische Variablen	30
4.5. Haskell - Bedingte Ausdrücke	31
4.6. Haskell - Listen und Tupel	33
4.7. Haskell - Mehrfachauswahl und Musteranpassung	35
4.8. Haskell - Funktionen	36
4.9. Haskell - Sequenzielle Ausführung	39
4.10. Haskell - Eingabe und Ausgabe	40
4.11. Haskell - Bedingte Ausführung	41
5. Funktionen und Lambda Calculus	41
5.1. Definition, Deklaration, Applikation, Komposition	42
5.2. Lambda Calculus	43
5.3. Funktionen als Werte	45
5.4. Anonyme Funktionen	45
5.5. Data Mining - Datenverarbeitung	46
5.6. Curry Form	50
5.7. Funktionen höherer Ordnung	52
5.8. Totalisierung	53
5.9. Polymorphie	54
5.10. Anwendungsbeispiel: Quicksort	55
5.11. Anwendungsbeispiel: Quicksort 2.0	56
5.12. Funktoren	56
6. Datentypen	56
6.1. Typisierung	57
6.2. Typprüfung	58
6.3. Typinferenz	58
6.4. Algebraische Datentypen	59
6.5. Algebraische Datentypen :: Produkttypen	60
6.6. Algebraische Datentypen :: Summentypen	64
6.7. Datentyp Some-or-None	66
6.8. Aufzählungstypen	67
6.9. Rekursive Algebraische Typen	67
6.10. Module	68
6.11. Typklassen	69
7. Abstrakte Datentypen	71
7.1. Abstrakte Datentypen	72
7.2. Wörterbuch	73
7.3. Hashtabellen	74
7.4. Arrays	78
7.5. Listen	82
7.6. Sortieren von Listen	84
7.7. Stack	90
7.8. Warteschlange (Queue)	91

7.9. Bäume	95
7.10. Graphen	99
8. Lost & Found	99
8.1. Laufzeiteigenschaften von Algorithmen	100
8.2. Landau-Symbole	101
8.3. Landau-Symbole	103
8.4. Laufzeitanalyse	103
8.5. Lazy Evaluation	106
8.6. Memoization	107
8.7. Kategorien und Funktoren	108
8.8. Monaden	109
8.9. Automaten als Monaden	113
9. Referenzen	114
9.1. Bücher	114
9.2. Literatur	114
9.3. Videos und WEB	114

2. Überblick

2.1. Schwerpunkte in diesem Kurs

- ▶ Grundlagen funktionaler Programmiersprachen
- ▶ Konzepte funktionaler Programmiersprachen
- ▶ Praktische Relevanz und Anwendung der funktionalen Programmierung
- ▶ Funktionale Programmiersprache Haskell

Begleitet von Übungen um obige Techniken konkret anzuwenden

Vorlesung

2 SWS mit Grundlagen und Live Programming mit einfachen Programmierübungen zum mitmachen (Notebook zur Vorlesung mitbringen!)

Übung

2 SWS mit Programmierung und angewandter Vertiefung

Voraussetzungen

Grundlegende Kenntnisse der Mathematik, Grundlegende Programmierfähigkeiten (in einer beliebigen Sprache)

2.2. Literatur

Vorlesungsskript

Die Inhalte der Vorlesung werden als navigierbarer Foliensatz und kompiliertes Skript bereitgestellt

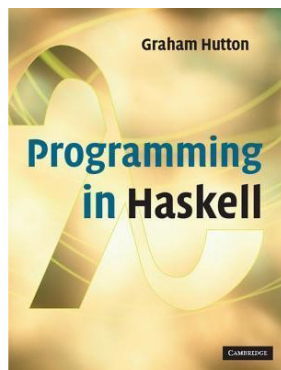
Funktionale Programmierung: in OPAL, ML, HASKELL und GOFER

Peter Pepper, Springer Berlin, 2003.



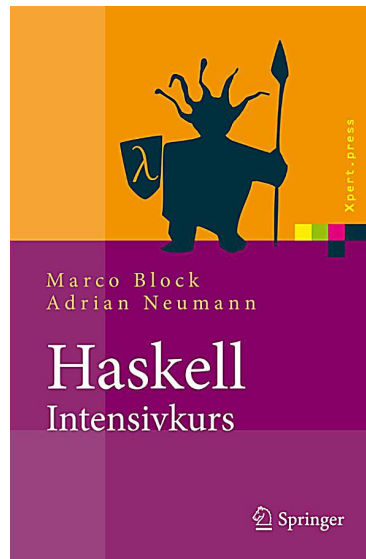
Programming in Haskell

Graham Hutton, University of Nottingham, Cambridge University Press, 2007



Haskell Intensivkurs

M. Block and A. Neumann, Springer, 2011



Haskell Programming Tutorial

Tutorials Point, www.tutorialspoint.com, 2017



2.3. Software

Verwendete Software (Vorlesung):

Haskell WEB Laboratory

Überblick

- Einfacher WEB basierter Haskell Compiler und Interpreter (REPL)
 - Ursprung: JSHC entstanden in einer Bachelor Arbeit von Stefan Björnesjö und Peter Holm (2011, Universität Göteborg)
- Benötigt wird nur ein WEB Browser und eine einzige Datei (*hslab.html*) → Kann auch offline mit lokaler Kopie ausgeführt werden!
- Enthält Editor, Interpreter Shell, und Online Hilfe
- Compiler übersetzt (Untermenge) von Haskell in JavaScript → Übersetztes Programm kann auch außerhalb des Labs ausgeführt werden!
- Integriertes WEB basiertes Clipboard zum Austausch von Code in der Vorlesung (Lehrer-Student-Student)!

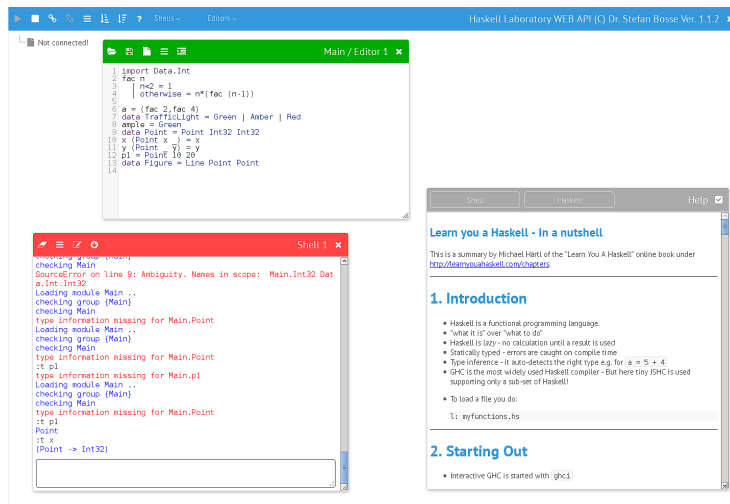
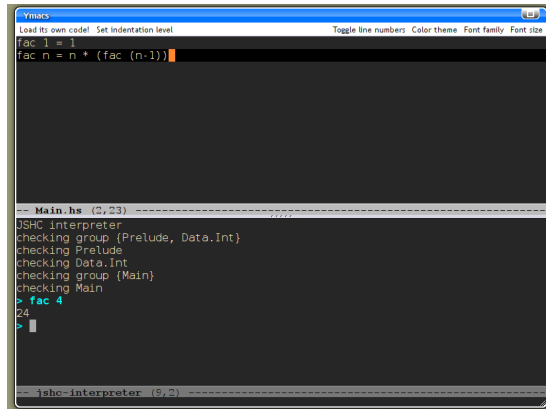


Abb. 1. Haskell WEB Laboratory in Aktion

Haskell to go

- Eine weitere Version (*hseddy.html*) basierend auf einem Emacs Editor ist auch für Smartphones und Tabletrechner geeignet → Begleitender Einsatz in der Vorlesung!



```
Ymscs
Load its own code! Set indentation level Toggle line numbers Color theme Font family Font size
fac 1 = 1
fac n = n * (fac (n-1))

----- Main.hs (2,23) -----
JSHC Interpreter
checking group (Prelude, Data.Int)
checking Prelude
checking Data.Int
checking group (Main)
checking Main
> fac 4
24
>
```

Verwendete Software (Vorlesung und Übung):

Glasgow Haskell Compiler <http://haskell.org/ghc/download.html>

Wird von der Kommandozeile aus benutzt:

```
$ ghci
GHCi, version 6.10.4:
http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
Prelude> let sq x = x*x
Prelude> sq 3
9
```

Leksah <http://leksah.org>

Grafische Entwicklungsumgebung für Haskell

```

626   })
627   } ++ extras
628
629 update :: (BuildInfo -> Int -> (BuildInfo -> BuildInfo)) -> Int -> (BuildInfo -> BuildInfo)
630 update bis index func =
631   map (\(bi,ind) -> if ind == index
632     map
633     mapAccum
634     mapAccumM
635     build
636     build
637     build
638     mapAccumM
639     mapAccumR
640     mapAccumWithKey
641     mapAccumWithKey
642     mapAdjust
643     mapAdjust
644     mapAllBuildInfo
645     mapAlter
646     mapAlter
647     mapAndRecoverM
648     mapAndUnzip
649     mapAndUnzip
650     mapAndUnzip3
651     mapAndUnzip3M
652     mapAndUnzip3M
653     mapAndUnzip3M
654     mapAndUnzip3M
655     mapAndUnzip3M
656     mapAndUnzip3M
657     mapArray
658
659

```

2.4. Ziele

1. Die Studierenden verstehen das Paradigma der funktionalen Programmierung und beherrschen eine entsprechende Programmiersprache wie etwa
 - Haskell
 - OCAML
 - (JavaScript, Lua)
2. Die Studierenden können einfache algorithmische Probleme und Datenstrukturen funktional modellieren und können dabei Funktionen höherer Ordnung und Typkonstruktoren wie etwa
 - Funktoren, Monaden und Monoiden einsetzen
3. Die Studierenden kennen typische Szenarien der funktionalen Programmierung etwa im
 - Bereich der Numerik oder im
 - Kontext der Daten-/Web- oder parallelen Programmierung.

2.5. Inhalte

1. Funktionen mit der **Mathematik als Ausgangspunkt** → *Mathematische Funktionen als "Programmiersprache"*

2. Einführung in die Programmierung mit Haskell
3. Mehr über Funktionen: Funktionen als Werte, Funktionen höherer Ordnung, Anonyme Funktionen, usw.
4. Typisierung
 - Wie kann man Daten strukturieren und strukturiert verwenden
 - Wie kann man Typen benutzen um Randbedingungen zu beschreiben, z.B., welche Art von Daten können mit einer Funktion verarbeitet werden
 - Wie kann man Daten unterscheidbar machen (Im Programm für den Übersetzer und zur Laufzeit)
 - Wie kann man Typen konstruieren - alles nur eine Funktion?
5. Anwendung der funktionalen Programmierung

2.6. Übung

Mo. 16:00 s.t. in M 201
Umfang: 2SWS
Marcel Heinz

Ablauf der Übung:

- Besprechung der Lösungen zu den letzten Assignments
- Vorbereitung auf die neuen Assignments mit Live-Programmierung

Formalitäten:

- Es gibt 9-12 Assignments-
- Abgaben werden in SVN hochgeladen.
- Vorschlag bzgl. der Zulassung: siehe "Exam admission rules" hier: <http://www.softlang.org/course:fp17>
 - ❑ 3/4 der Abgaben müssen mindestens eine Bewertung von 1 Punkt bekommen.
 - ❑ 1/2 der Abgaben müssen eine Bewertung von 2 Punkten bekommen.
- Alles, was in der Übung programmiert wird, wird hoch geladen.

2.7. Geschichte

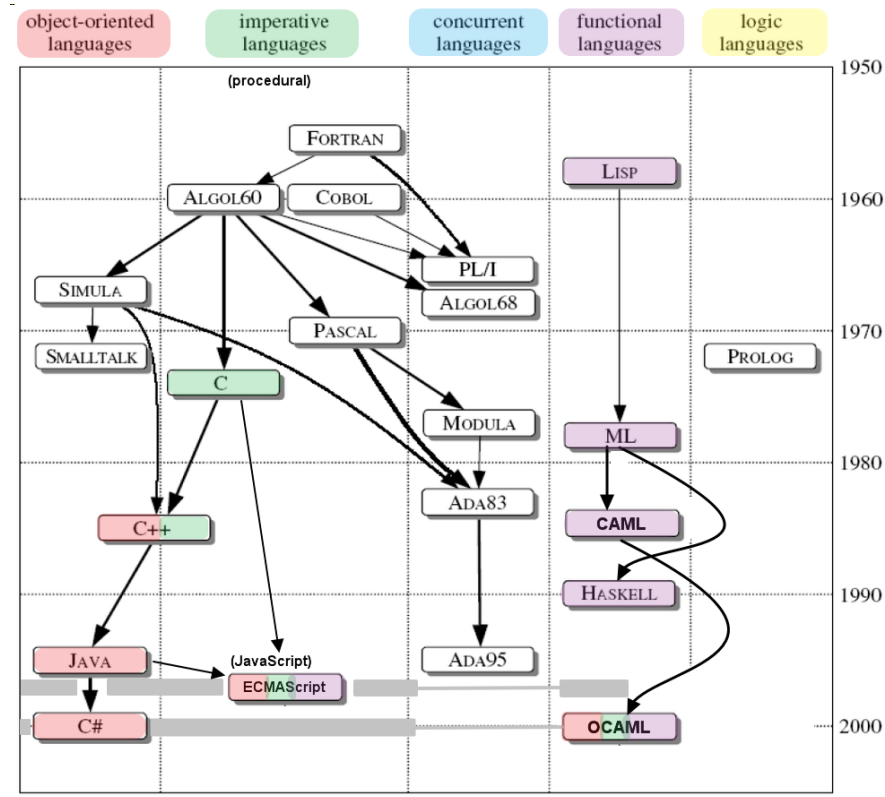


Abb. 2. Historische Entwicklung der Programmiersprachen und deren Klassifizierung

2.8. Übersicht der Programmiersprachen

Imperative Sprachen

Folge von nacheinander ausgeführten Anweisungen;
Spezifikation wie berechnet werden soll

► Prozedurale Sprachen

- Variablen, Zuweisungen, Kontrollstrukturen

➤ **Objektorientierte Sprachen**

- ❑ Objekte und Klassen
- ❑ ADT und Vererbung

Deklarative Sprachen

*Spezifikation dessen, was berechnet werden soll;
Compiler legt fest, wie Berechnung verläuft*

➤ **Funktionale Sprachen**

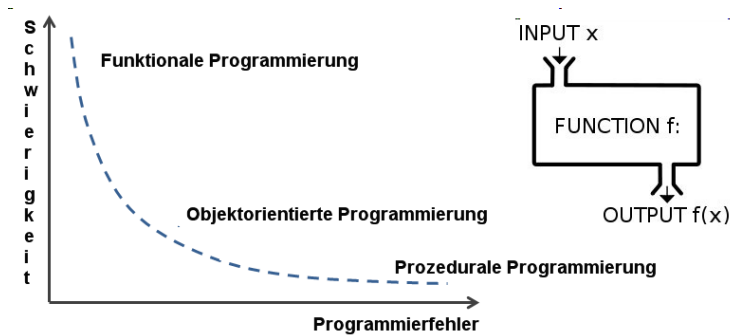
- ❑ Symbolische Variablen, keine Seiteneffekte, Typen
- ❑ Funktionen und Rekursion

➤ **Logische Sprachen**

- ❑ Regeln zur Definition von Relationen

2.9. Funktionale Programmierung - Warum?

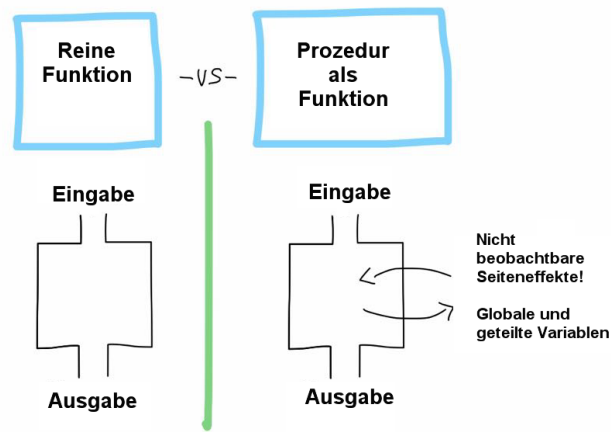
- Funktionale Programmierung kann schwieriger zu Erlernen sein als prozedurale und objektorientierte Programmierung
- Aber: Mit funktionaler Programmierung lassen sich Programmierfehler reduzieren



➤ Probleme in der prozeduralen/imperativen Programmierung:

- ❑ Nicht beobachtbare Seiteneffekte durch globale/geteilte Variablen

- Prozeduren statt reine Funktionen



2.10. Kenntnis verschiedener Sprachen

Eigene Ideen bei der Software-Entwicklung können besser und effizienter ausgedrückt werden

Nötig, um in konkreten Projekten geeignete Sprache auszuwählen

Erleichtert das Erlernen weiterer Programmiersprachen

Grundlage für den Entwurf neuer Programmiersprachen

2.11. Funktionale Programmiersprachen

- Haskell: Reine funktionale Programmiersprache
- OCaml: Funktionale Programmiersprache kombiniert mit prozeduraler und objektorientierter Programmierung
- JavaScript: Prozedurale Programmiersprache kombiniert mit objektorientierter und funktionaler Programmierung
- Lua: Prozedurale Programmiersprache kombiniert mit objektorientierter und funktionaler Programmierung

Primär wird in diesem Kurs Haskell gelehrt und in den Übungen eingesetzt. Es werden aber immer wieder Analogien zu den drei weiter aufgelisteten Sprachen gezeigt.

2.12. Funktionale Programmierung - Geschichte

1930



Alonzo Church entwickelte den Lambda Calculus, eine einfache und ausdrucksstarke Theory und Notation von Funktionen

1950



John McCarthy entwickelte Lisp, die erste Funktionale Sprache, mit etwas Einfluß vom Lambda Calculus, aber noch mit Speichervariablen und Zuweisungen

1970



Robin Milner und andere entwickelten ML, die erste moderne Funktionale Sprache mit Typinferenz und polymorphen Typen

1987



Ein internationales Komitee von Forschern starteten die Entwicklung von Haskell, eine standardisierte Funktionale Sprache mit Bedarfsauswertung (lazy evaluation)

2.13. Funktionale Programmierung - Haskell

- Haskell-Interpreter `ghci`
 - ❑ Eingabe: Auszuwertender Ausdruck
 - ❑ Ausgabe: Ergebnis des ausgewerteten Ausdrucks
- Bsp:
 - ❑ Eingabe: `sum [15, 36, 70]`
 - ❑ Ausgabe: `121`

Interpreter

- Führen jede Zeile des Programms nacheinander aus.
- **Vorteil:** Keine vorherige Übersetzung nötig, gut zur Programmentwicklung geeignet.
- Schnelles Testen und steile Lernkurve
- **Nachteil:** Ineffizienter als Compiler
- Ausnahme: Interpreter mit Just-in-Time Übersetzern (z.B. JavaScript und V8 Maschine)

2.14. Interpreter

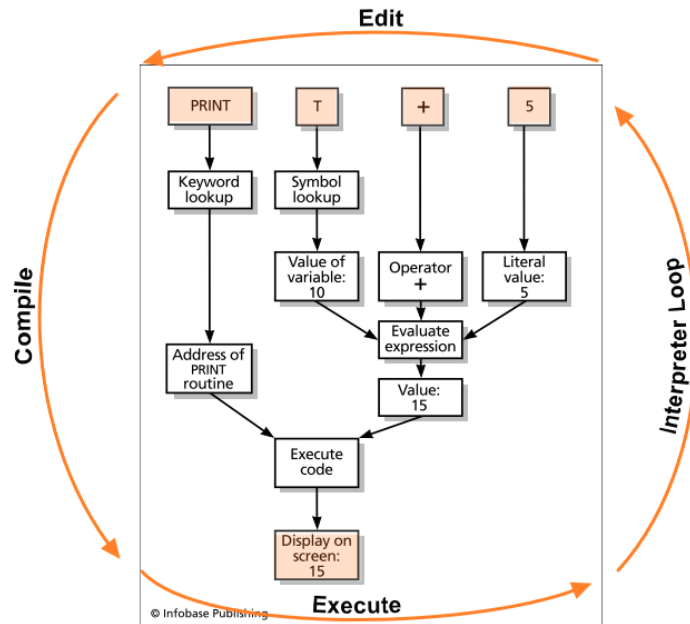


Abb. 3. Interpreter Zyklus: Editieren → Übersetzen → Ausführen

2.15. Just-in-Time Compiler

- Interpreter können im wesentlichen auf drei Arten (Architekturklassen) implementiert werden:
 1. Direkte Ausführung des Quelltextes (die Nutzereingabe und bereits geschriebene Skripte) (*Parse* → *Execute*)
 2. Virtuelle Maschine und Übersetzung des Quelltextes in eine Zwischenrepräsentation die von einer virtuellen Maschine ausgeführt werden kann → Bytecode
 3. Virtuelle Maschine mit Bytecode Übersetzung, Ausführung des Bytecodes, und ausgewählter Übersetzung des Bytecodes in nativen Maschinencode → JIT Compiler!

