Einführung in die Informatik I

Wintersemester 2002|2003

Alois Knoll
Technische Universität München
LS Informatik VI – Robotics and Embedded Systems

Informatik I: Organisation

Bestandteile der Veranstaltung

- **Vorlesung**
- **Zentralübung**
- **Tutorübungen**
- **Zwischenklausur**
- **Semestralklausur**

Vorlesung

- **Dienstag, 12 … 14 Uhr**
- **Donnerstag, 8 … 10 Uhr**
- **jeweils im Hörsaal MW 2001**

- **Keine Anwesenheitspflicht!**

Zentralübung

- **Zeit: Mittwoch, 12 … 13 Uhr**
- **Raum: MW 2001**
- **Beginn: 23.10.2002**

Tutorübungen

- **Einschreibung:**
  - **Do., 17.10. bis Fr., 18.10. 17.00 Uhr**
  - **Informatikhalle: Raum 00.05.11**
- **Einteilung: Di., 22.10**
- **Beginn: 22.10.**
- **Wöchentlich 3 Stunden, Di. oder Mi.**

Tutorübungen

- **Haus- und Programmieraufgaben**
  - **Wöchentlich**
  - **Verschlüsselt per E-mail an den Tutor**
  - **Format: PDF**
  - **Star-Office: eine Möglichkeit, PDF-Dateien zu erzeugen**

- **Aufgabenblätter erhältlich unter:**
  - `http://wwwknoll.in.tum.de/vorlesungen/info1/`

Schriftliche Zwischen- und Semestralklausur

- **Zwischenklausur**
  - **14.12.2002, 13.00 Uhr**
  - **1/3 der Gesamtnote**
- **Semestralklausur**
  - **08.02.2003, 9.00 Uhr**
  - **2/3 der Gesamtnote**

Rechnerarbeitsplätze

- In der Rechnerhalle befinden sich ~ 200 Workstations vom Typ Sun, Betriebssystem Solaris

Kennungen für die Informatikhalle

- **Studenten im Diplom- und Bachelorstudiengang Informatik**
  - **Bereits schriftlich erhalten!**
- **Studenten mit Nebenfach Informatik:**
  - **17. & 18.10. in der Informatikhalle (00.05.11) beantragen.**
  - **Di., 22.10. Erhalt der Kennungen im *Servicebüro* der Rechnerbetriebsgruppe (00.05.041).**

Weitere Informationen

- **WWW-Startseite:**
  - `http://wwwknoll.in.tum.de/vorlesungen/info1/`
- **Beim jeweiligen Tutor**
- **Übungsleitung:**
  - **email: uebungsleitung-info1@in.tum.de**
- **newsgroup: tum.info.info12**

Informatik I: Vorlesungsstruktur und -inhalt

Vorlesungsstruktur und -inhalt I

Allgemeine Vorlesungsmodalitäten (1 VL)

Ablauf der Vorlesung, Übungen, Anmeldung zu den Übungen
Wesentliche Werkzeuge: OCAML, Acrobat, StarOffice, Emacs

„Info I in a nutshell“: Illustration/Kurzüberblick funktionales Programmieren (4 VL)

OCaml auf den verschiedenen Plattformen
Funktionen, einfache Rekursion über natürlichen Zahlen
Tyddeklarationen; Rekursion auf selbstdeklarierten Typen

Zum Begriff der Information (5 VL)

Unterschiedliche Annäherungen an den Begriff (1 VL)

Repräsentationen von Information
Interpretation von Repräsentation
Beispiele: Sprache, Bild, Mobiltelefon als informationsverarbeitendes System

Shannonsche Informationstheorie (2 VL)

Grundlagen: Codierung, Entropie, ...
Codierungsverfahren: Huffman, LZW, ...

Kryptologie (2 VL)

Historie, Grundkonzepte
Zahlentheoretische Techniken der Kryptografie zur Erzeugung
geeigneter Schlüssel: RSA-Algorithmus
Anwendungen: ssh, pgp mit Vorlesungbetrieb als Beispiel

Vorlesungsstruktur und -inhalt II

Algebren, Rechen- und Datenstrukturen (5 VL)

Algorithmen und Textersetzungssysteme (1 VL):

- Was ist ein Algorithmus ?
- Beispiel: Codierungsalgorithmen
- Beispiel: Textersetzungssysteme, insbes. Markovsysteme (leichte Programmieraufgaben)

Algebren/Rechenstrukturen und Signaturen (4 VL)