Beispiele

- Python: Klasse 2 (Bytecode)

- JavaScript: Klasse 2 (Spidermonkey, WEB Browser) und Klasse 3 (Google Chrome/V8, nodejs)
- OCaML: Klasse 2 (und native Codeerzeugung mit Compiler)
- Lua: Klasse 2 (Lua) und Klasse 3 (LuaJit)

2.16. Haskell WEB Labor

- Die Datei *hslab.html* enthält das Haskell Labor bestehend aus den folgenden Komponenten:
 - ❑ Haskell Editor mit Syntaxkontrolle
 - ❑ Interpreter Shell
 - ❑ JSHC Haskell → JavaScript Compiler
 - ❑ Ein WEB Clipboard zum Austausch von Programmcode zwischen Lehrer und Studenten (und untereinander)
- JSHC unterstützt nur eine Untermenge von Haskell und ersetzt daher GHC nicht!
- Es gibt leichte syntaktische Abweichungen zu Glasgow Haskell (wird darauf hingewiesen)
- Die Typsignaturen von JSHC haben etwas andere Notation als bei Glasgow Haskell
- Das Haskell WEB Labor ist experimentell! Daher unterliegt es fortlaufender Aktualisierung

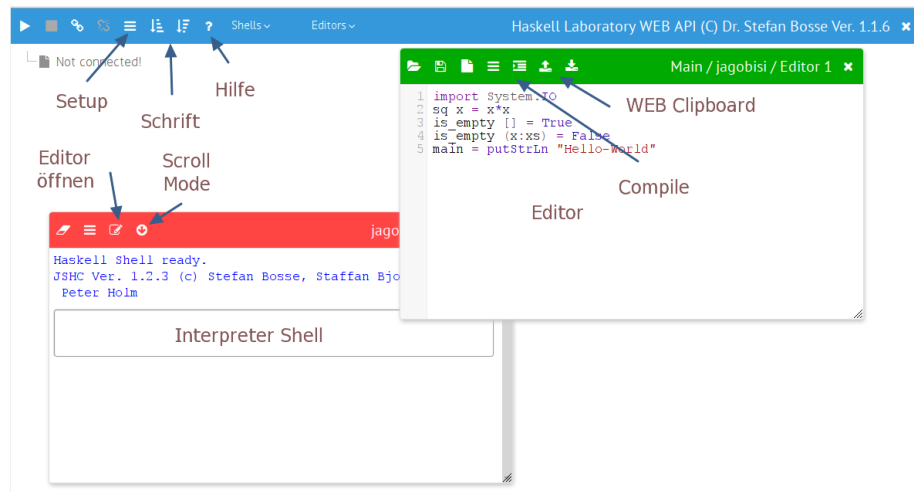


Abb. 4. Haskell WEB Labor GUI

3. Einführung in die Funktionale Programmierung

3.1. Mathematik

- Mathematik ist Funktionale Programmierung ;-)
- Mathematische “Methoden”: **Funktionen**, **Ausdrücke** (Terme), symbolische **Variable**, **Werte!**
- Mathematische Funktionen bestehen aus Termen und berechnen ein Ergebnis
- Mathematische Funktionen sind zeit- und zustandslos

Die Ergebnisse einer Berechnung hängen nur von den aktuellen Werten und nicht von Berechnungen aus der Vergangenheit ab → Kein Speicher!

- Eine symbolische Variable ist daher nur eine Ausdruckssubstituierung

- Mathematische Funktionen beschreiben Zusammenhänge und nicht im Detail die Berechnung:

Was und nicht Wie!

3.2. Terme

- Terme sind Ausdrücke und bestehen aus:
 - ❑ **Werten** (Konstanten) 1 1.0 3.14 (-1,1) ..
 - ❑ **Symbolischen Variablen** (Literale) a b c ..
 - ❑ **Operationen** + - * / mod ..
 - ❑ **Konditionalen Ausdrücken**

$$a = 117, b = \frac{a+1}{a-1}, c = \begin{cases} 1, & a < 100 \\ 2, & a \geq 100 \end{cases}$$

3.3. Zusammengesetzte Werte

Vektoren

- **Bindung** von Werten des **gleichen Typs** (z.B. aus \mathbb{N})
- Einzelne Elemente können durch einen ganzzahligen Index ausgewählt werden

Tupel

- Ein Tupel ist eine **Bindung von verschiedenen Werten u.U. unterschiedlichen Typs**
- Notation eines n-stelligen Tupels: $\langle v_1, v_2, v_3, \dots, v_n \rangle$
- Beispiel komplexe Zahlen: $\langle \text{real}, \text{complex} \rangle : \dots$
- Beispiel mehrsortiges Tupel: $\langle \text{True}, 1 \rangle \in \mathbb{B} \times \mathbb{N}$
- Anders als bei einem Vektor kann man einzelne Werte nicht direkt indizieren
- Man benötigt Musteranpassung um einzelne Elemente eines Tupels lesen zu können (wird später eingeführt, oder vordefinierte Funktionen aus der *Prelude* Bibliothek)

Beispiele Tupel**3.4. Typen**

- Werte, Variablen, und Funktionen können ganz verschiedene Werttypen (Datentypen) besitzen und verarbeiten

\mathbb{I}

Die Menge aller ganzen Zahlen - unendlich groß aber abzählbar! $\{\dots, -2, -1, 0, 1, 2, \dots\}$

\mathbb{N}

Die Menge aller natürlichen Zahlen - unendlich groß aber abzählbar!

\mathbb{R}

Die Menge aller reeller Zahlen - unendlich groß und *nicht* abzählbar! $\{\dots, 3.14, \dots\}$

\mathbb{B}

Boolesche Werte $\{0, 1\}$ oder $\{\text{False}, \text{True}\}$

\mathbb{C}

Die Menge aller Textzeichen (alphanumerisch) $\{\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \dots\}$

- In Programmiersprachen ist häufig die Zuordnung eines Wertes zu einem Typ nicht unmittelbar eindeutig, z.B. $0 \in \mathbb{R}, \mathbb{N}, \mathbb{I}$

3.5. Funktionen

- Eine Funktion bildet Werte (Argumente) von Eingabevariablen a, b, c, \dots (Parametern) auf Werte durch Ausgabevariablen x, y, z, \dots (Ergebnisse) ab, d.h. sie führt eine Berechnung durch:

$$f(a, b, c, \dots) : (a, b, c, \dots) \rightarrow (x, y, z, \dots)$$

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

- Man unterscheidet:
 - ❑ **Funktionsdefinition** (Algorithmus)
 - ❑ **Funktionsdeklaration** (Typisierung)

□ **Funktionsapplikation** (Anwendung und Evaluierung einer Funktion)

$$\begin{aligned}sq(x) &= x * x \\sq(x) &: \mathbb{R} \rightarrow \mathbb{R} \\a &= sq(2) + 1\end{aligned}$$

Formal

Def. Funktion: Eine Funktion f ist ein Tripel $\langle \mathbb{D}_f, \mathbb{W}_f, \mathbb{R}_f \rangle$ aus drei Mengen: Definitionsmenge \mathbb{D}_f , Wertemenge \mathbb{W}_f , und einer Relation \mathbb{R}_f

Definitionsmenge \mathbb{D}_f

Die Menge aller Eingabewerte. Z.B. die Additionsfunktion (+) kann die Definitionsmenge $\mathbb{D}_f = \mathbb{N} \times \mathbb{N} = \{, , ..\}$ besitzen.

Wertemenge \mathbb{W}_f

Die Menge aller Ausgabewerte (alle möglichen Berechnungsergebnisse). Z.B. die Additionsfunktion (+) könnte die Wertemenge $\mathbb{W}_f = \mathbb{N} = \{0, 1, 2, ..\}$ besitzen.

Relation \mathbb{R}_f

Die Relation verknüpft Eingabewerte mit Ausgabewerten. Im Beispiel der Addition wäre dies $\mathbb{R}_f = \{ \langle ,0 \rangle, .. , \langle ,5 \rangle .. \}$

Diagramme

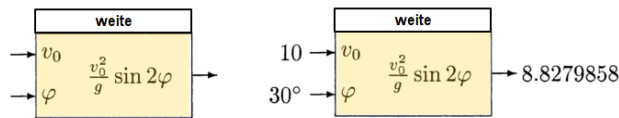
- Funktionen kann man als **Blöcke** in **Diagrammnotation** darstellen
- Jeder Funktionsblock hat Eingangsports (die Funktionsparameter) und ein oder mehrere Ausgangsports (Ergebnisvariablen)

Beispiel: Wurfweite

- Mathematisch:

$$w = \frac{v_0^2}{g} \sin 2\varphi$$

- Block:



Links: Funktionsdefinition, Rechts: Funktionsanwendung mit konkreten Werten

Beispiele Funktionen

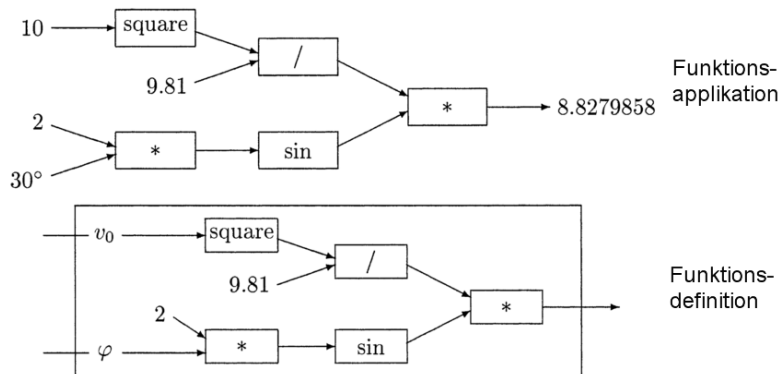
Aufgabe

1. Momentan gibt es nur Ganzzahlarithmetik (*Int32*). Im vorherigen Beispiel war die Sinusfunktion nicht implementiert. Verändere die Funktion *sin* und *w* so dass eine Näherung eines Sinus im Wertebereich $\phi = [0, 90]$ Grad einen Wert von $[0, 1000]$ ausgibt. Eine Skalierung in *w* ist dann erforderlich.

- Beachte Rundungsfehler der Ganzzahlarithmetik
- benutze das Haskell WEB Labor

3.6. Funktionen und Datenfluss

- Das Blockdiagramm ist **hierarchisch**. Die Implementierung einer Funktion wird durch Unterdiagramme spezifiziert (Funktionale Komposition).
- Die Auswertung von Ausdrücken kann man sich leicht als **Datenfluss** durch ein Netz von Funktionskästen (Blöcken) vorstellen.
- Zum Beispiel kann die obige Formel zur Berechnung der Wurfweite folgendermaßen visualisiert werden (wobei hier der Fluss nicht von oben nach unten, sondern von links nach rechts läuft)



3.7. Funktionsrekursion

- Rekursion ist der **Selbstauf** einer Funktion
- Beispiele: Fakultät und Fibonacci

$$fac(n) = \begin{cases} 1 & , n = 0 \\ n \cdot fac(n-1) & , n > 0 \end{cases} \quad fib(n) = \begin{cases} n & , n < 2 \\ fib(n-2) + fib(n-1) & , n > 0 \end{cases}$$

- Was ist zu beachten?

Terminierung

- Eine Rekursion benötigt ein **Terminierungskriterium!**
- Gefahr: *Endlosrekursion*
- Zwei ähnliche Beispiele:

$$sum_1(x) = \begin{cases} 0, & x \leq 0 \\ x + sum_1(x-1), & x > 0 \end{cases}$$

$$sum_2(x) = \begin{cases} 0, & x = 0 \\ x + sum_2(x-2), & x \neq 0 \end{cases}$$

Aufgabe

- Wo ist das Terminierungskriterium definiert?
- Terminieren beiden Funktionen garantiert???
- Was passiert technisch bei Endlosrekursion?
- Man unterscheidet zwei Arten der Rekursion, die erhebliche Auswirkungen zur Laufzeit und auf die Programmausführung haben können:
 - ❑ In vielen Programmiersprachen werden Funktionsaufrufe mittels eines Stackspeichers realisiert.
 - ❑ Bei jedem Funktionsaufruf wird ein Satz "neuer" Funktionsparameter auf dem Stack erzeugt → Gefahr: Stack Überlauf!!!

End (Tail) Rekursion

- Rekursionsaufruf einer Funktion am Ende der Funktion bzw. des Ausdrucks ohne weitere Berechnung

- Vorteil dieser Funktionsdefinition ist, dass kein zusätzlicher Speicherplatz zur Verwaltung der Rekursion benötigt wird

Kopf (Head) Rekursion

- Rekursionsaufruf einer Funktion am Anfang oder in der Mitte der Funktion bzw. des Ausdrucks mit nachfolgender Berechnung.
- Jede Kopfrekursion kann durch eine funktionale Transformation in eine Endrekursion überführt werden
- Folgendes Beispiel zeigt den Unterschied zwischen Kopf- und Endrekursion und den Einfluss von Akkumulatoren für Zwischenergebnisse

Kopfrekursion (!)

$$\text{sum}(n) = \begin{cases} 0, n = 0 \\ n + \text{sum}(n - 1), n > 0 \end{cases}$$

- Die letzte Berechnung die durchgeführt wird ist die Addition (+)!
- Solange wird der Funktionsparameter n noch benötigt!

Endrekursion

$$\begin{aligned} \text{sum}(n) &= \text{addsum}(0, n) \\ \text{addsum}(m, n) &= \begin{cases} m, n = 0 \\ \text{addsum}(m + n, n - 1), n > 0 \end{cases} \end{aligned}$$

- Das Laufzeitverhalten unterscheidet sich grundsätzlich (Stack):

Kopfrekursion

<code>sum(3) = 3 + sum(2)</code>	Stack <code>[n₁=3]</code>
<code>sum(2) = 2 + sum(1)</code>	Stack <code>[n₁=3, n₂=2]</code>
<code>sum(1) = 1 + sum(0)</code>	Stack <code>[n₁=3, n₂=2, n₃=1]</code>
<code>sum(0) = 0</code>	Stack <code>[n₁=3, n₂=2, n₃=1, n₄=0]</code>
<code>sum(1) = 1 + 0 = 1</code>	
<code>sum(2) = 2 + 1 = 3</code>	
<code>sum(3) = 3 + 3 = 6</code>	

Endrekursion

```

sum(3) = add_sum(0, 3)   Stack [m=0,n=3]
       = add_sum(3, 2)   Stack [m=3,n=2]
       = add_sum(5, 1)   Stack [m=5,n=1]
       = add_sum(6, 0)   Stack [m=6,n=0]
       = 6

```

- Kopfrekursion hat progressiven Speicherbedarf
- Endrekursion hat konstanten Speicherbedarf

Jede (End)rekursion kann in eine bedingte Schleife (while do) mit konstanten Speicherbedarf transformiert werden

```

sum=0;
while (n>0) do
  sum := sum + n;
  n := n -1;
done

```

Die Praxis und Realität :-)

Haskell

JavaScript

Die Praxis und Realität :-)

OCAML

3.8. Funktionale Komposition**Funktionsdefinition**

Die Zuordnung eines *Funktionsnamens* und benannten *Funktionsparametern* zu einer Berechnung mit einem Ausdruck: $f(p_1, p_2, \dots) = e(p_1, p_2, \dots)$. Die Funktionsparameter sind nur im *Funktionskörper* nutzbar und sichtbar.

Funktionsapplikation

Die Anwendung einer Funktion i.A. in Ausdrücken, d.h. die Berechnung mit konkreten Werten: $f(\epsilon_1, \epsilon_2, \dots)$, wobei ϵ ein beliebiger Wert oder Ausdruck sein kann.

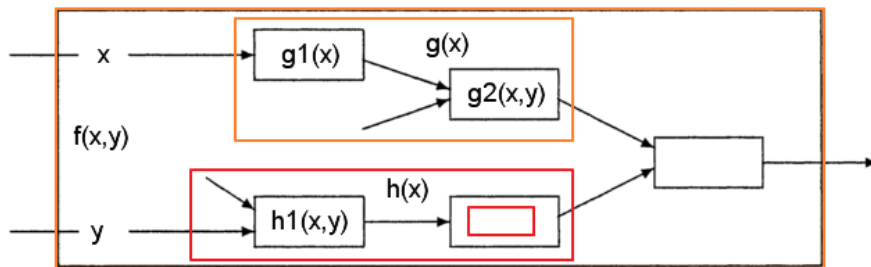
Funktionskomposition

Die Zusammensetzung neuer Funktionen aus bereits definierten Funktionen:

$$\begin{array}{ll} f(x) = \epsilon(x), & sq(x) = x * x \\ g(x) = \epsilon(x), & pre(x) = x - 1 \\ h(x) = \epsilon(f, g, x) = f(x) \circ g(x) & presq(x) = sq(pre(x)) \end{array}$$

Definition, Applikation, Komposition, und Rekursion sind elementare Eigenschaften der Programmierung und von Programmiermodellen!

- Die Funktionsdefinition und Applikation erlaubt die **Zerlegung von Programmen** in kleine wiederverwendbare Einheiten!
- Die (Funktionale) Komposition ist *hierarchisch* und kann durch Blockdiagramme graphisch dargestellt werden:

**3.9. Funktionale Programmierung**

Die funktionale Programmierung ist ein Programmierparadigma welches die Berechnung als Evaluierung von mathematischen Funktionen behandelt und Zustand mit veränderlichen Speicher vermeidet!

- Funktionale Programmierung hat seinen Ursprung im **Lambda Calculus** (1930), einer formalen Methode
- Es gibt eine Vielzahl von funktionalen Programmiersprachen:

ML

Meta Language (Standard)

Haskell

Rein funktionale Programmiersprache

OCAML

Basierend auf ML, aber mit zusätzlichen prozeduralen/imperativen und objektorientierten Programmierparadigmen

LISP

Eine der ersten funktionalen Sprachen?

Programmierparadigma funktionaler und deklarativer Sprachen

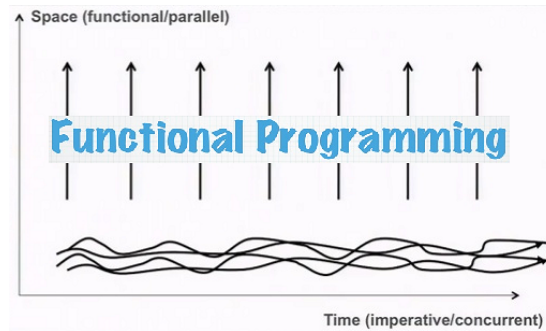
Spezifiziere was berechnet werden soll, nicht wie es berechnet werden soll.

- Imperative Sprachen (C, JAVA, ..) starten mit low-level Code (z.B. Assembler) und führen Abstraktionen ein um die Programmierung zu vereinfachen:
 - ❑ Schleifen
 - ❑ Funktionen (!)
- Imperative Sprachen geben die Reihenfolge von Anweisungen vor!
- Deklarative Sprachen sind durch die Sprache der Mathematik motiviert und beschreiben Funktionen oder Beziehungen
- Die charakteristische Eigenschaft deklarativer Sprachen ist die **Church-Rosser Eigenschaft**: *Intuitiv besagt sie, dass die Reihenfolge der Auswertung unerheblich für das Resultat ist.*

Vorteile deklarativer und funktionaler Sprachen

- Korrektheitsbeweise von Programmen sind relativ einfach → Terminierung!
- Programme sind meist sehr viel kürzer als z.B. in imperativen Sprachen.

- Erste Prototypen des Codes können schnell erstellt werden (“rapid prototyping”).
- Es gibt keine inhärent sequentielle Auswertung → **Parallelisierung** !



Nachteile deklarativer und funktionaler Sprachen

- Es gibt keine Möglichkeit den Ressourcenverbrauch zu steuern und zur Programmierzeit abzuschätzen
- Es gibt weniger Werkzeuge für den Programmentwurf
- Ein- und Ausgabe (auch Netzwerkkommunikation) ist (evtl. strikt) sequenziell!
 - ❑ IO erfordert Erweiterung des funktionalen Programmiermodells (z.B. mit Monaden) um Sequenz und Speicher zu implementieren!

4. Einführung in Haskell

4.1. Haskell

- Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.
- Die wichtigsten Konzepte in Haskell sind:
 - ❑ Rein funktionale Semantik (d.h. keine Seiteneffekte)
 - ❑ Algebraische Datenstrukturen mit mächtigen Operationen wie “pattern matching”, “list comprehensions”

- ❑ Funktionen höherer Ordnung (“higher-order functions”)
- ❑ Statisches Typensystem
- ❑ Bedarfsauswertung (“lazy evaluation”)
- Neben Haskell sollen in diesem Kurs aber auch einige andere funktionale Sprachen beleuchtet werden: ML, JavaScript, Lua (teils,!)
- **Da es auch in Haskell nicht alle Konzepte der funktionalen Programmierung gibt wird eine allgemein verständliche Pseudonotation verwendet** (siehe [A])

4.2. Haskell - Basistypen

- Werte (Konstanten und symbolische Variablen) haben einen bestimmten Datentyp.

Basistypen in Haskell

Bool

Boolesche (logische) Werte: `True` und `False`

Char

Zeichenkette `String` \Leftrightarrow Zeichenfolgen sind Listen von Zeichen, d.h. `[Char]`

Int

Ganze Zahlen (“fixed precision integers”)

Integer

Ganze Zahlen beliebiger Länge (“arbitrary precision integers”)

Float

Fließkommazahlen (“single-precision floating point numbers”)

4.3. Haskell - Terme

- Ausdrücke werden in Pseudonotation allgemein mit ϵ bezeichnet
- Ausdrücke bestehen aus (symbolischen) Variablen, Werten, Funktionsapplikationen, und Operatoren

Arithmetische Ausdrücke

`+` , `-` , `*` , `/` , `**`

Boolesche Ausdrücke

`&&` , `||` , `not`

Relationale Ausdrücke

`==` , `>` , `<` , `<=` , `>=` , `/=`

4.4. Haskell - Symbolische Variablen

- Werte und Terme können **symbolischen (unveränderlichen) Variablen** zugewiesen und in folgenden Ausdrücken wieder verwendet werden (Substitution):

```
x =  $\epsilon$ 
let x =  $\epsilon_1$  in  $\epsilon_2$ 
```

- `let in` wird innerhalb von Ausdrücken verwendet und ist nur **innerhalb des Ausdrucks und nach der Definition** sichtbar

4.5. Haskell - Bedingte Ausdrücke

Bedingte Ausdrücke

Definition 1.

```
f x = if cond- $\epsilon_1^x$  then  $\epsilon_1$  else
      if cond- $\epsilon_2^x$  then  $\epsilon_2$  else
      ...
      else  $\epsilon_0$ 
```

Bewachte Ausdrücke

Definition 2.

```
f x | cond- $\epsilon_1^x$  =  $\epsilon_1$ 
    | cond- $\epsilon_2^x$  =  $\epsilon_2$ 
    ...
    | otherwise =  $\epsilon_0$ 
```

Beispiel: Entscheidungsbäume

- Entscheidungsbäume werden im Maschinellen Lernen (ML) eingesetzt. Zweck des ML ist u.A. das Lernen von Zusammenhängen (Modellfunktionen) mit Trainingsdaten.
- Das Ergebnis vom überwachten Lernen ist eine Funktion die einen Satz von Eingabevariablen x_1, x_2, \dots, x_n auf Ausgabevariablen y_1, y_2, \dots, y_m abbildet:

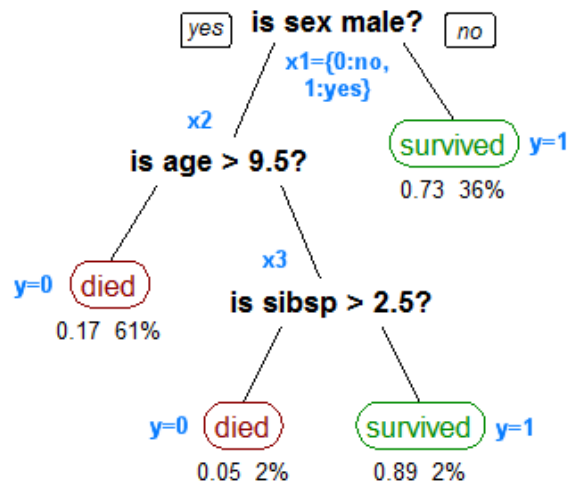
$$\text{ML } \textit{learn}(x_1, \dots, x_n, y_1, \dots, y_m): \mathbb{R}^{n+m} \rightarrow f(x_1, \dots, x_n)$$
$$\text{Model } f(x_1, \dots, x_n): \mathbb{R}^n \rightarrow \mathbb{R}^m$$

Beispiel: Überlebenswahrscheinlichkeit Titanik Unfall

- Eingabevariablen: x_1 :Geschlecht, x_2 :Alter, x_3 :Anzahl Verwandte
- Ausgabevariable: y :Überleben?