- Definition: Algebra/Rechenstruktur und Signaturen
- Zentrale Rechenstruktren:Natürliche Zahlen, Sequenzen, Bäume, ..
- Terme und Termersetzungssysteme (-> Programmieraufgaben)
- Korrektheit von Termersetzungssystemen (-> Induktionsprinzip als Werkzeug)
- Aussagenlogik -> Prädikatenlogik als Rechenstruktur

Modelle der Informatik (7 VL)

- Formale Sprachen und BNF: Definitionen
- Beschreibung formaler Sprachen (Chomsky 2) durch BNF-Ausdrücke
- Fixpunktdeutung von BNF-Ausdrücken
- Chomsky-Hierarchie
- Automaten und reguläre Ausdrücke
- Graphen
- Turingmaschinen, etc

Vorlesungsstruktur und -inhalt III

Applikative / Funktionale Programmierung (5 VL)

Definition, Eigenschaften (2VL)

Rekursive Sorten
Syntaktische Beschreibung
Sequenzen, Stapel, Keller, Warteschlangen
Bäume
Fixpunktdeutung

Rekursive Funktionen (2 VL)

Syntaktische Beschreibung Kategorisierung; Linear, Nichtlinear, ...
Semantik mit Fixpunktdeutung
Terminierung, Korrektheit, Spezifikation

Lambda-Kalkül zur einheitlichen Beschreibung von Daten und Funktionen (-> Lisp) (1 VL)

Grundlegende Algorithmen (3 VL)

Sortierverfahren, Algorithmen auf rekursiven Strukturen
Effizienz

Entwurf komplexer Systeme (1 VL)

Ein einfacher Editor (~ Wordpad) oder ein Spiel

Info I in a nutshell

- Klärung von Grundbegriffen

- Informelle Illustration einiger Grundkonzepte

Vom Problem zu Algorithmus & Programm

- Das Ziel jeder Programmentwicklung besteht darin, eine **gegebene Problemstellung** unter zur Hilfenahme eines Rechnersystems **effizient** zu **lösen**.
- Der *goldene Weg* erfordert neben einer möglichst exakten Analyse der Problemstellung eine **effiziente** Umsetzung auf das Rechnersystem; ein *„eben mal Hacken“* führt kaum je zu einer effizienten Lösung.
- „Gute“ Programme können Berechnungszeiten u.U. extrem verkürzen. Aktuelles Beispiel: Schach – Eine vollständige Berechnung aller Schach-Züge ist nicht empfehlenswert, weil es etwa 10^{155} Stellungen gibt. Frage: Wie viele Stellungen pro Sekunde müßten berechnet werden, um das Ergebnis erleben zu können?

Einzelschritte:

1. Problembeschreibung → ***Spezifikation*** der Aufgabenstellung (hierfür bis zu 90% des gesamten Zeitaufwands der Erstellung von SW-Systemen)
2. Lösungsbeschreibung → Mehr oder weniger informelle Programm-Ablaufbeschreibung in Textform in einem ***Algorithmus***
3. Programmierung → Erstellung eines ***Programms*** in einer ***Programmier-sprache***
4. Ausführung → ***Programmablauf*** auf einem Rechnersystem

Problembeschreibung

Eine **Spezifikation** ist eine **vollständige**, **detaillierte** und **eindeutige Problembeschreibung**.

- **Vollständig:** Alle relevanten Rahmenbedingungen werden angegeben.
- **Detailliert:** Genaue Beschreibung aller zugelassenen Hilfsmittel sowie der zur Verfügung stehenden Grundaktionen.
- **Eindeutig:** Klare Kriterien, wann eine vorgeschlagene Lösung akzeptabel ist.

Wesentliche Techniken zur Behandlung von Komplexität bei der Probelmstellung: Modellbildung/Abstraktion (was / wie), Hierarchisierung, Trennung von Struktur und Verhalten, ... (siehe später)

Lösungbeschreibung

- Ein **Algorithmus** ist ein *systematisches*, *reproduzierbares* Problem-lösungsverfahren, d.h. eine präzise, endliche Verarbeitungsvorschrift für eine Maschine oder einen Menschen, die oder der in der Lage ist, die im Algorithmus angegebenen Elementaroperationen schrittweise durchzuführen.

- Formaler: Verfahren zur Berechnung von Ausgabedaten aus Eingabedaten aufgrund einer gegebenen Problemfunktion (= exakt definierte Aufgabe).
- Algorithmen finden sich z.B. in Form von:
  - Kochrezepten
  - Bastelanleitungen
  - Math. Berechnungsvorschriften

Eine **universelle Programmiersprache** ist formalisierte Sprache, in der man jeden Algorithmus formulieren kann.