Entscheidungsbaum

- Ein mögliches ML Modell der die Funktion $f()$ approximiert ist der Entscheidungsbaum:
 - Abgeleitet aus Protokollen des Unglücks und Statistiken



Mathematisches Modell

$$f(x_1, x_2, x_3) = \begin{cases} \begin{cases} 1, & \text{wenn } x_3 > 2.5 \\ 0, & \text{wenn } x_3 \leq 2.5 \end{cases} \\ \begin{cases} 0, & \text{wenn } x_2 \leq 9.5 \\ 1, & \text{wenn } x_1 = 0 \end{cases} \end{cases}$$

Haskell

4.6. Haskell - Listen und Tupel

Listen: Definition

- Listen werden in Haskell statisch durch eine Aufzählung von Werten (oder Ausdrücken) mit Kommatrennung in eckigen Klammerpaaren erzeugt:

Definition 3.

```
[ v1, v2, v3, .. ]
```

- Die Listenelemente sind eindeutig wie bei einem Array durch einen Index $\{0, 1, 2, \dots, n-1\}$ referenzierbar.

Tupel: Definition

- Tupel werden in Haskell statisch durch eine Aufzählung von Werten (oder Ausdrücken) mit Kommatrennung in runden Klammerpaaren erzeugt:

Definition 4.

```
( v1, v2, v3, .. )
```

Listen: Zugriff

- Auf einzelne Listenelemente oder Bereiche kann mit verschiedenen eingebauten Funktionen zugegriffen werden:

```
FUN head: [α] → α
FUN last: [α] → α
FUN tail: [α] → [α]
FUN drop: Int → [α] → [α]
FUN take: Int → [α] → [α]
FUN (!!): [α] → Int → α
```

Listen: Komposition

- Listen können zur Laufzeit neu zusammengesetzt werden:

```
FUN (++): [α] → [α] → [α]
FUN reverse: [α] → [α]
```

Listen: Mathematische Mengen

$$\{x^2 \mid x \in \{1..5\}\}$$

```
[x^2 | x <- [1..5]]
```

Tupel: Elementzugriff und Zerlegung

- Auf einzelne Tuplelemente oder Bereiche kann *nicht direkt* zugegriffen werden.
- Stattdessen muss **Musteranpassung** oder Zerlegung verwendet werden, d.h., im einfachsten Fall wird eine Ausdruckszuweisung verwendet mit einem Variablentupel auf der linken Seite und dem zu zerlegenden Tupel:

Definition 5.

```
tuple = ( v1, v2, .. , vn )
( x1, x2, .. , xn ) = tuple
let ( x1, x2, .. , xn ) = tuple
f ( x1, x2, .. , xn ) = ..
fst (x,_) = x
snd (_,y) = y
```

- Wenn ein Element eines Tupels nicht verwendet wird, kann ein Platzhalter `_` eingefügt werden (wildcard)
- Tupelzerlegung kann auch bei der Definition von Funktionsparametern angewendet werden

4.7. Haskell - Mehrfachauswahl und Musteranpassung

- Wie in allen gängigen Programmiersprachen gibt es auch in Haskell eine Mehrfachauswahl.
- Die Mehrfachauswahl ist aber deutlich ausdrucksstärker als z.B. switch-case in C
- Die Mehrfachauswahl ist ein *Ausdruck* der einen Wert berechnet!

Definition 6.

```

case  $\epsilon$  of {
   $v_1 \rightarrow \epsilon_1$  ;
   $v_2 \rightarrow \epsilon_2$  ;
  ..
   $v_n \rightarrow \epsilon_n$  ;
  OTHERWISE
   $x \rightarrow \epsilon_0$  ; ODER
   $_ \rightarrow \epsilon_0$ 
}

```

Ein Case-Ausdruck muss mindestens eine Alternative haben und jede Alternative muss mindestens einen Körper haben. Jeder Körper muss denselben Typ haben, und der Typ des gesamten Ausdrucks ist dieser Typ.

- Werte v_i der Mehrfachauswahl können skalare Werte, Tupel, und Listenmuster (z.B. $x : xs$) enthalten!

Beispiel

```

farbe = "rot"
ist_rot col = case col of {
  "rot" -> True ;
  _ -> False
}
ist_mischung col = case col of {
  "rot" -> False ; "blau" -> False ; "gruen" -> False ;
  _ -> True
}

```

Tupelzerlegung

- Mit der Mehrfachauswahl können auch zusammengesetzte Werte (Tupel) zerlegt werden → **Musteranpassung**

Definition 7.

```

case tuple-ε of {
  (v1, v2, ..) -> ε1 ;
  (v1, x2, ..) -> ε1 ;
  (x1, x2, ..) -> ε1 ;
}

```

Aufgabe

1. Es soll eine Funktion *disto* in Haskell und dem Haskell Laboratory implementiert werden, die die Distanz zwischen zwei Punkten p_1 und p_2 im kartesischen Koordinatensystem berechnet.

► Punkte werden durch ein Tupel (x,y) repräsentiert.

$$\text{disto}(p_1, p_2) = \sqrt{((p_{1,x} - p_{2,x})^2 + (p_{1,y} - p_{2,y})^2)}$$

2. Implementiere eine *iterative* Näherung der Wurzelfunktion *sqrt* für ganze Zahlen

4.8. Haskell - Funktionen

Definition

DEF $f(x, y, z, \dots) = \epsilon(x, y, z) \Leftrightarrow$ DEF $f\ x\ y\ z = \epsilon(x, y, z)$

In vielen Programmiersprachen werden die Funktionsparameter x, y, z, \dots in Klammern und durch Komma getrennt definiert. Aber in ML/Haskell ist (x, y, z, \dots) ein Parameter (ein Tupel)! Geht auch!

```

f x y z .. = εx,y,z,..
f (x,y,z,..) = εx,y,z,..

```

Applikation

Definierte (oder eingebaute) Funktionen können durch Angabe des Namens und folgende Argumente für die Parameter evaluiert werden.

Typsignatur

FUN $f : \text{typ}_1 \rightarrow \text{typ}_2 \dots \rightarrow \text{typ}_r \Leftrightarrow$
 FUN $f : \text{typ}_1 \times \text{typ}_2 \times \dots \rightarrow \text{typ}_r$

Die Typsignatur einer Funktion (oder einer Variable) gibt die Datentypen der Funktionsparameter und des Rückgabewertes an. Kann mittels der `:type` Funktion angezeigt werden.

Überladung und Polymorphie

Wenn eine Funktion, z.B. Addition, für Werte von verschiedenen Typen anwendbar ist, wird eine Typklasse angegeben. Beispiel: `:type (+) → (+) :: Num a => a -> a -> a` mit `Num` als Typklasse aller numerischen Typen (`ℕ ℝ ..`)

Beispiel: Berechnung der Wurfweite

Eingebaute Funktionen (Prelude)

Arithmetische, Boolesche, Logische Operation, Mathematische

Gehören zu den eingebauten Funktionen, häufig überladen/polymorph definiert.

Listen

Viele Funktionen wie die Summationsfunktion `sum` des Basismoduls `Prelude` arbeiten mit Listen, d.h., einsortige Listen von Elementen:
`[v1, v2, v3, ..]`

Funktionskomposition

`FUN f1, FUN f2, .., DEF fN x y z .. = f1 ∘ f2 ∘ ..`

Verwendung von Funktionen in Funktionen oder Definition von Funktionen und Verwendung in Funktionen!

Muster mit Funktionen

- Klassisch ist eine Funktionsdefinition monolithisch, d.h. es existiert eine Definition mit:
 - ❑ Namen, Parametersatz, Körper (Ausdruck)
 - ❑ Typsignatur
- In der funktionalen Programmierung können Funktionen aber durch mehrere partielle Funktionsdefinitionen bestehen (nicht zu verwechseln mit Funktionen höherer Ordnung), die durch verschiedene Muster unterscheidbar sind.

Definition 8.

```
f pat1 = ε1
f pat2 = ε2
. . .
f patn = εn
```

Beispiel

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

Aufgabe

1. Implementiere im HSLAB eine Zählfunktion, die alle Nullwerte einer Liste zählt, mit funktionaler Musterzerlegung:
 - In Mustern können auch konkrete Werte (anstelle von symbolischen Variablen), wie z.B. der Wert 0, eingesetzt werden, z.B. `div (x,0) = None`
 - Starte mit `count [] = 0`
2. Spielt die Reihenfolge der musterbasierten Definitionen von Funktionen eine Rolle?
 - Warum?

4.9. Haskell - Sequenzielle Ausführung**Problem**

- In einer rein funktionalen Sprache wie Haskell haben Funktionen *keine* Seiteneffekte. Deshalb spielt auch die Reihenfolge des Aufrufs eine weniger wichtige Rolle.
- Ein- und Ausgabe erzeugen einen Seiteneffekt auf die Programmumgebung. Sie müssen in der richtigen Reihenfolge ausgeführt werden.

do

Für die **sequentielle Ausführung** von IO-Aktionen stellt Haskell die *do*-Notation zur Verfügung:

Definition 9.

```
do {  
  Anweisung 1;  
  Anweisung 2;  
  ..  
  Anweisung n  
}
```

JSHC Implementierung

- In JSHC gibt es keine *do* Anweisung.
- Daher wird die Anweisungssequenz einer *do* Anweisung durch einen Präprozessor in eine Liste transformiert:

```
do {  
  I1;  
  I2; ⇒  
  ..  
  In  
}
```

```
reduce (+) [  
  I1,  
  I2,  
  ..  
  In  
]
```

- Eine Anweisung wird als Berechnung verwendet die einen Zahlenwert als Ergebnis liefert (i.A. 0)
- Die gesamte Sequenz liefert ebenfalls einen Wert, der durch die Listenreduktion (Summation) berechnet wird

- **Annahme:** Die Listenausdrücke werden der Reihe nach durch JSHC evaluiert!

4.10. Haskell - Eingabe und Ausgabe

putStr

Ausgabe einer Zeichenkette auf der Konsole

putStrLn

Ausgabe einer Zeichenkette mit nachfolgenden Zeilenumbruch auf der Konsole

getLine

Einlesen einer Zeichenkette von der Konsole (Tastatur)

Definition 10.

```
FUN putStr: String -> IO ()
FUN putStrLn: String -> IO ()
FUN getLine: IO String
```

Speichervariablen und Seiteneffekte

- Neben den IO Anweisungen gibt es Anweisungen um Daten zwischenspeichern → Speichervariablen!

Definition 11.

```
x <- getLine;
```

Beispiel


```
main = do {
  hSetBuffering stdout NoBuffering ; -- import System.IO
  putStr "Gib eine Zeichenreihe ein: ";
  s <- getLine ;
  putStr " Revertierte Zeichenreihe : ";
  putStrLn ( reverse s)
}
```

4.11. Haskell - Bedingte Ausführung

if-then-else

Bedingte Ausführung von Anweisungen nur in einer do-Umgebung. Achtung: Ausführung und nicht Berechnung! Es wird ein boolescher Ausdruck *cond* evaluiert. Ist dieser wahr dann wird die erste *then* Anweisung ausgeführt, sonst die zweite *else* Anweisung.

Definition 12.

```
do {
  if cond then
    Anweisung 1
  else
    Anweisung 2
  ..
  Anweisung n
}
```

5. Funktionen und Lambda Calculus

5.1. Definition, Deklaration, Applikation, Komposition

Zusammenfassung der Methoden

Definition (Implementierung)

```
DEF f par1 par2 .. =  
    €(par1,par2,..)  
DEF f = λ p1,p2,.. .  
    €(p1,p2,..)  
DEF f (par1,par2,..) =  
    €(par1,par2,..)
```

Deklaration (Typisierung)

```
FUN f : partyp1 → partyp2 → ..  
    → restyp  
  
FUN f : (partyp1 × partyp2 × ..)  
    → restyp
```

Applikation (Berechnung)

```
(f arg1 arg2 ..) → Wert  
(λ x,y,.. . €(x,y,..))
```

Komposition

```
(f ◦ g ◦ h ◦ .. ) arg1 arg2 ..
```

5.2. Lambda Calculus

Konzept: Jeder Ausdruck und jeder Wert wird als auswertbare Funktion betrachtet, so dass die ganze Programmierung auf der Übergabe von Funktionen als Parameter an Funktionen beruht.

Lambda Notation

- In der Lambda Notation werden Variablen an Ausdrücke gebunden

Definition 13.

$$\begin{aligned} \lambda x, y, z.. \rightarrow \varepsilon(x, y, z, ..) &\Leftrightarrow \\ \lambda x, y, z.. \bullet \varepsilon(x, y, z, ..) & \end{aligned}$$

Beispiel

```
v/g sin(2*phi) ⇔
λ v,g,phi → (v2/g sin(2*phi))
λ v,g,phi . (v2/g sin(2*phi))
```

Lambda Ausdrücke

- Ein Lambda Ausdruck ist eine anonyme Funktion, d.h. er definiert wie das Resultat in Abhängigkeit von der Eingabe berechnet wird ohne der Funktion einen Namen zu geben!

Freie und gebundene Variablen

- Die Bindungsregeln in einer Abstraktion $\lambda x \rightarrow \epsilon$ sind so geregelt:
 - ❑ Alle Vorkommen der Variablen x im Ausdruck ϵ gehören zu diesem Ausdruck (sind an dieses x und die Abstraktion *gebunden*).
 - ❑ Die gebundenen Variablen sind *nur* in dem Ausdruck sichtbar und verwendbar!
 - ❑ Zum Beispiel ist im Ausdruck $(\lambda x \rightarrow (y x))$ die Variable x eine gebundene Variable (durch λ gebunden), und y eine freie Variable
- Variablen (bzw. Vorkommen von Variablen), die in keinem Gültigkeitsbereich einer Bindung stehen, sind *frei*.
- Da es auch Konflikte geben kann durch Verwendung des *gleichen* Variablennamens, gilt dann die Regel: die *innerste Bindung* ist *sichtbar*/anwendbar.
- Ein Ausdruck ohne freie Variablen heißt *geschlossener Ausdruck*, anderenfalls *offener Ausdruck*.

Bindungsregeln und Klammerung

- ▶ Notation: $(s t)$ sind Anwendungen, d.h. t wird auf s angewendet, z.B. $(sq\ 4) = 16$
- ▶ Warum gilt: $((x\ y)\ z) = (x\ y\ z)$
Aber nicht: $(x\ (y\ z)) \neq (x\ y\ z)$?
Auflösung folgt gleich ...

Beispiele

- ▶ Wo sind freie und wo gebundene Variablen?

```

g = 9.81
sq1 x = x*x
sq2 = λ x -> x*x
wurfweitel v0 phi =
  let sq x = x*x in
  (sq v0)/g*(sin phi) ⇔
wurfweite2 = λ v0 phi -> ((λ x -> x*x) v0)/g*(sin phi)
wurfweite3 = λ v0 phi -> (sq2 v0)/g*(sin phi)

```

5.3. Funktionen als Werte

- ▶ Funktionen sind Werte erster Ordnung, d.h. $f = \lambda p . \epsilon \Leftrightarrow f\ p = \epsilon$
- ▶ Eine Funktion kann symbolischen Variablen und Funktionsparametern zugeordnet werden
 - D.h. Funktionen können Funktionen als Argumente übergeben werden
- ▶ Ein Lambda Ausdruck ist eine Funktion als Wert und kann von einer Funktion als Ergebnis geliefert werden!

Beispiele**Haskell**

```
f comb x y = (comb x y)/2
p = f
m = p (+) 1 2
incr = (+) 1
```

JavaScript

```
function f (comb,x,y) { return comb(x,y)/2 }
function add (x,y) { return x+y }
var p = f;
var m = p (add,1,2)
incr = function (x) { return x+1 }
```

5.4. Anonyme Funktionen

- ▶ Eine anonyme Funktion beschreibt die Abbildung von Variablen auf Werte ohne der Funktion einen konkreten Namen zu geben!
- ▶ Da Funktionen auch Werte sind und Funktionsparametern übergeben werden können gibt es zwei Anwendungsfälle:
 - ❑ Eine anonyme Funktion wird “identifizierbar”, d.h. anwendbar, durch Zuweisung (Bindung) an eine gebundene oder freie Variable
 - ❑ Eine anonyme Funktion wird “identifizierbar” durch Bindung an einen Funktionsparameter (eigentlich das gleiche)
- ▶ Ein Lambda Ausdruck ist eine anonyme Form (wie gehabt)!

Definition 14.

```
\ x y z -> e(x,y,z,..) ⇔
\ x y z . e(x,y,z,..) ⇔
λ x y z → e
```

- ▶ Sehr nützlich bei wiederholter Anwendung einer Funktion auf Listenelemente
- ▶ Wichtige Methode der Datenverarbeitung: **Map & Reduce**

Beispiele

Haskell

```
l = [1,2,3]
sq x = x*x
squares = map sq l
```

JavaScript

```
var l = [1,2,3]
function sq (x) { return x*x }
var squares = l.map(sq)
```

5.5. Data Mining - Datenverarbeitung

Ein kleiner Ausflug in Big Data ..

Motivation

- Großskalige Datenverarbeitung
 - ❑ Viele Daten verarbeiten (> 1 TB)
 - ❑ Parallelisierung auf Hunderttausende von CPUs
 - ❑ Einfache Handhabung und Umsetzung
- Map-Reduce bietet:
 - ❑ Automatische Parallelisierung und Verteilung
 - ❑ Fehlertoleranz
 - ❑ Stellt Status- und Überwachungstools bereit.
 - ❑ Abstraktion für Programmierer

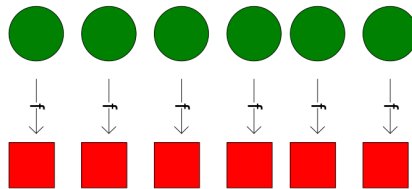
Funktionale Programmierung

- Map-Reduce Algorithmen, wie sie z.B. im Hadoop System verwendet werden, haben Ähnlichkeit mit funktionalen Programmiersprachen (wenn auch Map&Reduce kein spezifisches funktionales Problem ist):
 - ❑ Funktionale Operationen verändern keine Datenstrukturen: Sie erstellen immer neue;
 - ❑ Originaldaten sind unverändert vorhanden;
 - ❑ Datenfluss ist im Programmdesign implizit;
 - ❑ Reihenfolge der Operationen spielt keine Rolle.
 - ❑ Funktionale Programmierung wendet Operationen auf Listen an

Map0

```
function (f: 'a->'b, 'a list) -> 'b list
```

Ursprüngliche funktionale *map* Funktion. Erzeugt eine neue Liste durch Anwendung einer Funktion f auf jedes Element der Eingangsliste (oder Arrays)
→ **Abbildung**

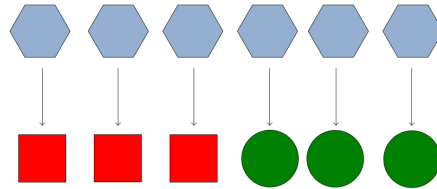


[a]

Map

```
function (in_key,in_value) -> (out_key,intermediate_value) list
```

Adaption der ursprünglichen *map* Funktion. Datensätze von einer Datenquelle (Zeilen aus Dateien, Zeilen einer Datenbank usw.) werden als Schlüssel / Wert-Paare in die Abbildungsfunktion eingegeben: z. B. (Dateiname, Zeile). *map* erzeugt aus der Eingabe einen oder mehrere Zwischenwerte zusammen mit einem Ausgabeschlüssel.

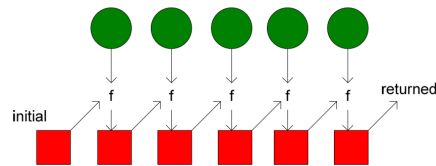


[a]

Fold

```
function (f:'a*'b->'b, x0:'b, lst;'a list) ->'b
```

Ursprüngliche funktionale *fold* Funktion. Iteriert über eine Liste und wendet f auf jedes Element plus einen Akkumulator an (Anfangswert x_0). f gibt den nächsten Akkumulatorwert zurück, der mit dem nächsten Element der Liste kombiniert wird → **Reduktion**.

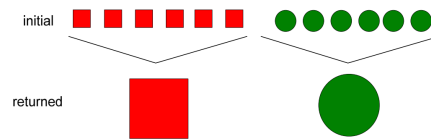


[a]

Reduce

```
function (out_key, intermediate_value list) -> out_value list
```

Adaption der ursprünglichen *fold* Funktion. Nach der Beendigung der Abbildungsphase werden alle Zwischenwerte für einen bestimmten Ausgabe Schlüssel zu einer Liste zusammengefasst. *reduce* kombiniert diese Zwischenwerte zu einem oder mehreren Endwerten für denselben Ausgabe Schlüssel (in der Praxis normalerweise nur ein Endwert pro Schlüssel)



[a]

Map-Reduce Methode in Hadoop

- Daten werden verteilt im HDFS (Hadoop Distributed File System) abgelegt

- Datenblöcke werden mehreren Mappern zugewiesen, die Schlüssel-Wert-Paare ausgeben, die parallel gemischt und sortiert werden.
- Der Reduzierungsschritt gibt ein oder mehrere Paare (mit Daten die den gleichen Schlüssel haben) aus, wobei die Ergebnisse im HDFS gespeichert werden

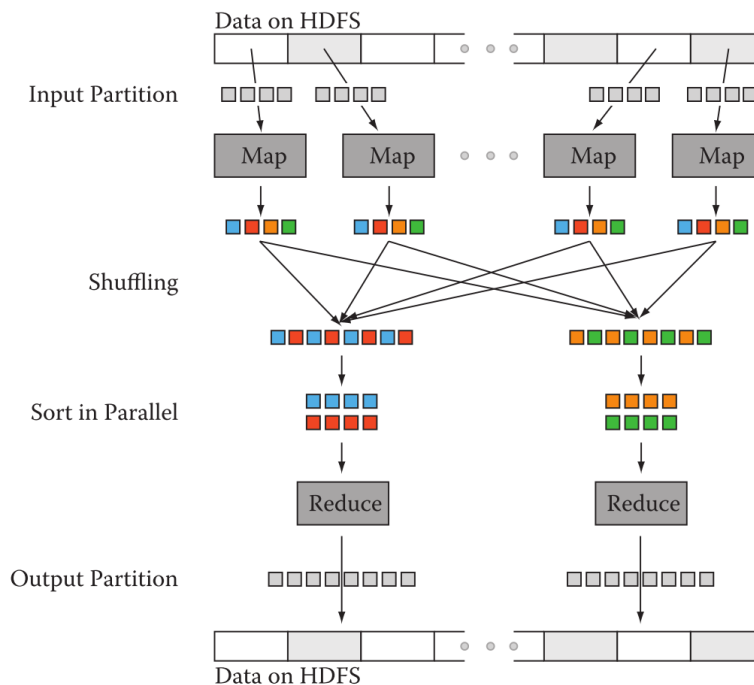


Abb. 5. Datenübertragung und Kommunikation eines MapReduce-Jobs in Hadoop [F].

5.6. Curry Form

Bei der Curryfizierung findet eine Umwandlung einer einzelnen Funktion mit n Argumenten in n Funktionen mit je einem einzelnen Argument statt.

- Traditionell kennen wir nur Funktionen die eine Menge von Argumenten als “ganzes” evaluieren, d.h., Eingabewerte führen immer zu Ausgabewerten (Zahlen usw.)
- Da Funktionen aber auch Funktionen zurück geben können kann man (wie bisher schon geschehen) die Funktion zerlegen und die Funktion-

sargumente einzeln evaluieren \rightarrow *Curry* Schreibweise (vom Mathematiker Haskell Curry)

Definition 15.

```
FUN f x y z : typx  $\rightarrow$  typy  $\rightarrow$  typz  $\rightarrow$  typr  $\Leftrightarrow$ 
FUN f x y : typx  $\rightarrow$  typy  $\rightarrow$  ( typz  $\rightarrow$  typr )  $\Leftrightarrow$ 
FUN f x : typx  $\rightarrow$  ( typy  $\rightarrow$  typz  $\rightarrow$  typr )
```

- Die Curry Form ist häufig flexibler (nützlicher) als die Tupelform durch *partielle Applikation*
- Beispiel: Die Definition einer neuen Inkrementalfunktion und Abbildungsfunktion für Listen

```
incr = (+) 1
mapsq = map \x -> x*x
```

Beispiel

Haskell

```
f (x,y,z) = x+y+z  $\Leftrightarrow$ 
f x y z = x+y+z  $\Leftrightarrow$ 
f x = \ y -> \ z -> x+y+z
f = \ x -> \ y -> \ z -> x+y+z
```

Lambda

```
 $\lambda$  (x,y,z) . x+y+z
 $\lambda$  x y z . x+y+z
 $\lambda$  x .  $\lambda$  y .  $\lambda$  z . x+y+z
```

JavaScript

```
function f(x,y,z) = { return x+y+z } ⇔
function f(x) = {
  return function (y) {
    return function (z) {
      return x+y+z
    }
  }
}
```

- Man kann die Curry Form immer in eine nicht-Curry Form transformieren:

```
uncurry f (x,y) = f x y
curry f x y = f(x,y)
```

- Die folgenden Funktionen unterscheiden sich daher mathematisch nur minimal (bezüglich des Funktionsraumes)

```
f1 (x,y) = x*2+y*3
f2 (y,x) = x*2+y*3
f3 x y = x*2+y*3
f3 y x = x*2+3*y
```

5.7. Funktionen höherer Ordnung

- Konzepte:
 - ❑ Funktionen sind Werte
 - ❑ Funktionen als Parameter
 - ❑ Funktionen als Rückgabewerte von Funktion

Funktionen höherer Ordnung ("higher-order functions") nehmen Funktionen als Argumente oder liefern Funktionen als Resultate.

- Bei der funktionalen Komposition werden Funktionen *statisch* (also zur Programmierzeit) zu neuen Funktionen zusammengesetzt

- Mit Funktionen höherer Ordnung kann man *dynamisch* zur Laufzeit neue Funktionen erzeugen!
- Funktionen höherer Ordnung stellen eine Abstraktion und Generalisierung dar.

Beispiel: Listen

- Listen (oder Vektoren) werden häufig mit der gleichen Funktion transformiert, d.h. `sum sq [1,2,3]`

Beispiel: Integral

- Das Integral einer Funktion f im Intervall $[a,b]$ (Bestimmtes Integral):

$$I(f, a, b) = \int_a^b f(x) dx$$

$$I(f, a, b) = \sum_{i=a, a+\Delta, a+2\Delta, \dots}^b f(i)\Delta \text{ mit } \Delta = \frac{(b-a)}{N}$$

```
int f delta = \ a b ->
  (sum (map f [ x*delta | x <- [a/delta..b/delta] ]))*delta
```

- Die Anwendung der `int f delta` Funktion ist eine partielle Applikation von f und liefert eine neue Funktion als Wert zurück!

5.8. Totalisierung**Partielle Funktionen**

- Es gibt Funktionen bei denen der Definitionsbereich nicht vollständig auf den Wertebereich abgebildet werden kann.
- Die bekannteste partielle Funktion ist die Ganzzahldivision $div = \lambda ab.a/b$:

$$\mathbb{D} = \langle \dots, , , , \dots \rangle$$

Teilmenge des Definitionsbereiches von div

$\mathbb{W} =$

Wertebereich der Divisionsfunktion

$\mathbb{R} = \langle \dots, \langle \perp \rangle, \langle 0 \rangle, \langle \perp \rangle, \langle 1 \rangle, \langle 2 \rangle, \dots \rangle$

Die Relationsmenge enthält ‘Löcher’ wo der Nenner 0 ist

Totalisierung

- ▶ Unvollständige Funktionen stellen mathematisch wie numerisch/informativ ein Problem dar
- ▶ Die Totalisierung fügt ein symbolisches “Not a Number” für ein undefiniertes Ergebnis einer Funktion ein um die Relationsmenge vollständig zu machen
- ▶ Dazu wird das Symbol “NaN” oder mathematisch \perp dem Wertebereich hinzugefügt:

$$\square \mathbb{W}^* := \mathbb{W} \cup \{\perp\}$$

Beispiel

- ▶ Totalisierung der Divisionfunktion durch Einführung eines Summentyps:

```
data Num x = NaN | Value x
div a b = if b == 0 then NaN else Value (a/b)
add (Value a) (Value b) = Value (a+b)
```

Aufgabe

1. Erweitere die Totalisierung der arithmetische Funktionen damit diese mit den “Werten” *NaN* umgehen können

5.9. Polymorphie

- ▶ Polymorphie bedeutet die Möglichkeit dass Variablen und Funktionsparameter Werte mit verschiedenen Datentypen annehmen können.
- ▶ Dabei gibt es zwei (+1) Arten von Polymorphie:

- ❑ *Statisch*: Bei der Definition und Applikation im Programm: Der konkrete Datentyp wird zur Kompilierungszeit durch die Applikation festgelegt → **Typinferenz** (Haskell)
 - ❑ *Dynamisch*: Zur Laufzeit können Variablen und Funktionsparameter noch Werte mit unterschiedlichen Datentypen aufnehmen und verarbeiten (JavaScript)
 - ❑ *Überladung*: Es wird nicht eine polymorphe Funktion definiert und mit unterschiedlichen Typsignaturen verwendet, sondern eine Menge gleichnamiger Funktionen mit unterschiedlichen Typsignaturen die dann bei der Applikation zur Kompilierungszeit zugeordnet werden.
- Polymorphe Funktionen können polymorphe Funktionen verwenden/aufrufen. Aber:
- Das Problem: Irgendwann zur Kompilierungs- oder Laufzeit müssen Werte mit konkreten Funktionen/Operatoren verarbeitet werden.

Beispiel: Arithmetik

Eine polymorphe Funktion zur Addition von: Ganzen Zahlen (*Int*), reellen Zahlen (*Real*), und Zeichenketten (*[char]*):

```
FUN add:  $\alpha \rightarrow \alpha \rightarrow \alpha$ 
FUN addInt: Int  $\rightarrow$  Int  $\rightarrow$  Int
FUN addReal: Real  $\rightarrow$  Real  $\rightarrow$  Real
FUN addString: [char]  $\rightarrow$  [char]  $\rightarrow$  [char]
```

Beispiel: Tupel und Listen

```
fst :: (a,b)  $\rightarrow$  a      snd: (a,b)  $\rightarrow$  b
fst(x,_) = x           snd(_,y) = y
```

Beispiel: Listen

```
map :: (t  $\rightarrow$  u)  $\rightarrow$  [t]  $\rightarrow$  [u]
map f [] = []
map f (a : tail) = f a : map f tail
```

5.10. Anwendungsbeispiel: Quicksort

- Sortierung ist ein gutes Beispiel für ein *Divide&Conquer* Problem, welches man auf einen *map-reduce* Algorithmus zurückführen kann.

Definition 16.

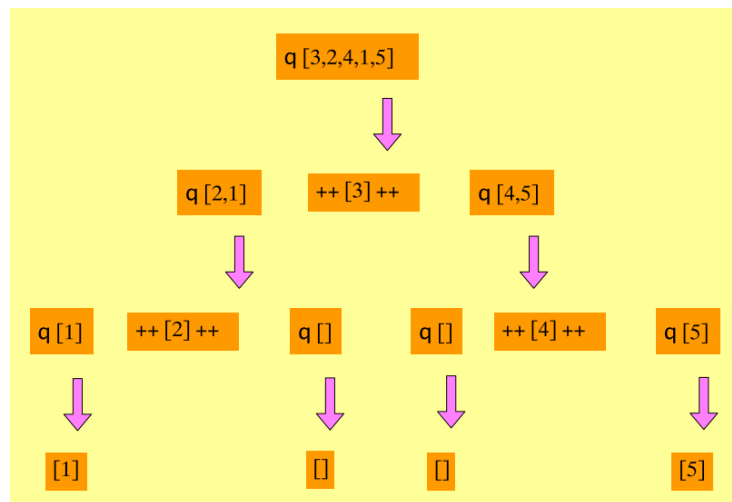
```

qsort :: [Int] → [Int]
qsort [] = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x ]

```

- Bei der Sortierung können die Werte einer Liste aufsteigend [1,2,3,..] oder absteigend [10,9,8,..] sortiert werden.
- Frage: Werden im obigen Algorithmus die Werte einer Liste aufsteigend oder absteigend sortiert?

Beispiel



5.11. Anwendungsbeispiel: Quicksort 2.0

- Im vorherigen Beispiel wurden die Elemente mittels der eingebauten Relationen Funktionen `<=` und `<` sortiert, anwendbar nur auf numerische Werte.
- Erweiterung der Sortierung auf beliebige Werttypen mittels nutzerdefinierter Vergleichsfunktionen `cmp(a,b)` die 0 liefert wenn `a==b`, -1 wenn `a<b`, sonst 1:

```

qsort :: (Int -> Int -> Bool) -> [Int] -> [Int]
qsort cmp [] = []
qsort cmp (x:xs) =
  qsort cmp smaller ++ [x] ++ qsort cmp larger
  where
    smaller = [a | a <- xs, (cmp a x) /= 1 ]
    larger  = [b | b <- xs, (cmp b x) == 1 ]

```

5.12. Funktoren

- **Funktoren** sind *Parametrisierte Strukturen*
- **Strukturen** werden im nächsten Kapitel eingeführt und dienen der namentlichen Bindung von Werten.
- Neben der klassischen Polymorphie mit ihrer Einschränkung auf Typvariablen stellt eine Parametrisierung von Strukturen eine Möglichkeit bereit, in der neben Typvariablen auch Funktionsvariablen vorkommen können.
- Dieses Programmierkonstrukt basiert auf Modularisierungstechniken.

Später mehr ...

6. Datentypen

6.1. Typisierung

- Bisher wurden z.B. Funktionen oder auch Ausdrücke ohne explizite Typdeklaration definiert oder berechnet. Untypisch in traditionellen Programmiersprachen wie C, C++, JAVA, usw.

- Die Datentypen werden dabei in der funktionalen Programmierung durch Ableitung (*Typinferenz*) ermittelt. Wenn nicht möglich gibt wird der Typ eines Wertes bzw. Funktionsparameters durch einen Platzhalter vorläufig bestimmt $\rightarrow \alpha, \beta, ..$
- Man kann aber auch explizit Typen und Typsignaturen angeben, in folgender Form:

Definition 17. $\epsilon :: \text{typ}$ **Beispiele**

```
sum a b = a+b
sumInt a b = (a::Int)+(b::Int)
mapInt f l = map f (l::Int)
```

- Typen sind auch nur Terme!
- Daher kann man neue Typen mit Typausdrücken konstruieren
- Einfachster Typausdruck: **Synonym**

Definition 18. (*Nutzereigene Typdefinition*)type T = ϵ_t **Beispiele**

```
type Euro = Double
type Dollar = Double
type Exchange = (Euro , Dollar)
type ExchangeBack = (Dollar , Euro)
let exchange euro = ((euro::Euro)*1.2)::Dollar
::t exchange
>> exchange :: Euro -> Dollar
```

6.2. Typprüfung

Dynamische Prüfung (run-time check)

Bei der Ausführung des Programms wird bei jedem Aufruf $f(e)$ einer Funktion $f: A \rightarrow B$ überprüft, ob der Argumentwert e dem geforderten Typ A genügt und ob das Resultat dem Typ B genügt.

- Vorteil: Ausdrucksstarkes Typsystem; Nachteil: Performanz zur Laufzeit (Typprüfung)

Statische Prüfung (compile-time check)

Der Compiler analysiert das Programm (mit Typdeklarationen und Typinferenz) um sicherzustellen, dass jeder Funktionsaufruf die Typisierung erfüllt. Man kann das als den einfachsten Grenzfall einer Programmverifikation auffassen.

- Vorteil: Prüfung findet nur einmal statt; Nachteil: Typkonzepte müssen einfach sein um automatisch bewiesen zu werden (durch den Compiler)

6.3. Typinferenz

- Die *Typeinferenz* ist die Fähigkeit eines Übersetzers oder einer Laufzeitumgebung den Datentypen von Variablen, d.h. Ausdrücken, Funktionsparametern und Rückgabewerten aus eine Kette (Baumstruktur) von Applikationen und Ausdrücken *abzuleiten*
- Bei der Typinferenz wird nach konkreten Datentypen gesucht und diese entlang des Evaluierungsbaums von Ausdrücken propagiert und aufgelöste Typen substituiert.
- Beispiel:
 $\lambda a b. (b - a + 1.0) \Rightarrow$
 $1.0 :: Fractional \Rightarrow a + 1 :: Fractional \Rightarrow a :: Fractional$
 $\text{da } + :: \alpha \rightarrow \alpha \rightarrow \alpha \Rightarrow (b - \epsilon :: Fractional) :: Fractional \text{ da } - :: \alpha \rightarrow$
 $\alpha \rightarrow \alpha \text{ und } b :: Fractional \Rightarrow$
 $(\lambda a :: Fractional b :: Fractional . \epsilon :: Fractional) :: Fractional \Leftrightarrow$
 $Fractional \rightarrow Fractional \rightarrow Fractional$

Beispiel: Listen

```

id x = x
incr x = x+1
incr2 x = x+1.0
map :: (t -> u) -> [t] -> [u]
map f [] = []
map f (a : tail) = f a : map f tail
mapDecr = map (\ x -> x-1)

```

6.4. Algebraische Datentypen

- Algebraische Datentypen werden benutzt um Werte (Daten) namentlich zu binden und identifizierbar zu machen.

Eine der häufigsten Gründe für die Einführung neuer Datenstrukturen ist die Beobachtung, dass eine Gruppe von Daten logisch zusammengehört und gemeinsam etwas Neues, Eigenständiges darstellt.

- Interessanterweise zeigt sich bei Datenstrukturen das gleiche Phänomen wie bei Funktionen: Die reiche Fülle von Möglichkeiten erwächst aus einer kleinen Zahl von elementaren *Konstruktionsprinzipien*. Dies sind bei Funktionen:
 - ❑ Funktionsdefinition, Funktionsapplikation, Funktionskomposition, Tupelbildung, Fallunterscheidung und Rekursion;
- Bei Datenstrukturen:
 - ❑ Definition, Komposition, Produktbildung, Summenbildung und Rekursion!
- Unser Ziel ist also: Wir wollen aus vorhandenen Datenstrukturen neue Datenstrukturen aufbauen.
- Dafür gibt es im Wesentlichen drei Konstrukte:
 - ❑ Tupelbildung, Arrays (Produkt);
 - ❑ Variantenbildung (Summe oder Gruppe);
 - ❑ Aufzählung.
- Dass diese Konstrukte die elementaren Bausteine eines ganzen Universums von Datenstrukturen sind, ist keine Entdeckung der Informatik, sondern

- wieder einmal - der Mathematik, genauer: der elementaren **Mengenlehre**.

- Auch eine zweite Beobachtung verdanken wir der Mathematik, und zwar diesmal der Algebra: Die Konstruktion von Datenstrukturen liefert nicht nur Mengen neuer Werte, sondern gleichzeitig auch kanonische **Operationen** für diese Werte: Datenstrukturen und ihre kanonischen Operationen bilden eine logisch zusammengehörige Einheit - *das Eine macht ohne das Andere keinen Sinn*.

6.5. Algebraische Datentypen :: Produkttypen

- Produkttypen sind **Aggregationen** und Gruppentypen

Einsortige Produkttypen

- Listen und Arrays sind Produkttypen, d.h. die Elemente eines Produkttyps werden prinzipiell alle zusammen in Berechnungen verwendet (e_1 und e_2 und ..)
- I.A. ist der Datentyp der Elemente gleich, d.h. es handelt sich um einen **einsortige Komposition**
- Die Elemente von Produkttypen sind i.A. durch einen **Index** (ganze Zahl) identifizierbar und referenzierbar.

Mehrsortige Produkttypen: Tupel

- **Tupel** sind mehrsortig, d.h., die Werte der einzelnen Tupelemente können zu verschiedenen Datentypen gehören.
- Aber die Elemente sind anonym (wie anonyme Funktionen) und *nicht selektierbar*
- Tupel werden i.A. ohne vorherige Typdefinition erzeugt (anonym und dynamisch zur Laufzeit)
- Es können aber auch namentliche **Tupeltypen** definiert werden:

Definition 19.

data $T = T \text{ typ}_1 \text{ typ}_2 \dots$

Mehrsortige Produkttypen: Datenstrukturen/Records

- Hier sind die Elemente des Produkttyps über einen **eindeutigen Namen** (i.A. auf den speziellen Produkttyp begrenzt) identifizierbar und referenzierbar (selektierbar).
- Die gesamte Typsignatur eines mehrsortigen Produkttyps ist genau wie beim Tupel ein Produkt der Typen der Einzelelemente:

Definition 20.

TYPE $T = \text{typ}_1 \times \text{typ}_2 \times \text{typ}_3 \times \dots$

- Die Definition eines Strukturtyps (Gruppenbildung) erfordert die Aufzählung der Typelemente in Form von Tupeln $\langle \text{Name}_i, \text{Typ}_i \rangle$,
- Die Definition eines Strukturtyps kann wieder rein funktional durch eine **Typkonstruktionsfunktion** erfolgen, die i.A. gleichnamig wie der Produkttyp ist:

Definition 21.

DATA $T = T(e_1 : \text{typ}_1, e_2 : \text{typ}_2, \dots)$

Beispiele

```
DATA point == point(x : real, y : real)
DATA line  == line(p1: point, p2: point)
DATA circle == circle(center:point, radius: real)
```

Erzeugung von Datenstrukturen

- Die reine Typdefinition ist hier nicht ausreichend. Sinn macht sie nur bei der Instanzierung (Erzeugung) von Daten (zusammengesetzten Werten) mittels der Konstruktionsfunktion

Definition 22.

DEF $x = T(\epsilon_1, \epsilon_2, \dots)$

Zugriff auf Strukturelemente

- Der Sinn und Zweck der namentlichen Kennzeichnung von Elementen ist der einfache Zugriff auf die Elemente über deren Namen. Auch dieser Zugriff kann funktional über eine gleichnamige Selektorfunktion erfolgen:

Definition 23.

DATA $T = T(e_1 : \text{typ}_1, e_2 : \text{typ}_2, \dots)$

FUN $e_1 : T \rightarrow \text{typ}_1$

- D.h. um auf ein Element e_i einer Datenstruktur s des Typs T (lesend) zugreifen zu können wird die Funktion $e_i(s)$ verwendet \rightarrow Die Selektorfunktion liefert den Wert des Elements.
- Häufig wird eine Kurzschreibweise in der Form $s.e_1$ verwendet, wobei s eine vom Typ T abgeleitete Datenstruktur ist.
- In vielen Programmiersprachen sind die Strukturelemente (Variablen) ebenso (über die Selektorfunktion) veränderlich. Nicht so in der (strikten) funktionalen Programmierung!