Zwei wesentliche Eigenschaften von Algorithmen

- **Korrektheit**: Algorithmus soll das Problem so lösen, wie in Spezifikation vorgesehen. Bei **formaler Spezifikation** kann Korrektheit (im Prinzip) formal **bewiesen** werden.

  Definition: Algorithmus *A* ist *korrekt* bezüglich der zugehörigen Spezifikation, falls gilt: Für alle Werte aus dem Definitionsbereich von *A* terminiert *A* und liefert bei Anwendung auf *x* ein Resultat *a*, welches das zu *x* gehörige Ergebnis ist (totale Korrektheit, im Gegensatz zu partieller Korrektheit, die nur Terminierung fordert).

- **Effizienz**: Laufzeit und Speicherbedarf sollen minimal werden. Grundlage für die Analyse: Aufwand, entspricht der Anzahl der problemrelevanten Elementaroperationen.

Programmerstellung

- **Programmierung** ist die Umsetzung der (z.B. umgangssprachlichen) Beschreibung eines Algorithmus in eine maschineninterpetierbare Form.
- **Programm** ist die Formulierung eines Algorithmus' in einer bestimmten Programmiersprache.

Grundtypen von Programmier*sprachen*, u.a.:

1. Funktionale Programmiersprachen (Lisp, Haskell, Gofer, ML, OCaml, ...)
2. Logik-basierte Programmiersprachen (Prolog, ...)
3. Imperative Programmiersprachen (Fortran, Pascal, C, C++, Java, ...)

Grundtypen von Programmier*stilen*

1. Funktional
2. Komponentenbasiert
3. Ereignisbasiert
4. Objektorientiert (& Aspektorientiert)

Je näher die Formulierung des Algorithmus am später verwendeten Programmierstil ist, desto leichter fällt die Umsetzung.

Programmausführung

Unterschiedliche Möglichkeiten des Programmablaufs auf einem Rechensystem

1. Ausführung von „nativem Maschinencode“ (Sprache des Prozessors)
   Notwendigkeit eines speziellen Übersetzungsvorgangs durch ***Compiler***, der gegebenes Programm *P* in Programm *P*’ in Maschinensprache transformiert; *P*’ ist direkt vom Prozessor ausführbar.
   Vorteil: In der Regel hohe Ausführungsgeschwindigkeit
   Nachteil: Geringe Portabilität des Kompilats

2. Direkte Auswertung des Programmiersprachen „**Quell**-*Codes*“, ohne sichtbare Erzeugung eines weiteren Programms *P*’
   Erforderlich: ***Interpreter***, der den Quell-Code Zeile für Zeile liest, auswertet und ausführt
   Vorteil: Hohe Portabilität
   Nachteil: Langsamere Ausführung
   Veröffentlichung des Source-Codes

Einführende Bemerkungen zur funktionalen Programmierung

Illustration funktionaler Programmierung (FP)

- **Definition:** Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style. (www.cs.nott.ac.uk/~gmh/faq.html)

- **Damit gilt:**

  Funktionale Programmierung = Definition von Funktionen (Funktionsdefinition)

  Funktionale Programmausführung = Auswerten von Ausdrücken (Funktionsapplikation)

FP: Wesentliche Merkmale

1. Funktionsdefinitionen können ihrerseits auf weitere Funktionen zurückgreifen:
   ```
   result1 = function1(function2(function3, function4));
   ```

2. Keine Funktion darf sich während der Laufzeit in ihrem Eingabe-Ausgabe-Verhalten ändern. Vielmehr muss bei mehrmaliger Anwendung derselben Funktion mit identischen Argumenten stets *dasselbe Resultat* errechnet werden.

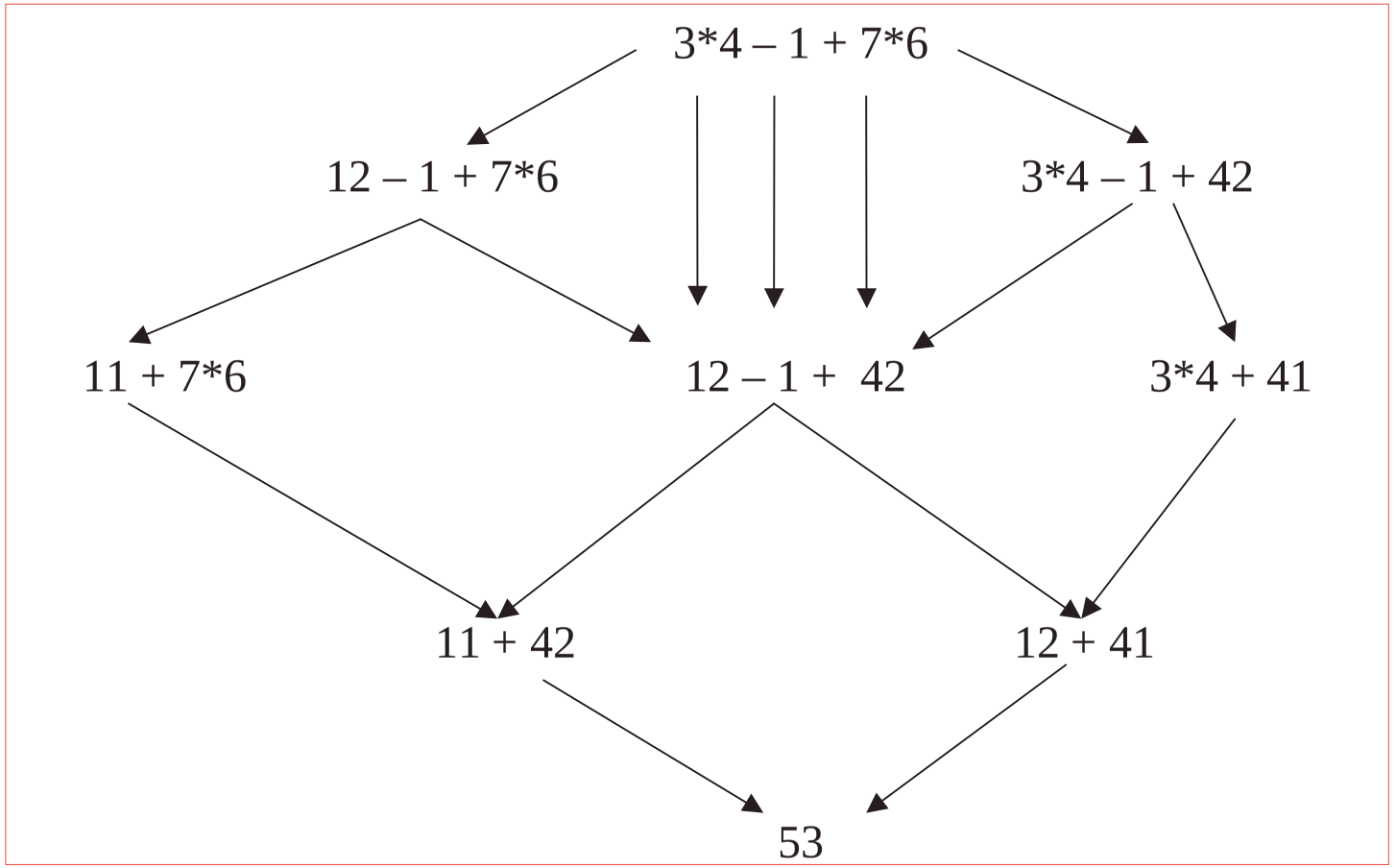
3. Konstanten sind Funktionen, welche stets denselben (konstanten) Wert zurückgeben, Variablen bezeichnen in einem gegebenen Kontext immer den gleichen Wert (**Werttreue**, referential transparency)

4. Einziger Anwendungszweck einer Funktion ist das Errechnen eines Resultatwertes. Aktionsausführungen die nicht in direktem Zusammenhang mit dem Errechnen des Funktionsresultats stehen, *sind nicht gestattet* (**keine *Seiteneffekte***)

Konsequenzen aus 1..4:

- Ergebnis einer Funktion wird **ausschließlich** durch die Parameter bestimmt, die an die Funktion bei ihrem Aufruf übergeben werden
- Ein Ausdruck wird nur verwendet, um einen Wert zu benennen. In einem gegebenen Ausdruck bezeichnet ein Ausdruck immer denselben Wert → Teilausdrücke können durch andere mit demselben Wert ersetzt werden (**Substitutionsprinzip**)
- Der Wert eines Ausdrucks ist **unabhängig von der Reihenfolge**, in der der Ausdruck ausgewertet wird → einfache parallele Auswertung

Reduktion eines Ausdrucks auf unterschiedlichen Wegen



Auswertung der Ausdrücke beeinflußt sich nicht gegenseitig: „Konfluenz“, Church-Rosser-Eigenschaft.