Definition eines Strukturtyps in Haskell

- Im Grunde sind Datenstrukturen (Records) in Haskell typisierte Tupel!
- Zunächst definiert man einen Summentyp (kommt später) T mit nur einem Untertyp (kann gleichnamig sein) T über das Schlüsselwort `data` und den Strukturelementen als Parameter des Summentypelements T :

Definition 24.

data $T = T \text{ typ}_1 \text{ typ}_2 \dots$

Beispiel

```
data Eintrag = E
String -- Vorname
String -- Nachname
String -- Straße
String -- Stadt
String -- Land
getVorname :: Eintrag -> String
getVorname (E n _ _ _ _) = n
setVorname :: String -> Eintrag -> Eintrag
setVorname n (E _ a b c d) = E n a b c d
createVorname :: String -> Eintrag
createVorname n = E n undefined undefined undefined undefined
-- analog für die weiteren Felder
-- ...
```

Handhabung von Datenstrukturen in Haskell

- Da es in Haskell nicht direkt die Möglichkeit gibt einen Strukturtyp (Record mit namentlicher Identifizierung der Elemente) zu definieren geht man wie folgt vor:
 1. Definition eines einelementigen Summentyps:
`data T = T typ1 typ2 ..`
 2. Definition der einzelnen Selektionsfunktionen über eine Musterzerlegung:
`ei (T _ .. xi _ ..) = xi`
 3. Instanzierung von Werten vom Strukturtyp *T*:
`s = T ε1 ε2 ..`

Recordsyntax in Haskell

- Da bei der Tupeltypisierung keine Elementnamen enthalten sind wird die Strukturtypdefinition und Verwendung schnell unhandlich und verwirrend.
- Erweiterte Tupeltypdefinition und Elementselektion mit Feldnamen:

Definition 25.

`data T = T { e1 :: typ1, e2 :: typ2, .. }`

`ei(T { ei = xi }) = xi`

Beispiel

```
data Eintrag = E {
  vorname :: String,
  nachname :: String,
  strasse :: String,
  stadt :: String,
  land :: String }
```

Beispiele in Haskell

Beispiele in JavaScript

6.6. Algebraische Datentypen :: Summentypen

- Sehr häufig hat man das Problem, dass ein Typ Elemente zusammenfassen soll, die inhaltlich etwas Gemeinsames darstellen, aber strukturell unterschiedlich aufgebaut sind.
- In diesen Situationen lassen sich die Elemente des Typs in verschiedene Varianten klassifizieren. Im Sinne der *Mengenlehre* haben wir es dann mit einer so genannten direkten *Summe* zu tun, d.h. im Wesentlichen mit einer disjunkten Vereinigung.

Beispiele

```
DATA figure == line(pl : point , p2 : point)
              triangle (pl : point, p2 : point, p3 : po int)
              circle( center: point, radius : real)
DATA address == st (plz : nat , ort: denotation, str : denotation)
                pf (plz : nat , ort : denotation, postfach : nat )
DATA result == ok(value : real)
              error(message : denotation)
DATA infNat == normal(value : nat)
              infinity
```

- Konstruktor-Varianten. Diese Beispiele illustrieren, dass ein Summentyp aus mehreren Varianten besteht, die ihrerseits Produkte sind.
- Als Grenzfall sind auch Konstanten als Varianten möglich → Aufzählungstyp. Da die einzelnen Varianten Produkte sind, können wir ihre Konstruktorfunktionen wie üblich benutzen, um die Elemente des Summentyps zu erhalten.

Summentypen in Haskell

Definition 26. (*Summentyp*)

```

data T =
  T1 typ1,1 typ1,2 .. |
  T2 typ2,1 typ2,2 .. |
  ..

```

Summentypen in JavaScript

- JavaScript kennt keine Summentypen
- Implementierung mit mehrsortigen Strukturen die ein gemeinsames Tag Element besitzen.

```

var line = {
  tag: 'line',
  p1: {tag: 'point', x: 0, y: 20} , .. };
var circle = {
  tag: 'circle',
  center: {tag: 'point', x: 0, y: 20} , .. };
function Line (p1,p2) {
  return {tag: 'line', p1:p1, p2:p2}
}
function Circle (center,radius) {
  return {tag: 'circle', center:center, radius:radius}
}
function height (o) {
  switch (o.tag) {
    case 'line': return Math.abs(o.p1.y-o.p2.y);
    case 'circle': return o.radius*2;
  }
}

```

6.7. Datentyp Some-or-None

- In vielen Funktionen können Fehler auftreten, die den Benutzer nicht weiter interessieren. Er möchte lediglich wissen, ob die Berechnung erfolgreich war.
- Z.B. die Suche in einem Telefonbuch ergibt als Ergebnis keinen Eintrag. Häufig wird dann ein "leeres" Ergebnis (0, [], ..) zurückgegeben.

- Besser ist ein Summentyp für das Ergebnis der entweder einen konkreten gebundenen Wert *Some φ* ; zurück gibt oder ein Symbol *None*:

```
data Maybe v = Some v | None
```

Beispiele

```
teilen :: Int -> Int -> Maybe Int
teilen n 0 = Nothing -- man kann nicht durch 0 teilen
teilen n m = Some (div n m)
teilenMoeglich :: Int -> Int -> String
teilenMoeglich a b = case (teilen a b) of
  Some _ -> " Der Divisor war nicht 0"
  Nothing -> " Die Division ist nicht möglich"
```

- Algebraische Datentypen können wie im Fall **Maybe** **polymorph** definiert werden!
- An die Daten von Summentypen und deren Elementen kann man wieder nur über eine Musterzerlegung gelangen, d.h., ein Ausdruck mit der Subtypkonstruktion (der Variante) auf der linken Seite mit Parametern und dem zu zerlegenden Wert:

Definition 27.

```
data T = T1 typ1,1 typ1,2 .. | ..
```

```
e1,i(T1 .. xi ..) = xi
```

6.8. Aufzählungstypen

- Der Aufzählungstyp ist ein Spezialfall des Summentyps mit rein symbolischen Elementen (d.h. Produkttypen nullter Ordnung):
- Die folgenden Typen illustrieren Situationen, in denen eine kleine Gruppe von Werten einen neuen Typ darstellt.

Definition 28.

```
data T = T1 | T2 | ..
```

- Achtung: In Haskell fangen Typnamen immer mit einem Großbuchstaben an, daher auch die Summentypen!

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
data Color = Blue | Green | Red | Yellow
data Switch = On | Off
data TrafficLight = Green | Amber | Red
today = Tuesday
```

6.9. Rekursive Algebraische Typen

- ▶ Typkonstruktoren können wie bereits eingeführt wurde auf gewisse Weise als Funktionen über Typen interpretiert werden.
- ▶ Rekursion lässt sich nicht nur als Entwurfstechnik für Funktionen verwenden, sondern auch für die Definition von Datentypen.
- ▶ Oft enthält eine Datenstruktur als Bestandteile wieder Elemente derselben Sorte.
- ▶ Ausdrücken lassen sich solche Typen mit Hilfe von Produkten und Summen, wobei - analog zur Funktionsdefinition - jetzt der deklarierte Typ selbst auf der rechten Seite vorkommt.
- ▶ Die Liste haben wir bereits als rekursiven Datentypen kennen gelernt. Eine Liste ist entweder leer oder sie besteht aus einem Kopfelement und einer Restliste. Als algebraischen Datentypen können wir das beispielsweise wie folgt definieren:

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

6.10. Module

- ▶ Aus Gründen der Wiederverwendbarkeit, Wartung und Fehlerlokalisierung ist es ratsam, Programme in Teile zu zerlegen und durch diese Modularisierung das Abstraktionsniveau zu erhöhen:
 - ❑ Funktionen
 - ❑ Typen
 - ❑ Module
 - ❑ Bibliotheken
- ▶ Gerade bei Projekten mit mehreren Programmierern ist es ratsam, Aufgaben in mehrere Bereiche aufzuteilen, die dann parallel bewältigt werden.

- Ein wichtiges Konzept für große Softwareprojekte stellt die Definition von Schnittstellen (Interfaces) dar → Typsignaturen
- Ein **Modul** besteht aus Deklarationen (**Schnittstelle**) und Definitionen (**Implementierung**)
- Haskell bietet in diesem Zusammenhang die Definition von Modulen an.
- Ein Modul wird in Haskell mit dem Schlüsselwort `module` definiert.
- Höchstens ein Modul pro Datei ist dabei erlaubt.
- Als Konvention gilt, dass der Name eines Moduls identisch mit dem Dateinamen ist.

Definition 29.

```
module M (  
    e1,  
    e2,  
    e3,  
    ..  
    ei  
) where  
ei :: typi  
ei = ei  
..
```

- Die in einem Modul definierten Elemente (Funktionen, Typen, Terme) sind nur innerhalb des Moduls nach *where* sichtbar. Daher:
- Module bzw. Ausschnitte (einzelne Funktionen usw.) können in andere Module über das Schlüsselwort `import` importiert werden (Ein Skript ist ebenso ein Modul):

Definition 30.

```
import M ( e1, e2, .. )
```

- Einzelne Elemente lassen über den Punktoperator `M.ei` selektieren.

6.11. Typklassen

- Typklassen bieten die Typisierung von Typen!

- Man erhält ein abgestuftes System der Typisierung (horizontal), verbunden mit einem abgestuften System der Modularisierung (vertikal).
- Der innerste Bereich betrifft die Werte. Dies sind atomare Werte wie Zahlen, Buchstaben etc., zusammengesetzte Werte wie Tupel, Listen, Arrays etc. und auch Funktionen.

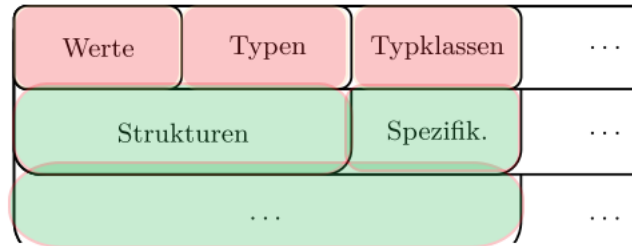


Abb. 6. Ein Stufenmodell der Typisierung und Modularisierung [B]

- Die Werte werden durch Typen klassifiziert. Es gibt atomare Typen, Tupeltypen, Funktionstypen etc.
- Die Typen wiederum werden durch Typklassen klassifiziert. Dazu gehören Konzepte wie *Eq*, *Ord*, *Numeric* usw.
- Auf der ersten Modularisierungsstufe erlauben Strukturen die Zusammenfassung von Werten und Typen zu neuen Einheiten.
- Diese Strukturen werden durch Spezifikationen typisiert.

Werte	Typen	Typklassen
2, (7.5, 12), "hallo", <i>sin</i> , <i>curry</i> , $\lambda x \cdot x + 1$	<i>Int</i> , (<i>Real</i> , <i>Real</i>), (<i>Int</i> \rightarrow <i>Real</i>), <i>Seq</i> ($_$)	<i>Eq</i> , <i>Ord</i> , <i>Numeric</i> <i>Fun</i> , <i>Interval</i>
<u>Wert gehört zum Typ</u>		<u>Typ gehört zur Typklasse</u>
2: <i>Int</i> <i>sin</i> : (<i>Real</i> \rightarrow <i>Real</i>)		<i>Int</i> : <i>Ord</i> <i>Set</i> : <i>Eq</i> \rightarrow <i>Type</i>

Abb. 7. Typisierungsrelationen: Werte, Typen, und Typklassen [B]

Definition 31.

Eine **Typklasse** C charakterisiert eine Menge von Typen (genauso wie ein Typ eine Menge von Werten charakterisiert). Eine Typklasse ist weiterhin eine Menge von Typen, die bestimmte Eigenschaften erfüllen.

- Zu **Typklassen** gehören **Operatoren** wie z.B. die Gleichheitstest und Relationen (Klasse *Eq*) oder ein Formatted Printer (Klasse *Show*)

Eq

Alle Typen in *Eq* unterstützen die Funktionen (`==`) und (`/=`).

Ord

Alle Typen in *Ord* haben eine (`<=`) Relation auf ihren Typen und unterstützen alle weiteren Vergleichsoperatoren

Show

Alle Typen in *Show* verfügen über eine textuelle Formatierungsfunktion für die Ausgabe von Werten auf dem Bildschirm.

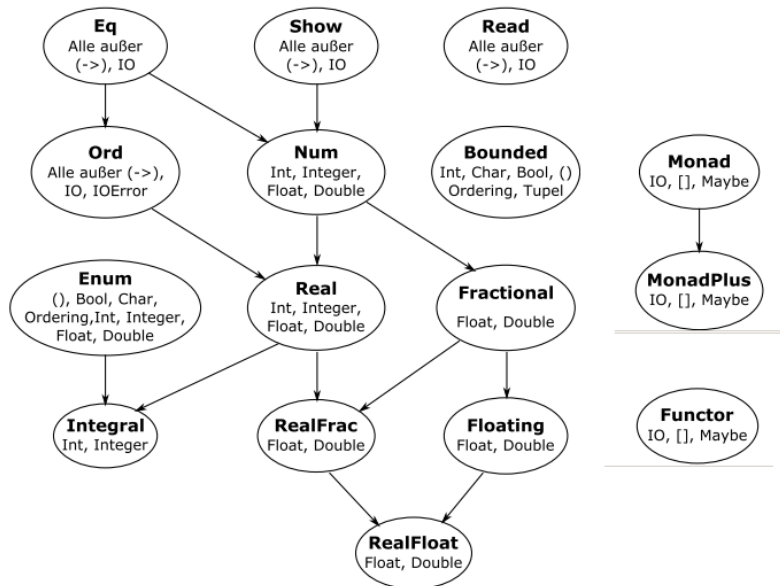


Abb. 8. Haskell: Definierte Klassen der Prelude und ihre Zugehörigkeiten [C]

Instantiierung

Um beispielsweise einen neuen Datentypen zu einer Instanz der Klasse *Show* zu machen, müssen wir eine Funktion schreiben, die die Werte dieses Typen in Strings umwandelt.

- Ein Beispiel einer Instantiierung einer nutzerdefinierten Enumeration mittels des Schlüsselwortes `deriving`:

```
data Saison = Fruehling | Sommer | Herbst | Winter
  deriving(Eq, Ord, Enum, Show, Read, Bounded)
```

- Der neue Datentyp wird den Typklassen *Eq*, *Ord*, *Enum*, *Show*, *Read*, und *Bounded* hinzugefügt (eine *Instanz von den Typklassen*).

Manuelles Instanzieren

Alternativ zur automatischen Instanziierung von Typen zu Klassen lassen sich diese auch manuell vornehmen.

- Das ist in den Fällen erforderlich, wenn entweder eine automatische Instanz für den Typen nicht erzeugt werden kann oder die erzeugte Instanz nicht das Gewünschte leistet.

Definition 32.

```
instance Klasse Datentyp
  where ...
```

Beispiele

```
data Boolean = T | F

instance Show Boolean
  where
    show T = "Wahr"
    show F = "Falsch"

instance Eq Boolean      instance Ord Boolean
  where                  where
    T == T = True        F <= T = True
    F == F = True        x <= y = x == y
    _ == _ = False
```

7. Abstrakte Datentypen

7.1. Abstrakte Datentypen

- Abstrakte Datentypen (ADT) werden in Programmen verwendet um neue

- ❑ **Datenstrukturen** (Realisierung der Objektmengen eines ADT's mit Mitteln der Programmiersprache) und
- ❑ **Operationen** (Algorithmen)

einzuführen (Vergleiche Haskell Intensivkurs [C]) .

- Dabei dienen die Operationen dem Zugriff und der Manipulationen der durch den ADT strukturierten Daten
- Ein ADT ist ein nutzerdefinierter Typ der Operationalität einführt die die Programmiersprache selber nicht bereit stellt.
- Beispiele
 - ❑ Listen (einfach und doppelt verkettet)
 - ❑ Warteschlangen
 - ❑ Wörterbücher und Hashtabellen
 - ❑ Stapelspeicher
 - ❑ Datenbanken
 - ❑ Bäume und Graphen
 - ❑ ...

7.2. Wörterbuch

- Der abstrakte Datentyp Wörterbuch findet in vielen Anwendungen seinen Einsatz und trägt den Namen aufgrund der Ähnlichkeit zum physikalischen Wörterbuch.
- Elemente können mit einem Schlüssel darin abgelegt und anschließend effizient wieder gefunden werden.
- Ein Wörterbuch ist eine Liste von (*Schlüssel, Wert*)-Tupeln.
- Die Frage ist wie können Einträge Schnell anhand des Schlüssels gefunden werden?

Randbedingungen

- Ist der Schlüssel eine ganze Zahl oder eine Zeichenkette? Au fälle Fälle sollte er abzählbar sein (Typ Ordinal).
- Der Schlüssel kann eindeutig oder mehrdeutig sein (relevant und kritisch)

- Ein Wert (der Datensatz) kann einzigartig sein oder nicht (Mehrdeutigkeit nicht relevant und unkritisch)
- Implementierung mit einem Modul

```

module Woerterbuch (
  WBuch,
  leeresWBuch ,
  einfuegen , finde, loesche)
where
  newtype WBuch k v = D [(k,v)] deriving Show
  leeresWBuch :: Eq k => WBuch k v
  leeresWBuch = D []
  einfuegen :: (k, v) -> WBuch k v -> WBuch k v
  einfuegen x (D l) = D (x:l)
  finde :: Ord k => k -> WBuch k v -> Maybe v
  finde k (D l) = lookup k l -- lookup aus Prelude
  loesche :: Eq k => k -> WBuch k v -> WBuch k v
  loesche key (D l) = D [(k,v) | (k,v) <- l, k/=key]

```

- In der Exportliste des Moduls wird nur der Typ des Wörterbuchs *WBuch* angeben, nicht aber den Typkonstruktor *D*.
 - So ist es den Anwendern nicht möglich eigene Wörterbücher zu erstellen.

Effiziente Implementierung

Eine einfache Möglichkeit, den ADT Wörterbuch zu implementieren, besteht darin, nur eine Liste mit dem Tupel (Schlüssel *k*, Wert *v*) anzulegen.

- Das Einfügen ist trivial ($O_{\text{einfuegen}}(N) = 1$), aber das Finden und das Löschen der Elemente lässt sich nur in linearer Zeit realisieren. $\rightarrow O_{\text{lookup}}(N) = N$.
- Das Suchen kann erfolgreich sein (Typ *Maybe*: *Just v*) oder ergebnislos (*Nothing*) und wird durch Iteration über die Wörterbuchliste durchgeführt:

```

lookup k ((x,v):tail) = if (x==k) Just v else lookup k tail
lookup k [] = Nothing

```

7.3. Hashtabellen

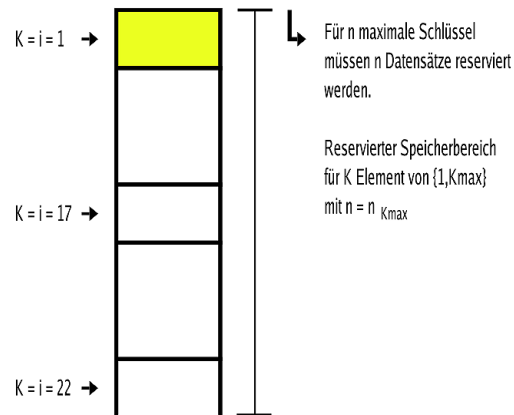
Ansatz

- ▶ Die Liste wird in einem linearen Array (Tabelle) gespeichert und der Schlüssel ist die Indexposition in der Tabelle.

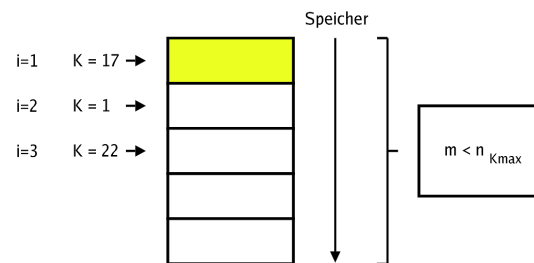
Vorüberlegungen

- ▶ Listen erlauben eine Menge von Datensätzen so zu speichern, dass die Operationen Suchen, Einfügen und Entfernen unterstützt werden.
- ▶ Jeder Datensatz ist gekennzeichnet durch einen eindeutigen Schlüssel, z.B. eine Integer-Zahl oder eine Zeichenkette.
- ▶ Zu jedem Zeitpunkt ist lediglich eine (kleine) Teilmenge σ aller möglichen Schlüssel Σ gespeichert.
- ▶ Das bedeutet bei einer linearen Speicherung der Datensätze mit einer Tabelle, wo der Index der Tabelle/des Arrays gleich (bzw. die Speicheradresse proportional) dem Schlüssel ist, eine große Tabelle mit geringer Zeilenausnutzung.
- ▶ Vergleich linearer und verteilter Speicherung von Datensätzen in Arrays:
 - ❑ Linear: Der Schlüssel des Datensatzes ist der Index
 - ❑ Verteilt (Scattered): Der Index in der Tabelle muss durch eine Abbildungsfunktion (Hashfunktion) vom Schlüssel abgeleitet werden.

Lineare Speicherung



Verteilte Speicherung



- Vorteil einer direkten Zuordnungsfunktion

$$F(k) = \text{Speicheradresse } A \sim k$$

liegt in der Suchzeit $O(1)$!

- Wünschenswert wäre eine indirekte Zuordnungsfunktion, die eine bessere Ausnutzung der Datentabelle ermöglicht:

$$H(k) = \text{Speicheradresse } A \neq k$$

$$H : k \rightarrow A$$

- Die Funktion H nennt man **Hash-Funktion**. Die Datensätze werden in einer linearen Tabelle (Array) abgelegt, die man Hash-Tabelle nennt.
- Die Größe der Hash-Tabelle ist zweckmäßigerweise $m < n = n_{k_{\max}}$

- Die Hash-Funktion ordnet jedem Schlüssel k einen Index der Tabelle (oder eine Speicheradresse) $h(k)$ mit $1 \leq h(k) \leq m$ im Falle eines Indexes zu.
- I.A. ist σ eine sehr kleine Teilmenge von Σ . Z.B. gibt es $26 \cdot 36^{79}$ mögliche Schlüssel für eine Zeichenkette mit 80 Zeichen, die mit einem Buchstaben beginnt, und mit Buchstaben oder Ziffern fortgesetzt wird.

□ Beispiel: Symboltabelle eines Compilers.

- Die Hash-Funktion besitzt die Eigenschaft, dass zwei verschiedene Schlüssel k_1 und k_2 die gleiche Hash-Adresse besitzen können:

$$H(k_1) = H(k_2)$$

k_1 und k_2 heissen dann Synonyme. Ergibt sich direkt aus der Tatsache $m < n$ und wird als Mehrdeutigkeit bzw. Adresskollision der Hash-Funktion bezeichnet.

- Die Adresskollision muss als Sonderfall getrennt behandelt werden!
- Eine gute Hash-Funktion sollte möglichst schnell berechenbar sein, und die zu speichernden Datensätze möglichst gleichmäßig, aber ohne erkennbare Struktur oder Ordnung auf den Speicherbereich verteilen, um Kollisionen zu minimieren.
- Problem: Die Schlüsselmenge $\sigma = \{k_i\}$ ist meistens nicht gleichmäßig verteilt.
- Z.B. bestehen Variablenamen in Programmen häufig aus 1-2 Zeichen mit den Buchstaben $\{x,y,z\}$ und den Zahlen $\{1,2,..\}$.
- Ein Maß für Wahrscheinlichkeit der Kollision in einer Tabelle mit m Zellen und nach k Einfügungen ist:

$$\gamma \approx 1 - \prod_{i=0}^{k-1} \frac{m-i}{m}$$

- Annahme: Gleichverteilung der Schlüssel!

Hashfunktion

- Die einfachste Hashfunktion für numerische ganzzahlige Schlüssel ist die Division-mit-Rest Funktion (mod):

$$H(k : \text{Natural}) = k \bmod m$$

JavaScript

In JavaScript sind Arrays immer Hash-Tabellen → Und Objekte (Strukturen) sind ebenfalls Hash-Tabellen → Der Schlüssel ist immer eine Zeichenkette (auch bei numerischen Index)!

Definition 33. (Hashtabelle)