Nicht parallel ausführbar wäre z.B.:

$h \leftarrow 3*4$
$h \leftarrow h - 1$
$h \leftarrow h + 7*6$

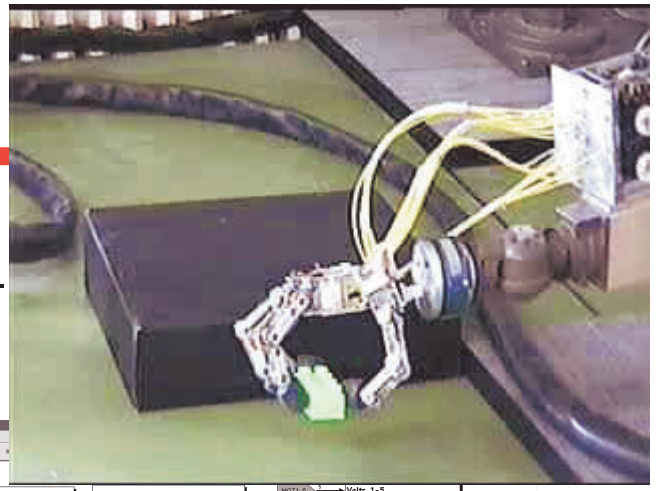
FP: Vorteile

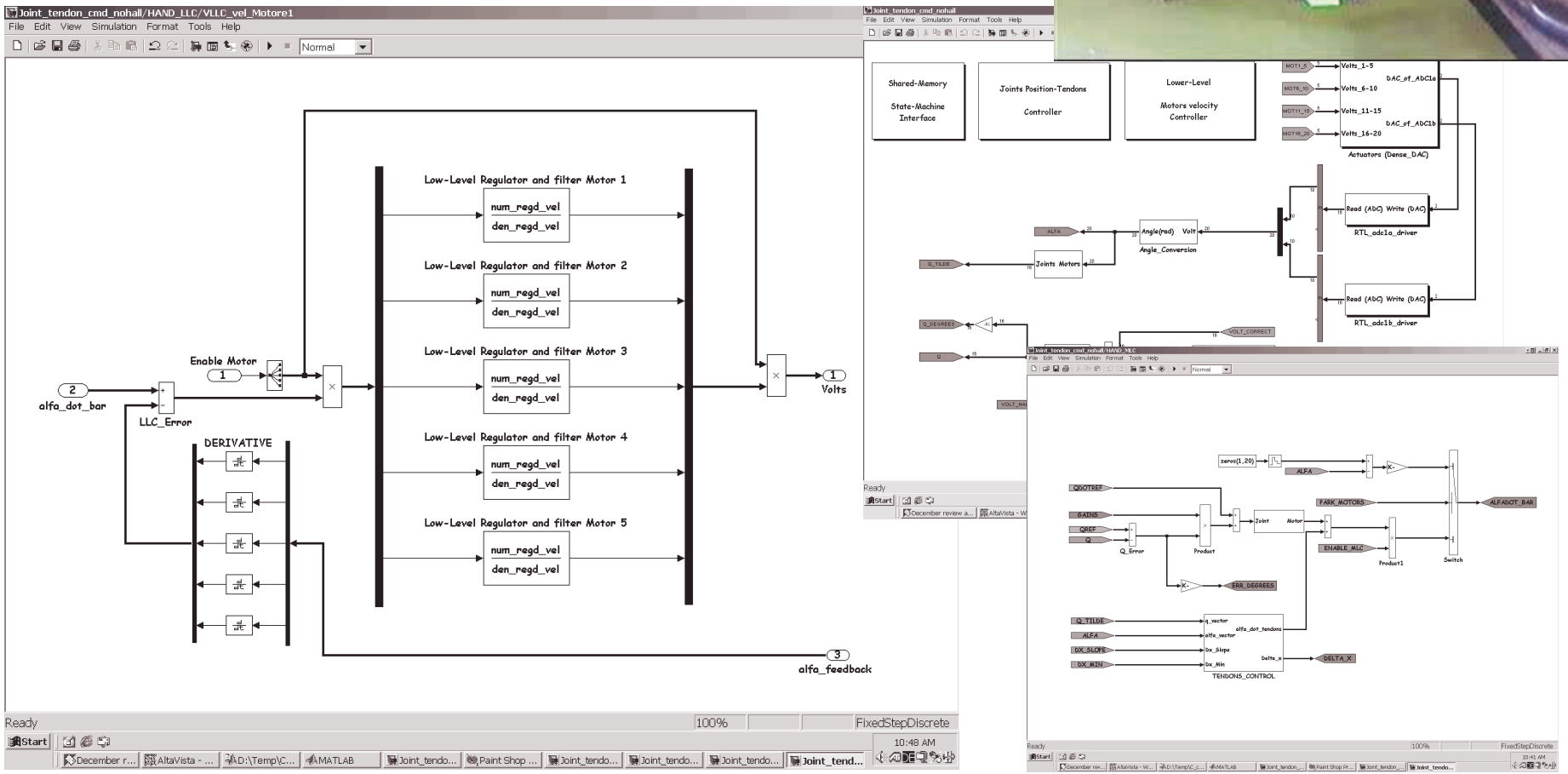
- Was macht die funktionale Programmierung interessant ?
  - Die Grundprinzipien sind sehr einfach
  - Funktionale Programme sind klar strukturiert (keine Seiteneffekte)
  - Bei den geeigneten Problemklassen sehr kurzer Weg vom Problem zum Programm

- Welche Vorteile ergeben sich ?
  - Förderung eines klaren Programmierstiles
  - Förderung einer Denkweise, die die Abstraktion in den Mittelpunkt stellt
  - (Relativ) leichte Überprüfung der Korrektheit von Programmen
  - etc. (siehe später)

FP: Vorteile

– Direkte Umsetzbarkeit des black-box- bzw. Datenflußdenkens z.B. in den Ingenieurwissen-schaften:



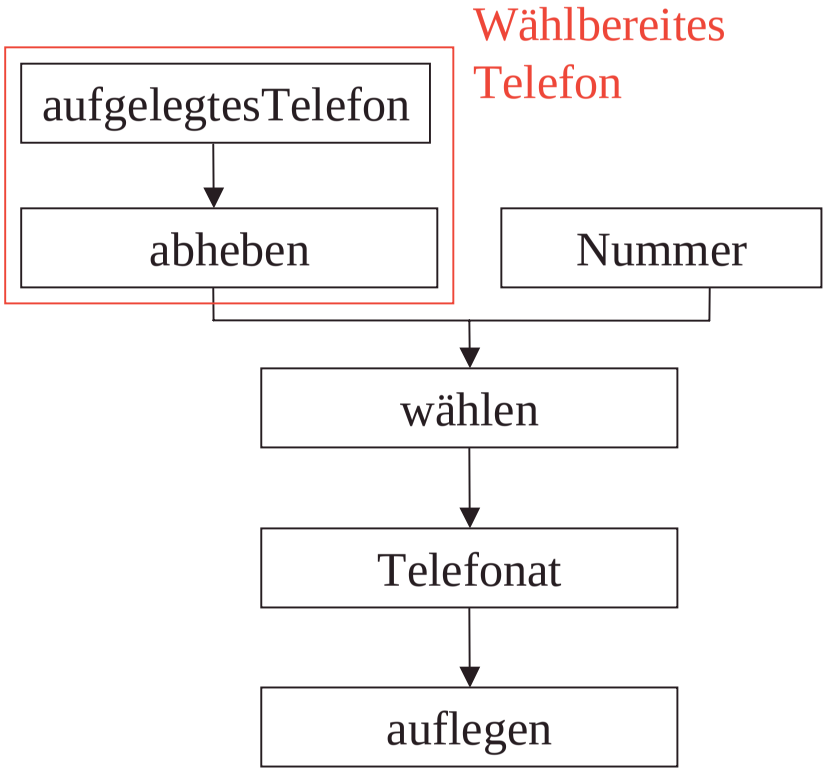


FP: Funktionale Beschreibung eines alltäglichen Vorgangs

- Gesucht: Funktionale Beschreibung (Algorithmus) eines ***Telefonats.***
- Gegeben: Einfaches Wähltelefon (aufgelegt), zu wählende Nummer
- Im Rahmen der Problemanalyse und Lösungsentwicklung kann eine graphische Darstellung hilfreich sein:

Modell 611
Das A-Modell. Dieses seit vielen Jahren bewährte Tischtelefon gibt es in vier attraktiven Farben
Farben: farngrün, kieselgrau, hellrotorange, ockergelb

Modell 611
Das A-Modell mit Schauzeichen
Zusätzliche monatliche Gebühren 1,90 DM. Stand 1.4.1980
Farben: farngrün, kieselgrau, hellrotorange Quelle: Prospekt der Deutschen Bundespost. Machen Sie sich's telefonbequem!(1981)



FP: Funktionale Beschreibung eines alltäglichen Vorgangs

- Zur Vorbereitung der Umsetzung in ein funktionales Programm wird der Algorithmus informell (Text) funktional aufgeschrieben:

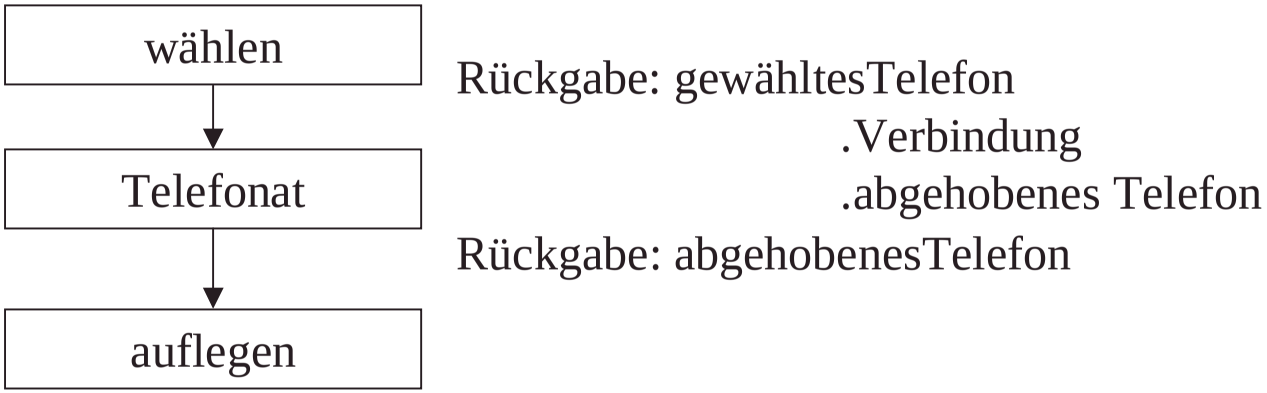
```
auflegen(Telefonat(wählen(abheben(aufgelegtesTelefon),Nummer)))
```

- Unberücksichtigt dabei: Überlegungen, was passiert, wenn die Funktionen die Ihnen gestellte Aufgabe nicht erfüllen können.
  - Es kann z.B. nicht davon ausgegangen werden, daß mit dem ***Wählen*** auch eine ***Verbindung*** zustande kommt und ein ***Telefonat*** mit ***Unterhaltung*** stattfindet.
  - Insofern ist der bisherige Algorithmus nicht detailliert genug, speziell fehlen alle Überlegungen zu Rückgabewerten der Funktionen.

FP: Funktionale Beschreibung eines alltäglichen Vorgangs

Detailliertere Betrachtung zur *schrittweisen Verfeinerung* der Funktion ***Telefonat***:

- Ausgabewert der Funktion ***wählen*** könnte eine Beschreibung (Datentyp) für **gewähltes Telefon** sein, welches neben dem Zustand der ***Verbindung*** auch das **abgehobeneTelefon** zur weiteren Auswertung bereitstellt. Letzeres kann schließlich ***aufgelegt*** werden.
- Für die **Verbindung** könnte gelten:
  - *gültig*: Der Partner hat abgehoben
  - *ungültig*: Die Leitung ist besetzt,
    oder wurde wegen max. Klingelzahl abgebrochen



FP: Funktionale Beschreibung eines alltäglichen Vorgangs

Textuell könnte die Funktion ***Telefonat*** damit folgende Gestalt haben:

Funktion ***Telefonat***(**gewähltesTelefon**)
liefert **gewähltesTelefon.abgehobenesTelefon** zurück,
falls **gewähltesTelefon.Verbindung** *ungültig* ist
oder
liefert das Ergebnis einer Funktion
Unterhaltung(**gewähltesTelefon**) falls
gewähltesTelefon.Verbindung *gültig* ist.

Dazu ist eine weitere Funktion ***Unterhaltung*** zu definieren, für welche gilt:

Funktion ***Unterhaltung***(**gewähltesTelefon**)
liefert als Ergebnis **gewähltesTelefon.abgehobenesTelefon** zurück
wenn das Gespräch beendet ist.