```
var h; h[key]=data; x=h[y];
```

Beispiele

```
var a = [1,2,3,4]
var e1 = a[2];    -- == 3
var e2 = a["2"]; -- == 3!
a.counter = 4;
a["counter"]= 4;
var o = {x:1,y:2};
print(o.y-o.x)    -- 1
print(o["y"]-o["x"]) -- 1
```

Wörterbuch in JavaScript

```
function WBuch() {
  this.data={};
  this.n=0;
}
WBuch.prototype.finde = function (k) {
  return this.data[k];
}
WBuch.prototype.einfuegen = function (k,v) {
  this.data[k]=v; this.n++;
}
WBuch.prototype.loesche = function (k) {
  this.data[k]=undefined; this.n--;
}
function leeresWBuch() {
  return new WBuch();
}
```

Umgang mit Überläufern

1. Ein Eintrag in einer Hashtabelle ist eine lineare Liste. Die Überläufer werden in dieser Liste angeordnet.
 - ▶ Vorteil: Einfach zu implementieren
 - ▶ Nachteile: Suche in der Liste hat Laufzeitklasse $O(n)$! Listen sind dynamisch (Speichermanagement)
2. Die Überläufer werden in freien Zellen der Hashtabelle untergebracht:
 - ▶ Vorteil: Statische Tabelle
 - ▶ Nachteil: Sondierung erforderlich! Überläufer erhöhen wiederum Kollisionswahrscheinlichkeit!
 - ▶ Verfahren: Lineares und quadratisches Sondieren

7.4. Arrays

- ▶ Anders als Listen sind Arrays feste Blöcke im Speicher eines Computers.
- ▶ Analog zu den Listen lassen sich viele Elemente desselben Datentyps in einem Array speichern.
- ▶ Arrays lassen sich nicht vergrößern oder verkleinern, ohne sie neu anzulegen.
 - Daher muss zu Beginn bekannt sein, wie viele Elemente maximal gespeichert werden sollen.
- ▶ Dieser Nachteil führt dazu, dass Arrays in Haskell seltener zum Einsatz kommen als Listen.
- ▶ Vorteil gegenüber Listen: Die **Zugriffszeit** auf ein Array-Element ist immer **konstant**. Um auf das k -te Element zuzugreifen, müssen nicht die ersten $k-1$ Elemente entlanggelaufen werden, wie es bei einer verketteten Liste der Fall ist!

Dynamische Arrays

- ▶ Für Hashtabellen werden veränderliche Arrays benötigt.

- Traditionell sind Arrays vor allem ein Zugeständnis an die Architektur der von Neumann-Rechner: Sie repräsentieren einen konsekutiven Speicherbereich, dessen Elemente sehr effizient und kompakt über Adressrechnung verarbeitbar sind.
- Ein Array ist ein i.A. monosortiger Produkttyp und besteht aus einer Anzahl N von Elementen die über einen abzählbaren Index selektiert werden können.
- Es gibt Schreib- und Leseoperationen auf Arrays bzw. deren selektierte Elemente:

```

a = ARRAY (ia .. ib) typelem ([v0,...])
write a index value
x = read a index

```

- Hat man ein Block- und Speichermodell, können Arrayoperationen und Elementzugriffe auf Blockoperationen abgebildet werden:

Das Array-Konstrukt wird übersetzt in Speicherblock
$Array(a..b)\alpha$	$Block(n)\alpha$ WHERE $n = b - a + 1$
$v(i)$	$get(block, i - a)$
$v\ i \leftarrow x$	$set(block, i - a, x)$
$step$	$(+1), (-1)$ etc.
$v = \lambda i \bullet f(v, i)$	$iterate(...)$

Abb. 9. Übersetzung von Array- auf Blockoperationen [B]

- Der Typ Array $(a..b)\alpha$ wird in einen Block der Länge $b-a+1$ konvertiert.
- Die Array-Selektion $v(i)$ wird auf die Blockselektion abgebildet, wobei ein Indexshift in das Intervall $0..n-1$ vorgenommen wird.
 - Die Einhaltung der Indexgrenzen $a \leq i \leq b$ ist Teil der Typprüfung, die vom Compiler generell eingebaut wird und deshalb hier nicht noch einmal angegeben werden muss.
- Der Array-Update wird entsprechend auf set abgebildet.
- Die Operation $step$ kodiert die Richtung und Schrittweite, in der das Array verarbeitet werden soll

- Die Erzeugung eines Arrays über einen Ausdruck, also $v = \lambda i \bullet f(v, i)$, wird sequenziell über einen Iterator gelöst, der mit Hilfe der Operation *step* die Arrayelemente nacheinander mit den entsprechenden Werten von $f(v, i)$ besetzt.
- Was wäre bei einem ‘rohen’ Speichermodell noch zu beachten?
- Funktional gibt es *keine* veränderlichen Daten!
- Daher würde strikt funktional die Änderung eines Arrayelements die (virtuelle) Erzeugung eines neuen Arrays bedeuten:

```

FUN read:  $\alpha$  Array  $\rightarrow \beta \rightarrow \alpha$ 
FUN write:  $\alpha$  Array  $\rightarrow \beta \rightarrow \alpha \rightarrow \alpha$  Array

```

- Auf semantischer und methodischer Ebene **sind Arrays nichts anderes als spezielle Funktionen**:
 - ❑ Arrays sind spezielle Funktionen. Sie haben die beiden folgenden Besonderheiten, von denen die erste semantisch ist, die zweite pragmatisch.
 - ❑ Der Definitionsbereich eines Arrays ist aus Intervallen der ganzen Zahlen gebildet. (Der Wertebereich ist beliebig.)
 - ❑ Arrays sind ‘eingefrorene’ Funktionen; d.h., sie liegen jeweils in einer tabellarischen Auflistung ihrer Werte vor. (Man kann das als eine Art genereller Memoization auffassen.)
- Aber: Es gibt in Haskell auch ein Modul für veränderliche Arrays:

Definition 34.

```

import Data.Array.IO
.. do ..
  a <- newArray (ia, ib) v0
           :: IO (IOArray typindex typelem )
  writeArray a index value
  readArray a index

```

- Die Grenzen eines Arrays sind nicht wie in vielen anderen Sprachen auf den Datentyp *Int* beschränkt.
- Alle Typen aus der Typklasse *Ix*, insbesondere auch Tupel, können als Arraygrenzen verwendet werden.

- So können problemlos mehrdimensionale Arrays verwaltet werden, beispielsweise um Digitale Verarbeitung von Bildern zu ermöglichen.

Statische Arrays

- Bei statischen Arrays können die Elemente nicht verändert werden.
- Arrays müssen in funktionalen Sprachen wie andere (symbolische) Variablen mit einem Wert initialisiert werden.
- Durch den Liste-Array Dualismus können **statische Arrays** auch aus Listen erzeugt werden:

```
import Data.Array.IArray
.. do ..
  a = (listArray (ia, ib) [..]) :: Array typindex typelem
  a!index
```

- In Haskell kann auf ein Arrayelemente vereinfacht mittels des *a!index* Operators zugegriffen werden (nur lesend!)

Beispiele in Haskell

7.5. Listen

- Eine Liste ist eine lineare Verkettung von Listenelementen (Daten).
- Ein Listenelement besteht dabei aus einem Verkettungs-, Schlüssel und Datenteil:

```
TYPE ( $\alpha, \beta$ ) LISTENELEMENT = {
  connect: [ ( $\alpha, \beta$ ) LISTENELEMENT ],
  key:  $\beta$ ,
  data:  $\alpha$ 
}
```

- Der Schlüssel ist optional assoziiert einen Datensatz (eindeutige oder mehrdeutige Assoziation)
- Listen unterstützen folgende wesentliche Operationen:
 - ❑ **Einfügen** eines Elements an einer bestimmten Position in der Liste
 - ❑ **Löschen** eines Elements an einer bestimmten Position in der Liste

- **Suchen** eines Elements entweder anhand Position oder Schlüssel
- **Spezielle Operation:** Einfügen/Löschen eines Elements am Kopf/Ende
- Listen können **statisch** mit Arrays und **dynamisch** durch Verkettung von Elementen implementiert werden.
- Es gibt dabei zwei Arten von Listen die sich durch die Art der Verkettung unterscheiden:

Einfach Verkettete Liste

Jedes Element besitzt eine Referenz (Link) zu dem nächsten rechten (oder linken) Nachbarn

Doppelt Verkettete Liste

Jedes Element besitzt zwei Referenzen zum jeweiligen linken und rechten Nachbarn

- Einfach verkettete Listen sind ein Kerndatentyp von Haskell und werden praktisch von allen funktionalen Programmiersprachen unterstützt.

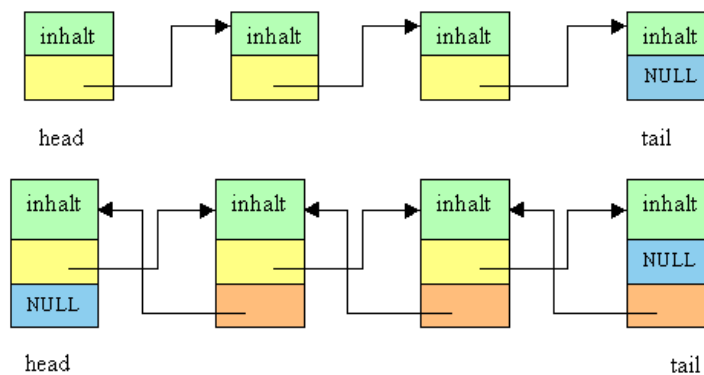


Abb. 10. Aufbau einer einfach und doppelt verketteten Liste

Doppelt verkettete Listen in Haskell

- Annahme: Jeder Datensatz besitzt einen assoziierten Schlüssel

```

data DList a b =
  Leaf |
  Node { prev::(DList a b), key:: a, elt::b, next::(DList a b) }

create = go Leaf
  where go _ [] = Leaf
        go prev ((key,val):xs) = current
          where current = Node prev key val next
                next    = go current xs

findRight _ Leaf = Nothing
findRight key (Node l k v r) =
  if k == key then Just v
  else findRight key r

findLeft _ Leaf = Nothing
findLeft key (Node l k v r) =
  if k == key then Just v
  else findLeft key l

```

- Bei einer einfach verketteten Liste muss immer das **Kopfelement** gespeichert werden, bei einer doppelt verketteten Listen können sowohl Kopf- als auch Endelement gespeichert werden
- Kopf- und Endelemente (Wurzelknoten) sind der Zugang zur Liste, z.B. beim Suchen

Laufzeiteigenschaften

- Laufzeit als Laufzeitklasse $\rightarrow \Theta(x) \rightarrow$ Größenordnung der groben Berechnungsschritte in Abhängigkeit von der Größe x der Datenstruktur
- Das Einfügen oder Löschen eines Elements am Kopf der Liste (und am Ende bei dopp. verk.) benötigt nur einen Schritt $\rightarrow \Theta(1) \rightarrow$ konstante Laufzeit unabhängig von n (Anzahl der Knoten)
- Das Suchen eines Elements benötigt bei beiden Listen maximal n Schritte $\rightarrow \Theta(n)$
- Das Einfügen oder Löschen eines Elements anhand eines Schlüssel innerhalb der Liste benötigt maximal $(n-1)$ Schritte (Suche) $\rightarrow \Theta(n)$
- Bei der doppelt verketteten Liste ist aber Einfügen an beliebiger Stelle mit konstanter Laufzeit möglich wenn das Element an der entsprechenden Position bereits bekannt ist $\rightarrow \Theta(1)$.

- Bei der einfach verketteten Liste kostet aber das Einfügen immer $\Theta(n)$. Warum?

Weitere Listenoperationen

- Haskell bietet im Modul `Data.List` zwei weitere wichtige Operationen auf Listen an:

minimum $[\alpha] \rightarrow \alpha$

Findet das kleinste Element einer Liste und gibt dieses als Ergebnis zurück (Wert, nicht Position!)

delete $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$

Löscht ein Element einer Liste. Der Wert des zu löschenden Elements muss angegeben werden, nicht die Position! Die modifizierte Liste wird zurück gegeben.

7.6. Sortieren von Listen

SelectionSort

- Der vermutlich einfachste Ansatz, eine Liste mit Elementen zu sortieren, ist
 - das kleinste Element aus der Liste zu entfernen,
 - es an die Ergebnisliste zu hängen und rekursiv fortzufahren
 - bis keine Elemente mehr übrig sind.
- Dieser Algorithmus wird als Selection Sort oder Sortieren durch Auswahl bezeichnet. Iterativ können wir den Algorithmus in etwa so angeben:

```
FOR i = 0 to n-1 DO
  FOR j=n-1 to i+1 DO
    IF x[j-1] > x[j] THEN vertausche x[j-1] und x[j]
```

- In Haskell kann man unter Verwendung von `minimum` und `delete` die Sortierung rekursiv formulieren:

```

import Data.List
sSort :: (Ord a) => [a]->[a]
sSort [] = []
sSort xs = m : sSort (delete m xs)
  where m = minimum xs

```

► Laufzeitanalyse mittels Rekursion:

- Das Finden des Minimums einer Liste ist in linearer Zeit zu schaffen.
- Das Finden und damit auch das Löschen eines beliebigen Elements benötigt ebenfalls lineare Zeit.
- $T(0) = 1$
- $T(n) = n + T(n-1)$
- $\Rightarrow \Theta_{\text{sSort}}(n^2)$

Zeitpunkt

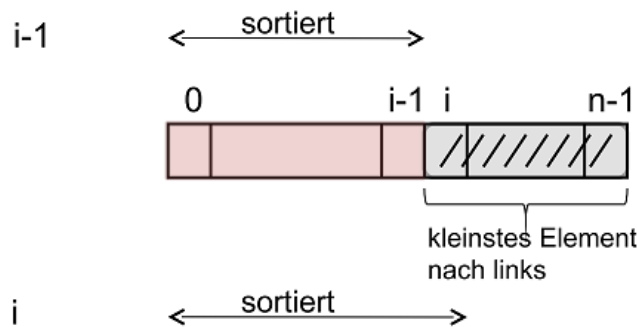
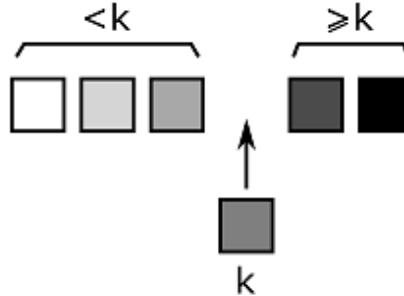


Abb. 11. Der Übergang von Zeitpunkt $i-1$ zu i im SelectionSort-Algorithmus und die daraus resultierende erweiterte, sortierte Liste [C]

InsertionSort

- Ein weiterer Sortieralgorithmus ist das „Sortieren durch Einfügen“ oder kurz InsertionSort.
- Angenommen, eine Liste liegt bereits aufsteigend sortiert vor, dann ist das Einordnen eines neuen Elements einfach.
- Es wird solange von links beginnend überprüft, ob das aktuelle Element kleiner oder gleich dem Einzufügenden ist, bis die korrekte Position ermittelt ist.

- An diese Stelle wird das Element eingefügt und die Liste bleibt sortiert



- Die Sortierung mit *InsertionSort* beginnt mit einer leeren Liste und fügt mit der Funktion einfügen nacheinander alle Elemente aus der zu sortierenden Liste ein:

```

einfuegen :: Ord a => a -> [a]-> [a]
einfuegen x [] = [x]
einfuegen x (y:ys)
  | x<=y = x:y:ys
  | otherwise = y:(einfuegen x ys)

iSort :: Ord a => [a]-> [a]
iSort [] = []
iSort (x:xs)= einfuegen x (iSort xs)

```

- **Laufzeitanalyse:**

- ❑ Im schlechtesten Fall benötigt dieses Verfahren ebenfalls eine quadratische Laufzeit, da das Auffinden der richtigen Stelle zum Einfügen im schlechtesten Fall lineare Zeit benötigt und die Liste für die Rekursion nur um eins verkürzt wird.
- ❑ Im besten Fall ist InsertionSort, anders als bei SelectionSort, allerdings sehr viel schneller.
- ❑ Der best-case tritt ein, wenn die Startliste umgekehrt sortiert vorliegt. Dann ist die richtige Stelle zum Einfügen immer ganz vorn zu finden und das benötigt konstante Zeit. Es wird also n -mal ein konstanter Aufwand betrieben, so dass $\Theta(n)$ Operationen gebraucht werden.

QuickSort

- Der QuickSort-Algorithmus ist ein typisches Beispiel für Algorithmen die auf dem Teile-und-Herrsche-Prinzip basieren → **Top-down Ansatz**
- In jedem Schritt, beginnend mit der gesamten Liste, wird ein Element ausgewählt → Pivotelement
- Die Liste wird dann dort zweigeteilt.
 - ❑ In der linken Liste sind die Elemente enthalten, die kleiner oder gleich dem Pivotelement sind und
 - ❑ in der rechten die größeren Elemente.
- Das Pivotelement verbleibt dabei in der Mitte und hat seinen endgültigen Platz in der sortierten Liste bereits eingenommen.

Die Auswahl des Pivotelements ist zentral in diesem Algorithmus und kann auf verschiedene Weisen erfolgen. Z.B.: Immer Auswahl des ersten Elements einer Teilliste

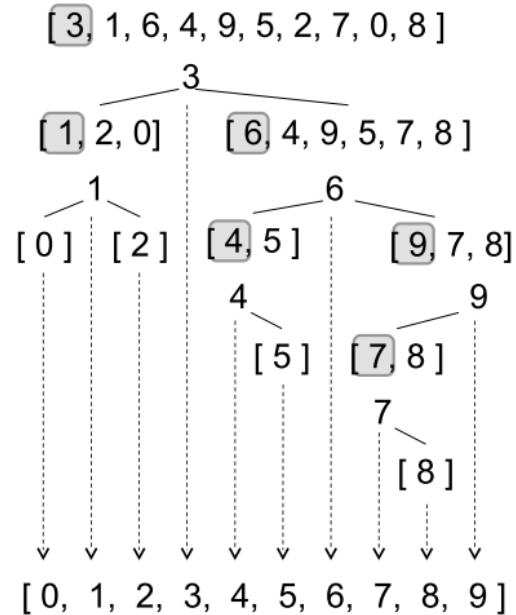


Abb. 12. Beispiel einer QuickSort Sortierung mit Auswahl der Pivotelemente und Teilung der Listen [C]

- In Haskell lässt sich der QuickSort-Algorithmus durch die einfache automatische Erzeugung von Listen wieder rekursiv mit einigen wenigen Zeilen formulieren:

```
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (x:xs) =
  qSort [y | y<-xs, y<=x] ++ [x] ++
  qSort [y | y<-xs, y> x]
```

- Die **Laufzeit** hängt dabei stark davon ab, wie das Pivotelement gewählt wird. Im besten Fall wird die Liste in jedem Rekursionsschritt halbiert.
 - ❑ $T(0) = 1$
 - ❑ $T(n) = n + 2 T(n/2)$
 - ❑ Im **besten Fall** erreicht man $\Theta(n \log(n))$
- Im schlechtesten Fall gehört das Pivotelement nicht in die Mitte der sortierten Liste, sondern an einen der Ränder.
- Dann benötigt man in jedem Rekursionsschritt weiterhin einen linearen Aufwand, aber die Größe des rekursiven Aufrufs wird nur um eins reduziert, genau wie bei Selection- und InsertionSort.
- Im schlechtesten Fall fällt die Laufzeit von QuickSort also auf $\Theta(n^2)$
- Aber: Parallelisierung durch Map-Reduce Partitionierung einfach möglich!!!
 - ❑ Es gibt keine tiefere Datenabhängigkeit zwischen den Teillisten und weiteren Sortierungen → geringer Kommunikationsaufwand!

MergeSort

- QuickSort ist ein effizientes Sortierverfahren, hat jedoch im schlechtesten Fall quadratische Laufzeit.
- MergeSort ist ein weiteres Sortierverfahren, das dem Teile-und-Herrsche-Prinzip folgt → aber **Bottom-up Ansatz**.
- Annahme: Es ist kostengünstig, aus zwei sortierten Listen eine neue sortierte Liste zu erzeugen.

- Dazu werden nur die jeweils ersten Elemente der beiden sortierten Listen verglichen und das kleinere von beiden in die Ergebnisliste geschrieben.
- Beim MergeSort-Algorithmus werden zunächst einelementige Listen rekursiv erzeugt und diese sortierten Listen anschließend mit zusammengefasst, bis nur noch eine Liste vorhanden ist

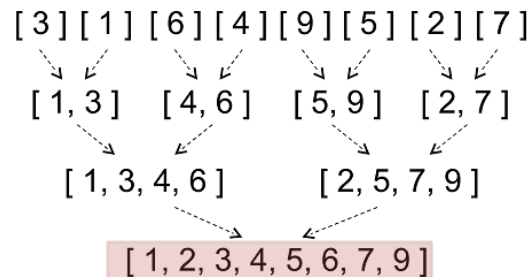


Abb. 13. Beispiel einer schrittweisen Zusammenführung von kleinen Teillisten zu größeren [C]

- In Haskell lässt sich MergeSort wieder rekursiv definieren.
- Die Idee von MergeSort ist, die zu sortierende Liste zu halbieren, die beiden Hälften rekursiv zu sortieren und die sortierten Teile wieder zusammenzufügen:

```

merge [] ys = ys
merge xs [] = xs
merge (x:xs)(y:ys)
  | x <= y = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)

mSort :: Ord a => [a]-> [a]
mSort [] = []
mSort [x]=[x]
mSort xs = merge (mSort erste)(mSort zweite)
  where
    (erste, zweite) = splitAt haelfte xs
    haelfte = div (length xs) 2
  
```

- **Laufzeitanalyse:**
 - Die Laufzeit der *merge* Funktion ist linear zu der Länge der beiden Listen.

- Wenn die Listen die Längen m und n haben, ist die Laufzeit $\Theta(m+n)$.
- Da bei diesem Verfahren die Listen *garantiert halbiert* werden und in einem Rekursionsschritt nur linearer Aufwand betrieben wird, ist die Laufzeit **auf jeden Fall** $\Theta(n \log(n))$

7.7. Stack

(Stapelspeicher / Kellerspeicher)

- Ein Stack ist eine lineare Liste (oder Array) mit zwei speziellen Operationen, die das Hinzufügen und Entfernen von Elementen in Last-In First-Out Reihenfolge ermöglicht:

push $s\ el \rightarrow s$

Ein Element el wird auf dem Stapelspeicher als oberstes Element abgelegt (top)

pop $s \rightarrow (el, s)$

Das oberste Element wird vom Stapelspeicher entfernt

- Stapelspeicher finden bei Funktionen und Funktionsaufrufen Anwendung
 - Auf dem Stapel werden bei einem Funktionsaufruf i.A. die Funktionsparameter abgelegt, der Rückgabewert der Funktion, und interne Informationen die die bei der Terminierung der Funktion erforderlich sind.

- Ein Stapelspeicher unterstützt keine Konkatination!

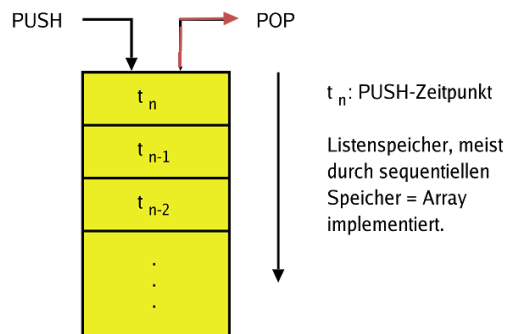


Abb. 14. Einfügen und Entfernen von Elementen bei einem Stapelspeicher

- Stapelspeicher kann direkt mit Haskell Listen implementiert werden
- Laufzeit für Einfügen und Entfernen ist immer $O(1)$

```

data Stack a = S [a]
create () = []
push (S s) el = el:s
pop S (el:tail) = (el,tail)

```

7.8. Warteschlange (Queue)



- Eine Warteschlange ist ebenfalls eine Liste mit speziellen Operationen die das Einfügen und Entfernen von Elementen in First-In First-Out Reihenfolge ermöglicht

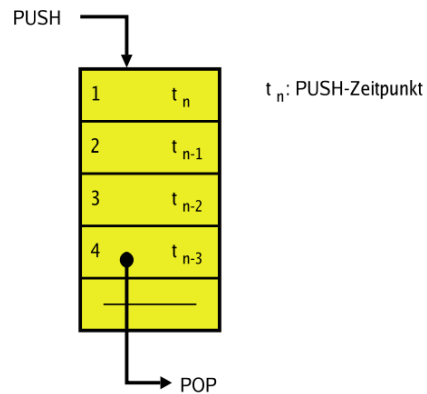
enqueue $el \rightarrow q$

Ein Element am Ende der Liste einfügen

dequeue $q \rightarrow (el, q)$

Ein Element vom Anfang der Liste (Kopf) entfernen

- Als abstrakter Datentyp ist eine Warteschlange ein Containertyp *Queue* α , der mehrere Elemente des Typs α enthalten kann.
- Warteschlangen sind für viele bekannte Algorithmen wichtig.
- Als erste Idee für die Implementierung einer Warteschlange könnte der Datentyp *Liste* angedacht werden.
 - ❑ Der Test auf eine leere Liste und das Entfernen des ersten Elements funktionieren bei Listen schnell.
 - ❑ Um ein Element an eine Liste anzuhängen, benötigen wir allerdings eine Laufzeit von $\Theta(n)$



```
class Queue a where
  -- Test auf leere Warteschlange
  empty :: a b -> Bool
  -- Einreihen eines Objekts am Ende
  enqueue :: b -> a b -> a b
  -- Entfernen des Kopfes
  dequeue :: a b -> (b, a b)
```

- Alternativ können zwei Listen verwendet werden.
 - ❑ Eine Liste bildet dabei den Anfang einer Warteschlange ab und ermöglicht so eine effiziente Entfernung eines Elements.
 - ❑ Durch die zweite Liste wird das effiziente Anhängen eines Elements realisiert.
- Wenn ein Element aus der Warteschlange herausgenommen wird, kann es das letzte Element der ersten Liste sein.
 - ❑ Dann müssen die Elemente aus der zweiten Liste umgekehrt an die Stelle der ersten Liste geschrieben werden!

```

data Queue a = Q [a] [a]

empty :: Queue a -> Bool
empty (Q [] []) = True
empty _ = False

create () = Q [] []

enqueue :: a -> Queue a -> Queue a
enqueue a (Q x y) = Q x (a:y)

dequeue :: Queue a -> (a, Queue a)
dequeue (Q [] y) = (y1, Q ys [])
  where (y1:ys) = reverse y
dequeue (Q [x] y) = (x, Q (reverse y) [])
dequeue (Q (x:xs) y) = (x, Qxs y)

```

- Die Laufzeit der Zweilistenimplementierung ist nicht generell konstant, denn im schlechtesten Fall hat *dequeue* einen linearen Aufwand $O(n)$, um die zweite Liste umzudrehen.
- Anschließend kann *dequeue* allerdings n -mal mit konstantem Aufwand aufgerufen werden.
 - Es scheint als würde der gelegentlich auftretende, höhere Aufwand damit ausgeglichen.

Amortisierte Laufzeitanalyse

- Da die Operationen *enqueue* und *dequeue* aber im Mittel gleich häufig verwendet werden ist eine **amortisierte** Laufzeitanalyse sinnvoll, d.h. die mittleren Kosten bei Verwendung der beiden Operationen
- Man kann die Laufzeit einer Sequenz der Länge n von *enqueue* und *dequeue* Funktionsaufrufen betrachten.
 - Wenn die Gesamtlaufzeit $O(f(n))$ ist, dann ist es sinnvoll, die amortisierte Laufzeit der einzelnen Aufrufe als $O(f(n)/n)$ anzugeben.

Bankiermethode

Eine Möglichkeit die amortisierten Laufzeiten zu bestimmen, ist die so genannte Bankiermethode (banker's method).

- Dabei werden den Operationen nicht nur die tatsächlichen Kosten zugewiesen, wie es bei der normalen Laufzeitanalyse gemacht wird.
- Es werden zusätzlich noch Kredite hinterlassen oder vorhandene Kredite konsumiert.
- Jeder *enqueue* Aufruf benötigt Laufzeit $O(1)$ und hinterlässt einen Kredit in Höhe 1, was die amortisierten Kosten auf 2 erhöht!
- Ein *dequeue* Aufruf mit nicht leerer erster Liste benötigt nur $O(1)$
- Ein *dequeue* Aufruf mit leerer erster Liste erfordert *reverse* und benötigt $O(1)$ um ein Element zu entfernen und $O(m)$ um die Liste umzudrehen
- Die teure Operation verbraucht m Kredite!
- Die amortisierte Laufzeit der Warteschlange wäre dann $O(1)$!!! Aber Datenstrukturen sind in Haskell persistent, d.h. unveränderlich, und in der Realität kann die amortisierte Laufzeit größer sein!

Lazy Evaluation

Lazy Evaluation: Ausdrücke werden nur bei Bedarf ausgewertet bzw. Operationen ausgeführt

- Hier: Kritisch, dass es bereits zu spät ist, die zweite Liste erst dann umzudrehen, wenn die erste Liste bereits leer ist
- Stattdessen muss dafür gesorgt werden, dass immer genügend Elemente vorhanden sind.
 - ❑ Dazu werden zwei Zähler eingeführt, die die Länge der beiden Listen beinhalten.
 - ❑ Wann immer die zweite Liste länger wird als die erste, drehen wird sie umgedreht und hinten an die erste Liste angehängt.

```

data Queue a = Q Int [a] Int [a]

empty :: Queue a -> Bool
empty (Q 0 _ _ _) = True
empty _ = False

queue n x m y
  | n > m     = Q n x m y
  | otherwise = Q (n+m) (x++reverse y) 0 []

enqueue :: a -> Queue a -> Queue a
enqueue a (Q n x m y) = queue n x (m+1) (a:y)

dequeue :: Queue a -> (a, Queue a)
dequeue (Q n (x:xs) my) = (x, queue (n-1) xs m y)

```

- Auch bei einem persistenten Gebrauch der Listen ist die amortisierte Laufzeit konstant!

7.9. Bäume

- Die Datenstruktur Baum liegt vielen Algorithmen zu Grunde.
- Ein Baum ist ein spezieller Graph (zyklenfrei)
- Bäume bestehen aus
 - ❑ Knoten (enthält Daten) und
 - ❑ Kanten (realisiert die Struktur).
- Der Startknoten wird als Wurzel bezeichnet.
- Daten können nur in den Endknoten (Blättern) enthalten sein → die Zwischenknoten dienen nur zum Auffinden der Daten
- In den Knoten und an den Kanten können bei Bedarf zusätzliche Informationen gespeichert werden.
- Alle kreisfreien Datenstrukturen, wie beispielsweise Listen, lassen sich durch Bäume repräsentieren.

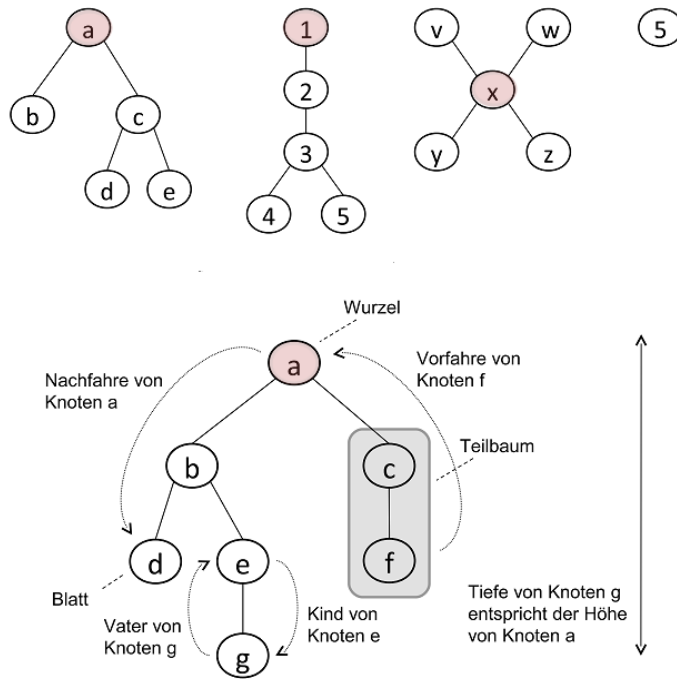


Abb. 15. (Oben) Verschiedene Bäume (Unten) Terminologie der Bäume

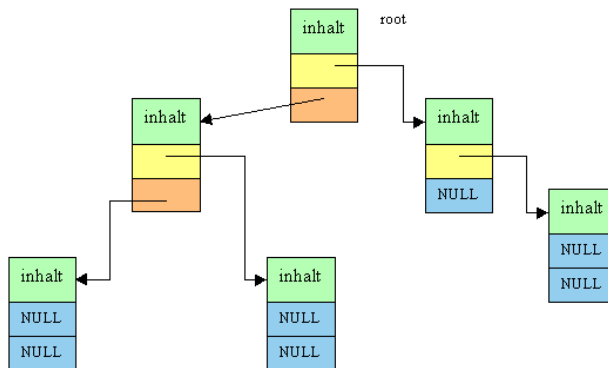


Abb. 16. Aufbau und Verkettung eines Binärbaumes

- Interessant sind schon wie bei den vorhergehenden Datenstrukturen das Einfügen und das Löschen von Elementen und deren Laufzeitkosten.
- **Beide Operationen lassen sich auf das Zusammenfügen von Bäumen zurückführen.**
- Wenn man ein Element in einen existierenden Baum einfügt, wird ein Baum mit einem Element zu einem neuen Baum zusammengeführt.

- Wenn ein Element gelöscht wird, dann erhält man zunächst viele Bäume die wieder miteinander sinnvoll zu einem neuen Baum verbunden werden müssen.
- Abstrakter Datentyp Baum:

```
data Baum a =
  Nil
  | Knoten { key :: a, inhalt :: b, lT, rT :: (Baum a) }
```

- Wichtiges Kriterium eines Baumes ist seine Höhe (die Anzahl der Ebenen):

```
height :: Baum -> Int
height Nil = 0
height x = max (height (lT x)) (height (rT x)) + 1
```

- Nur ein balanzierter Baum hat die geringste Höhe
- Die größte Höhe entsteht beim Grenzfall lineare Liste!
- Bei vielen Anwendungen ist es wichtig, dass die Höhe des Baumes möglichst gering bleibt, dieser also bestmöglich balanciert ist.
- Suche eines Elements kostet beim balanzierten Binärbaum $\Theta(\log n)$!
 - ❑ Bei der Suche kann ein abzählbarer und eindeutiger Schlüssel verwendet werden
 - ❑ Ist der aktuelle Knoten nicht der gesuchte, wird im Binärbaum der linke Teilbaum durchsucht wenn $key < key_i$, ansonsten der rechte Teilbaum bei $key > key_i$

```
search :: Baum -> a -> Maybe b
search Nil _ = Nothing
search x k
  | (key x) == k = x
  | (key x) < k = (lT x)
  | (key x) > k = (rT x)
```

- Für ein Zusammenfügen von Bäumen kann die Speicherung der Anzahl der Teilknoten *anzahl* sinnvoll sein.

```
data Baum a =
  Nil
  | Knoten { anzahl :: Int, key :: a, inhalt :: b,
            lT, rT :: (Baum a) }
```

- Die Rekursionstiefe des Verschmelzens von Bäumen ist abhängig von der Tiefe des entstehenden Baums und umgekehrt.
- Beim Zusammenfügen sollte möglichst gleich ein balanzierter Baum entstehen → bestenfalls mit Laufzeit $\Theta(\log n)$. Dazu wird der Parameter *anzahl* verwendet.

```
merge :: Baum a -> Baum a -> Baum a
merge Nil x = x
merge x Nil = x
merge x y |
  size x >= size y = x {
  elems = size x + size y,
  -- das kleine Kind rekursiv mergen
  lT = (merge s y),
  -- das größere Kind nicht anfassen
  rT = b}
| otherwise = merge y x where
(s,b) =
  if (size (lT x) <= size (rT x)) then (lT x, rT x)
  else (rT x, lT x)
```

Balancierung

- Nachträgliche Balancierung durch Kombination aus Rechts-Links Rotationen möglich

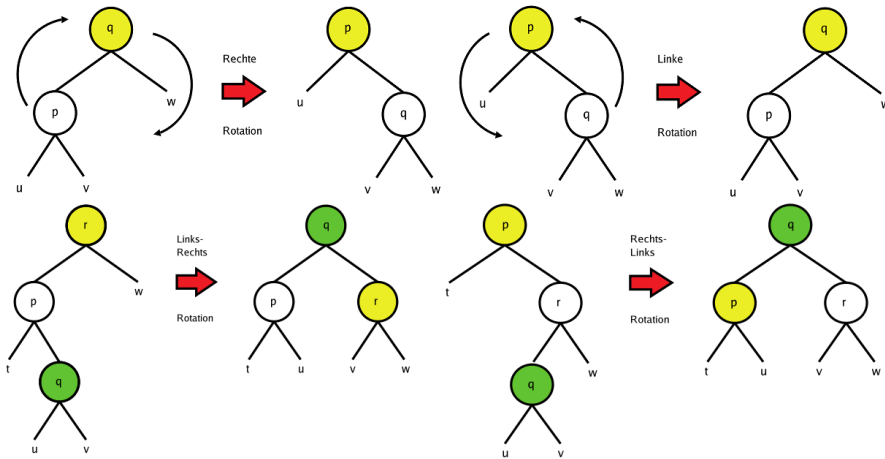


Abb. 17. (Oben) Grundoperationen rechte Rotation im Uhrzeigersinn und linke Rotation gegen den Uhrzeigersinn (Unten) Kombinationen

7.10. Graphen

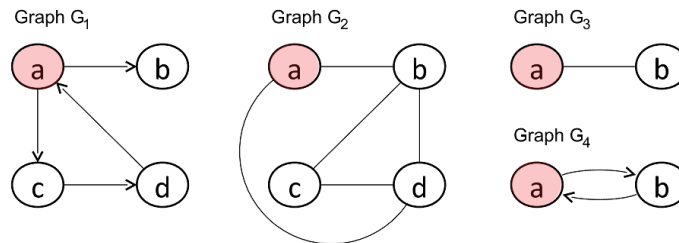


Abb. 18. Verschiedene Graphen: (G_1, G_4) Gerichtete Graphen (G_2, G_3) Ungerichtete Graphen (G_1, G_2, G_4) Zyklische Graphen mit Schleifen

8. Lost & Found

8.1. Laufzeiteigenschaften von Algorithmen

- Performanz von Algorithmen kann auf zwei Arten bestimmt werden:
 - ❑ Experimentell: Laufzeitmessung unter realen Bedingungen → Abhängig von Rechnerarchitektur und Eingabedaten (Monte Carlo Simulation erforderlich) → von Computer zu Computer unterschiedlich usw.

- Analytisch: Abschätzung der Anzahl der primitiven Operationen die ein Prozessor durchführen muss
- Was genau als primitive Operation festgelegt wird, ist im Allgemeinen nicht wichtig.
- Stattdessen geht es um die Abschätzung der Laufzeit in Abhängigkeit von der *Größe der Eingabedaten* $N \rightarrow$ **Problemgröße**.
- Die Laufzeitanalyse ist wichtig, um möglichst effiziente Lösungen zu finden. Das ist trotz der immer schneller werdenden Computer sehr wichtig. Vergleichen

Vergleich der Laufzeitklassen

- Eine gute Laufzeit(klasse) ist wichtiger als ein schneller Computer!

Laufzeit	$n = 5$	$n = 10$	$n = 10^2$	$n = 10^3$	$n = 10^6$
$\log n$	20 μs	30 μs	60 μs	100 μs	200 μs
n	50 μs	100 μs	1 ms	10 ms	10 s
$n \cdot \log n$	120 μs	330 μs	7 ms	100 ms	200 s
n^2	250 μs	1 ms	100 ms	10 s	116 d
$n!$	1 ms	36 s	$3 \cdot 10^{145}$ a	∞	∞
2^n	330 μs	10 ms	$3 \cdot 10^{17}$ a	∞	∞

Abb. 19. Exemplarische Laufzeiten unterschiedlich schneller Algorithmen bei jeweils 10^5 Operationen pro Sekunde [C]

8.2. Landau-Symbole

Obere Schranken O

Eine Funktion f ist eine asymptotische obere Schranke einer Funktion g , wenn es eine Stelle n_0 und eine Konstante $c > 0$ gibt, so dass für alle natürlichen Zahlen $n > n_0$ stets $g(n) \leq c \cdot f(n)$ gilt.

Mit Quantorennotation kann man schreiben: $\exists n_0, \exists c, \forall n > n_0 : g(n) \leq c \cdot f(n)$

- Das bedeutet, dass ab dem Wert n_0 für alle folgenden n gilt: $g(n)$ ist immer kleiner/ gleich $c \cdot f(n)$.

► Es gibt für ein $g(n)$ eine sehr große Menge von Funktionen $f(n)$, die diese Bedingung erfüllen.

□ Eine asymptotische obere Schranke ist definiert durch die Menge $O(f)$.

Untere Schranken Ω

Im umgekehrten Fall kann man eine asymptotische untere Schranke definieren. Man schreibt $g \in \Omega(f)$, wenn es eine Stelle n_0 und eine Konstante $c > 0$ gibt, so dass gilt:

$$g \in \Omega(f) \Leftrightarrow \exists c, \exists n_0, \forall n \geq n_0 : g(n) \geq c \cdot f(n)$$

Starke obere Schranken o

Die Definition ist ganz ähnlich, lediglich die Konstante c kann nun nicht mehr frei gewählt werden. Die Ungleichung muss für alle c gelten:

$$g \in o(f) \Leftrightarrow \forall c, \exists n_0, \forall n > n_0 > 0 : g(n) \leq c \cdot f(n)$$

Asymptotisch gleiches Wachstum Θ

Es gibt auch eine Notation, um auszudrücken, dass zwei Funktionen asymptotisch gleich schnell wachsen. Das ist genau dann der Fall, wenn sie sowohl obere als auch untere Schranke voneinander sind:

$$g \in O(f) \wedge g \in \Omega(f) \Leftrightarrow g \in \Theta(f)$$

Umgang mit Schranken und Regeln

1. Multiplikation mit Konstanten: Für alle $k > 0$ gilt: $k \cdot f(n) \in \Theta(f(n))$
2. Summen und Produkte. Es gilt:
 $f_1(n) \in O(g_1(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
3. Potenzen. Größere Potenzen sind starke obere Schranken von kleineren Potenzen. Es gilt:
 $a < b \Rightarrow n^a \in o(n^b)$
4. Polynome. Es gilt:
 $(n+a)^b \in \Theta(n^b)$

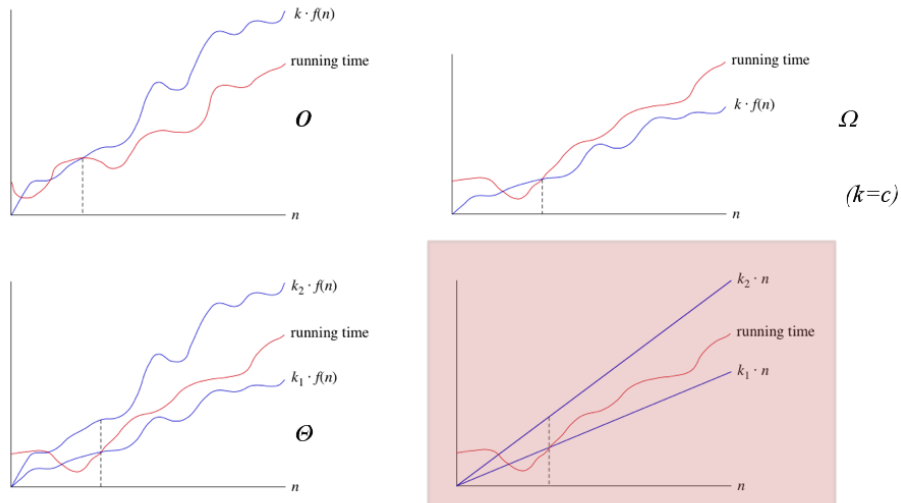
5. Logarithmen. Alle Logarithmen (mit einer Basis > 1) unterscheiden sich nur um eine Konstante, daher gilt:
 $\log_a n \in \Theta(\log_b n)$
6. Logarithmen gegen Potenzen: Potenzen von Logarithmen wachsen langsamer als normale Potenzen. Es gilt:
 $(\log n)^a \in o(n^b)$
7. Potenzen gegen Exponentialfunktionen: Potenzen wachsen immer langsamer als Exponentialfunktionen. Es gilt:
 $n^a \in o(b^n)$

Funktion	Beispiel	Typische Bezeichnung
1	Nachsehen in einem Array	konstant
$\log n$	Teile-und-Herrsche	logarithmisch
$\log \log n$	Suche in sortierten Arrays	-
$\log^k n$	-	polylogarithmisch
\sqrt{n}	Naiver Primzahltest	-
n	Maximum einer Liste bestimmen	linear
$n \log n$	Vergleichsbasiertes Sortieren	-
n^2	Alle Paare einer Grundmenge ansehen	quadratisch
n^3	Lineare Gleichungssysteme lösen	kubisch
n^k	-	polynomiell
2^n	Alle Teilmengen untersuchen	exponentiell
$n!$	Alle Permutationen untersuchen	faktoriell

Abb. 20. Häufig vorkommende Laufzeiten und deren Bezeichnungen [C]

8.3. Landau-Symbole

Vergleich der Schranken



8.4. Laufzeitanalyse

Best, Worst und Average Case

- Bei der Analyse von Algorithmen wird zwischen verschiedenen Fällen unterschieden. So dauert z.B. die Suche in Listen unterschiedlich lange in Abhängigkeit von der Position.
- Weiterhin kann Optimierung wie **Lazy Evaluation** oder **Memoization** das tatsächliche Laufzeitverhalten (Klasse) nach unten beeinflussen
 - ❑ Die Funktion $f\ n = [1..n]$ scheint $\Theta(n)$ viele Schritte zu benötigen. Wird sie im Kontext $g\ n = \text{head}\ (f\ n)$, wird sie eventuell mit $\Theta(1)$ optimiert abgearbeitet, da nur das erste Element benötigt wird und die restliche Liste gar nicht erst produziert wird (Lazy Evaluation).
- Man unterscheidet drei Fälle:
 - ❑ **Best case.** Das Element befindet sich gleich am Anfang.
 - ❑ **Worst case.** Im schlechtesten Fall steht es ganz am Ende und die ganze Liste muss durchlaufen werden, bevor es gefunden wird.
 - ❑ **Average Case.** Im Mittel müssen ungefähr die Hälfte der Elemente angesehen werden.

- Um eine Komplexitäts-klasse zu bezeichnen, gibt man immer die einfachste Funktion an, die geeignet ist, die jeweilige Komplexitäts-klasse zu repräsentieren. Tatsächlich handelt es sich bei $O(2n^2 + 7n - 10)$ und $O(n^2)$ um dieselben Mengen, man wählt aber $O(n^2)$ als Bezeichnung.
- Analyse von Funktionen wird unter der Annahme einer strikten Auswertung erfolgen (keine Lazy Evaluation)

Beispiel: Fakultätsfunktion

```
fakul 0 = 1
fakul n = n * fakul (n-1)
```

- Der Rekursionsanker (Terminierungsausdruck Zeile 1) benötigt nur konstante Zeit, also $\Theta(1)$.
- Annahme: Multiplikationen und Subtraktionen sind primitive Operationen und benötigen konstante Zeit, also $\Theta(1)$
- Die Laufzeit $T(n)$ der Fakultätsfunktion als Rekursionsgleichung aufschreiben:

$$T(0) = \Theta(1)$$

$$T(n) = \Theta(1) + T(n - 1)$$

- Der Aufwand ist dann $(n+1) \cdot \Theta(1) = \Theta(n+1) \in \Theta(n)$

Beispiel: Elemente in einer Liste finden

```
suche :: Eq a => a -> [a]-> Maybe a
suche _ [] = Nothing
suche a (x:xs)
  | a == x = Just a
  | otherwise = suche a xs
```

- Bei dieser Funktion hängt die Laufzeit stark von der Art der Eingabe ab. Wenn das gesuchte Element ganz am Anfang der Liste steht, sind wir schon mit nur einem Vergleich fertig. Die best-case Laufzeit ist also $\Theta(1)$.
- Ist es gar nicht vorhanden oder das letzte Element in der Liste, müssen wir uns alle anderen Elemente einmal ansehen. Im worst-case ist das recht offensichtlich eine lineare Laufzeit $\Theta(n)$.

- D.h. es werden $\Theta(k)$ Operationen benötigt wenn sich das Element an Position k befindet
- Es gilt dann:

$$T(n) = \frac{1}{n+1} \sum_{i=0}^n \Theta(1) = \frac{1}{n+1} \Theta\left(\frac{n(n+1)}{2}\right) = \Theta\left(\frac{n}{2}\right) = \Theta(n)$$

Beispiel: Listen umkehren

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

- Hier gilt dann:

$$T(0) = 1$$

$$T(n) = n + T(n-1)$$

- Man erhält dann folgende Ableitung der Laufzeitklasse aus vorheriger Rekursionsgleichung:

$$T(n) = n + (n-1) + (n-2) + \dots + 1 = \sum_{i=0}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

- Also quadratische Laufzeitklasse!
- Alternative durch Faltung benötigt nur $\Theta(n \cdot f) + \Theta(s) = \Theta(n)!!!$

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

- Das ergibt sich aus der Rekursionsgleichung der Laufzeitanalyse, der Rekursionsanker benötigt $\Theta(s)$ Schritte:

$$T(0) = s$$

$$T(n) = f + T(n-1)$$

8.5. Lazy Evaluation

- ▶ Lazy Evaluation (verzögerte Auswertung) ist die bedarfsabhängige Auswertung von Ausdrücken und Funktionsapplikationen
- ▶ Die Lazy Evaluation wertet nur Ausdrücke aus, die tatsächlich benötigt werden. So kann Rechenzeit eingespart werden, und sogar mit problematischen Werten gerechnet werden.
- ▶ Ein Nachteil der verzögerten Auswertung ist, dass man nicht immer wissen kann, wann ein Ausdruck ausgewertet ist. Falls die Auswertung Nebenwirkungen hat, ist nicht absehbar, wann diese eintreten.
- ▶ Nur wenige Laufzeitsysteme unterstützen eingebaute Bedarfsauswertung (OCaML z.B. nur explizit bei Mengen und Listen über Module) → Tiefe Codeanalyse u.U. erforderlich! → Lambda Calculus bietet Grundlage dafür
- ▶ Die Lazy Evaluation wird in JavaScript anders als in manchen funktionalen Sprachen nicht von der Sprache selbst unterstützt. Es gibt allerdings Bibliotheken, die die Lazy Evaluation in JavaScript ermöglichen (wu.js).

Auswertungsstrategien

Normal Order

Bei ihr wird von außen nach innen ausgewertet. Hier wird der Funktionsrumpf daher vor den Argumenten ausgewertet.

call-by-name

Bei ihr wird zuerst der Funktionsrumpf ausgewertet und dann die Argumente. Bei call-by-name werden dadurch bei einer Funktionsanwendung die noch nicht ausgewerteten Argumente im Funktionsrumpf ersetzt und müssen möglicherweise mehrfach ausgewertet werden. Im Gegensatz zur normal order wird bei call-by-name allerdings ein Funktionsrumpf, der nicht angewendet wird, nicht ausgewertet.

call-by-need

Im Gegensatz zu *call-by-name* wird ein Argument nur einmal ausgewertet und dieser ausgewertete Ausdruck wird für folgende Verwendungen zwischengespeichert.

- Die rein funktionale Programmiersprache Haskell benutzt die verzögerte Auswertungsstrategie *call-by-need*.

Haskell Beispiele

```
f n = [1..n]
g n = head (f n)

h order l1 l2 = if order == 'left' then reverse l1 else l2
let l3 = h "right" [1..100] [1..200]
```

JavaScript Beispiele

```
if (f(x) || g(y) || h(z)) I;
⇔
if (f(x)) I else if (g(y)) I else if (h(z)) I;
```

8.6. Memoization

- Memoization ist keine Programmiermethode sondern ebenfalls wie Lazy Evaluation eine Laufzeitfunktion um die Programmausführung zu beschleunigen.
- Annahme: Eine Funktion mit dem Parametertupel (p_1, p_2, \dots, p_m) liefert immer das gleiche Berechnungsergebnis für gleiche Eingabetupel $(v_1, v_2, \dots, v_m) \rightarrow (y_1, y_2, \dots, y_n)$.
 - D.h. Die Ausgabewerte hängen nur von den Eingabewerten und nicht von weiteren Variablewerten (freie Variablen mit Seiteneffekten) ab!
- Dann muss für jedes Eingabetupel (v_1, v_2, \dots, v_m) ein Ergebniswert (y_1, y_2, \dots, y_n) nur einmalig berechnet werden.
- Dieser Wert wird gespeichert (an die Funktion gebunden).
- Alle weiteren Funktionsapplikationen mit den gleichen Eingabewert benötigen nur noch das Lesen des Wertspeichers und keine Neuberechnung!!!!

```

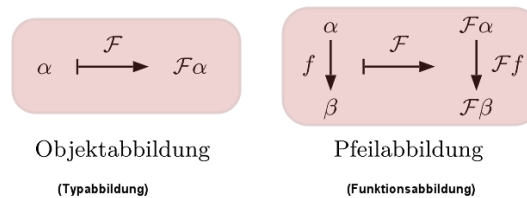
function alwaysCompute (x) {
  if (x<2) return 1;
  else return alwaysComoute(x-2)+alwaysCompute(x-1);
}

var memo=[];
function maybeCompute (x) {
  if (memo[x] != undefined) return memo[x];
  if (x<2) return 1;
  else return memo[x]=maybeComoute(x-2)+maybeCompute(x-1);
}

```

8.7. Kategorien und Funktoren

- Im Wesentlichen ist jede **Typklasse** eine **Kategorie**.
 - ❑ Die mathematische Definition verlangt dafür nur die Existenz der Identitätsfunktion und der Funktionskomposition.
- In der Mathematik sind **Funktoren** F generell Abbildungen von Kategorien in Kategorien.
 - ❑ Praktisch parametrisierte Typen und Module
 - ❑ In der programmiersprachlichen Verwendung sind es daher Abbildungen von Typen auf Typen, also das, was wir als generische oder polymorphe Typen bezeichnet haben (wie z.B. $Seq\ \alpha$)
 - ❑ Es gibt einen $*$ Operator der die Funktionen der einen Kategorie (d.h. z.B. Identität) auf Funktionen der anderen Kategorie abbildet \rightarrow map Operator!!!
- In der Mathematik wird ein Funktor üblicherweise in Form von zwei Abbildungen dargestellt, einer Objektabbildung und einer Pfeilabbildung.
- Der Zusammenhang wird im folgenden Diagramm dargestellt (wobei der Pfeiloperator nicht als $f *$ sondern als $F f$ geschrieben wird)



```
class Functor F where
  fmap :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  (F  $\alpha \rightarrow$  F  $\beta$ )
```

Die Applikation wird dann als entsprechende Instanz definiert, z. B.

```
instance Functor Seq where
  fmap = *
```

- Kategorien und Funktoren sind – im Kontext funktionaler Sprachen – eigentlich sehr einfache Konzepte (wenn man von den etwas merkwürdigen Namen absieht).
 - ❑ Das eine sind de facto Typklassen,
 - ❑ das andere generische Datentypen zusammen mit dem Map-Operator.
- Demgegenüber sind die Monaden sperriger im Formalismus und Notation.

8.8. Monaden

- In der Mathematik sind **Monaden** *spezielle Funktoren*, für die gewisse natürliche Transformationen existieren.
- Mathematisches Konzept aus der Kategorientheorie und werden in Haskell unter anderem auch für die Ein- und Ausgabe verwendet.
- Diese Transformationen sind im Wesentlichen die Funktionen
 - ❑ *lift* - ist einfach, sie stellt eine Einbettung des Basistyps in den generischen Typ dar (z.B. die Bildung der einelementigen Sequenz aus einem Element).
 - ❑ *flatten* für die Reduktion
 - ❑ und *ℰ* für die Kombination

Definition 35. (Monade)

Generell lassen sich Monaden einfach charakterisieren: Eine Monade ist ein polymorpher Typ $M \alpha$ zusammen mit vier speziellen Funktionen $*$, *lift*, *flatten* und *ℰ*

Beispiel: Debugger als Monade

- Bei der Lösung komplexer Probleme, kann schnell der Punkt erreicht werden, an dem sich ein geschriebenes Programm anders verhält als es erwartet wird. In solchen Situationen wäre es hilfreich, wenn neben den eigentlichen Berechnungen auch noch eine Mitschrift des Programmablaufs erstellt werden könnte.
- Angenommen es gibt zwei Funktionen von denen eine Mitschrift bei der Ausführung erforderlich ist:

```
f :: a -> b
g :: b -> c
```

- Die Funktionen könnten jetzt um einen weiteren Rückgabewert (Text) erweitert werden:

```
f :: a -> (b , string)
g :: b -> (c , string)
```

- Nun ist aber die Komposition $f(g(x))$ nicht mehr möglich da die Typsignaturen nicht mehr passen. Stattdessen wird es noch komplizierter:

```
h x = let
  (fErg , fMit) = f x
  (gErg , gMit) = g fErg in (gErg , gMit ++ fMit)
```

- Alternativ kann man eine Funktion *verbinde* definieren, die den Eingabetypen anpasst und sich um die nötige Verknüpfung der Mitschriften kümmert.
- Da die Erstellung der Mitschriften ein **sequentieller Ablauf** ist, wird das Argument vom Typ (a, string) an die erste Stelle gestellt
- Jetzt kann man eine Verkettung von Funktionen ermöglichen, wo die Argumente von links nach rechts gelesen werden (*Sequenz*)

```
verbinde :: (a,String) -> (a -> (b, String)) -> (b, String)
verbinde (x,s) f = let (fErg , fMit) = f x in (fErg ,s++fMit)

h x = f x 'verbinde' g
```

- Ein Beispiel mit der Identitätsfunktion:

```
f x = (x, "f aufgerufen. ")
g x = (x, "g aufgerufen. ")
h x = f x `verbinde` g `verbinde` (\x -> (x, "fertig. "))
```

Beispiel: Zufallszahlen

- Ebenfalls problematisch ist der Einsatz von Zufallszahlen. Ihre Verwendung scheint geradezu ein Widerspruch zum Konzept der mathematischen Funktionen zu sein, die nur von ihren Eingaben abhängig sind.
- Zufallszahlengeneratoren sind zustandsbehaftet! Daher in Haskell: Wenn eine Zufallszahl über einer Generator erzeugt wird, dann führt in der funktionalen Programmierung zu einem “neuen” Generator.
 - Zufallsgeneratoren sind tatsächlich Sequenzgeneratoren → Sequenz in FP → Monade!
- Haskell bietet zur Erzeugung eines Zufallswerts die in `Data.Random` definierte Funktion `random :: StdGen -> (a, StdGen)`
- D.h. eine Funktion die Zufallszahlen nutzen möchte muss ebenfalls den modifizierten Generator als Ergebnis “weiterreichen”:

```
f :: a -> StdGen -> (b, StdGen)
```

- Der veränderte Zufallszahlengenerator soll mit dem Ergebnis zurückgegeben werden, damit die nächste Funktion diesen wieder verwenden kann, um neue Zufallszahlen zu generieren.
- Jetzt wird es wieder mit der Komposition komplex ...
- Daher die Definition der “Monade” `random`:

```
verbinde :: (StdGen -> (a, StdGen)) ->
  (a -> StdGen -> (b, StdGen)) -> StdGen -> (b, StdGen)
verbinde g f gen = let (a, gen') = g gen in f a gen'
einheit x gen = (x, gen)
lift f = einheit.f
```

- Die Funktion *lift* ermöglicht es, deterministische Funktionen in eine Kette von nichtdeterministischen Funktionen einzureihen.

```

STRUCTURE Maybe: Monad RENAMING M AS Maybe = {
  TYPE Maybe α = α | {fail}           -- generischer Typ
  FUN _*: (α → β) → (Maybe α → Maybe β)  -- Map
  DEF f * a = f a                      -- Castings implizit!
  DEF f * fail = fail

  FUN lift: α → Maybe α                -- Up-Casting
  DEF lift a = a AS Maybe α

  FUN flatten: Maybe Maybe α → Maybe α    -- einmal Down-Casting
  DEF flatten (a: α) = (a: Maybe α)       -- Castings implizit!
  DEF flatten (fail: Maybe α) = (fail: Maybe α)
  DEF flatten (fail: Maybe Maybe α) = (fail: Maybe α)

  FUN _&: Maybe α → (α → Maybe β) → Maybe β
  DEF a & f = f a                        -- Castings implizit!
  DEF fail & f = fail
}

```

Abb. 21. Beispiel: Der generische Haskell Typ *Maybe* als Monade in formaler Pseudontation [B]

- Typklassen erklären die Klassendefinition und welchen Typ die geforderten Funktionen haben müssen
- Die Semantik bleibt aber verborgen und ist nur dem Programmierer überlassen.
- Damit ein Typ sich aber rechtmäßig Monade nennen darf, müssen die Funktionen die drei folgenden Gesetze erfüllen:

Rechtsidentität

```
m >>= return = m
```

Linksidentität

```
return x >>= f = f x
```

Assoziativität

```
(m >>= f) >>= g = m >>= (\x.f x >>= g)
```

- In Haskell gibt es die vordefinierte Typklasse *Monad*, mit der Monaden konstruiert werden können.

8.9. Automaten als Monaden

- Ein wichtiges Charakteristikum funktionaler Programmiersprachen ist, dass sie von Zeit und Zustand unabhängig sind.
- Aber Interaktion mit der Umwelt - insbesondere dem Benutzer - ist abhängig von der "Zeit" der realen Welt (und Reihenfolge).
- Interaktive Systeme arbeiten sequenziell und können durch einen endlichen Zustandsautomaten abgebildet werden (Finite State Machine, FSM).
- Automaten-Monaden oder Maschinen-Monaden können solche FSM umsetzen
 - Der Zustand (die Zustände) wird (werden) in einen (verborgenen) Typ *State* gekapselt.
 - Um diesen Typ *State* herum wird eine parametrisierte Struktur gebaut, die eine Instanz der Spezifikation *Monad* ist.

```

STRUCTURE Machine: Type → Monad RENAMING (M, lift) AS (Com, yield)
DEF Machine(State) = {
  TYPE Com α = (observe = State → α, evolve = State → State)
  FUN _*: (α → β) → (Com α → Com β)           -- Map
  DEF f * (obs, ev) = (f ∘ obs, id ∘ ev)
  FUN yield: α → Com α                             -- lift
  DEF yield a = (K a, id)
  FUN flatten: Com Com α → Com α                 -- iterierte Ausführung
  DEF flatten cc = cc & id
  FUN _&: Com α → (α → Com β) → Com β           -- sequ. Komposition
  DEF (obs, ev) & f = (f ∘ obs) S ev
  WHERE
    (f S g)x = (f x)(g x)                   -- S-Kombinator
  FUN _&: Com α → Com β → Com β                 -- Variante (Kurzform)
  DEF m1 & m2 = m1 & (K m2)                 -- konstante Funktion
  \

```

Abb. 22. Der generische Typ *Machine* als Monade in formaler Pseudonotation [B]

- Die Arbeit von Maschinen besteht in der Ausführung von Instruktionen.
- Diese Instruktionen haben üblicherweise zwei Effekte:
 1. Sie bewirken einen internen Zustandsübergang;
 2. Sie liefern nach außen sichtbare Resultate.
- Diese Situation wird in dem Typ *Com* (für Command) repräsentiert.

Definition 36. (*Command*)

Ein (monadisches)Kommando ist ein Paar von Funktionen. Die erste ist eine externe Beobachtungsfunktion *observe*, die aus dem inneren Zustand einen extern sichtbaren Wert extrahiert. Die zweite ist eine interne Transformation *evolve**, die den internen Zustand in einen neuen Zustand überführt.

9. Referenzen

9.1. Bücher

- A. P. Pepper, Funktionale Programmierung: in OPAL, ML, HASKELL und GOFER, Springer Berlin, 2003
- B. P. Pepper and P. Hofstedt, Funktionale Programmierung. Springer Berlin, 2006.
- C. M. Block and A. Neumann, Haskell Intensivkurs. Springer Berlin, 2011.
- D. H. Mehnert, J. Ohlig, and S. Schirmer, Funktional programmieren lernen mit JavaScript. O'REILLY, 2013.

9.2. Literatur

9.3. Videos und WEB