FP: Von einer Berechnungsfolge zum funktionalen Programm

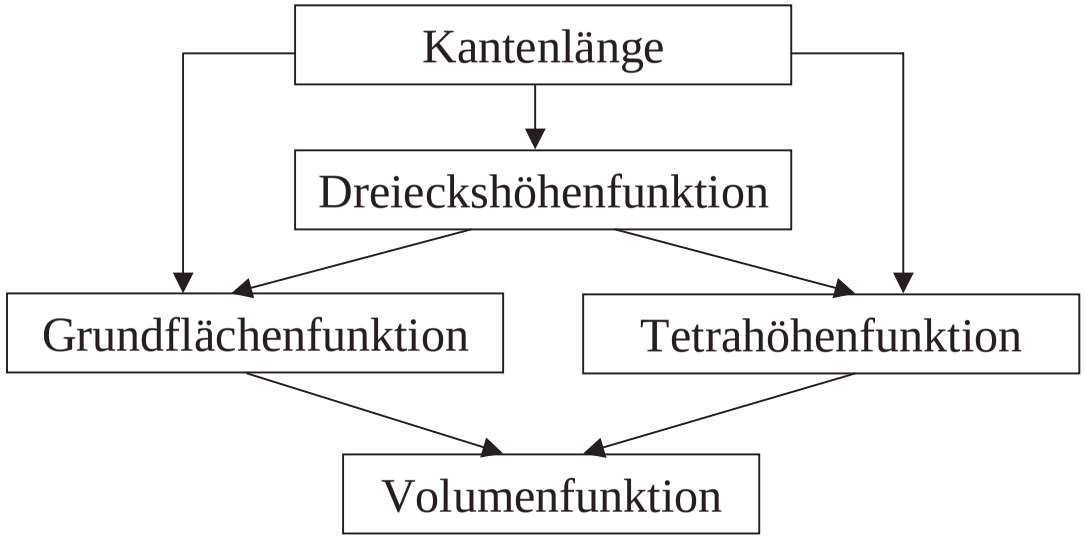
- **Gesucht**: Funktionales Programm zur Berechnung des Volumeninhaltes *V* eines geraden, gleichseitigen Tetraeders mit bekannter Kantenlänge. Dabei soll von geometrischen Überlegungen ausgegangen und auf algebraische Vereinfachungen verzichtet werden.

- Gegeben: Schulgeometrie, Kantenlänge,
  URL: http://www.mathematische-basteleien.de/tetraeder.htm

- Allgemeine Lösungsschritte:
  1. Aufstellung der Berechnungsvorschriften
  2. Funktionale Deutung dieser Vorschriften
  3. Umsetzung in ein funktionales Programm
  4. Ausführung auf einem Rechnersystems

FP: Von einer Berechnungsfolge zum funktionalen Programm

- Zur ***Problemspezifikation*** ist zunächst zu klären, was ein gerades, gleichseitiges Tetraeder ist: *Ein Körper aus vier gleichseitigen Dreiecken.*
- Die Kantenlänge ist gemäß Aufgabenstellung gegeben. Zur Lösung sind die entsprechenden geometrischen Beziehungen:
  - Volumen V = (1/3) * Grundfläche * Tetraederhöhe
  - Grundfläche A = (1/2) * Kantenlänge * Dreieckshöhe
  - Tetraederhöhe H = sqrt(Kantenlänge^2 – ((2/3)*Dreieckshöhe)^2)
  - Dreieckshöhe h = (Kantenlänge/2)*sqrt(3)



FP: Von einer Berechnungsfolge zum funktionalen Programm

Als funktionale Beschreibung ergibt sich folglich:

```
Volumen V =
        Volumenfkt(
        Grundflächenfkt(Kantenlänge,Dreieckshöhenfkt(Kantenlänge)),
        Tetrahöhenfkt(Kantenlänge, Dreieckshöhenfkt(Kantenlänge)))
```

- Nun Umsetzung in eine funktionale Programmiersprache, sowie die Ausführung des Programms auf einem Rechnersystem für unterschiedliche Kantenlängen (Funktionsapplikation).

Funktionale
Programmierung
mit OCaml

OCaml – Historie

- 1978 Entwurf der Metasprache ML (MetaLanguage) durch R. Milner
- 1981 – 1986 ML Compilerentwicklung u.a. bei INRIA (Frankreich)
- 1984 Erweiterung und Standardisierung der Sprache (Standard ML)
- 1984 Entwicklung der Sprache Caml bei INRIA
- 1990 Vorstellung von Caml-Light
- Heute: Objective Caml

FP: Eigenschaften von Ocaml I

Objective Caml is a functional language: it manipulates functions as values in the language. These can in turn be passed as arguments to other functions or returned as the result of a function call.

Objective Caml is statically typed: verification of compatibility between the types of formal and actual parameters is carried out at program compilation time. From then on it is not necessary to perform such verification during the execution of the program, which increases its efficiency. Moreover, verification of typing permits the elimination of most errors introduced by typos or thoughtlessness and contributes to execution safety.

Objective Caml has parametric polymorphism: a function which does not traverse the totality of the structure of one of its arguments accepts that the type of this argument is not fully determined. In this case this parameter is said to be *polymorphic*. This feature permits development of generic code usable for different data structures, such that the exact representation of this structure need not be known by the code in question. The typing algorithm is in a position to make this distinction.

FP: Eigenschaften von Ocaml II

Objective Caml has type inference: the programmer need not give any type information within the program. The language alone is in charge of deducing from the code the most general type of the expressions and declarations therein. This *inference* is carried out jointly with verification, during program compilation.

Objective Caml is equipped with an exception mechanism: it is possible to interrupt the normal execution of a program in one place and resume at another place thanks to this facility. This mechanism allows control of exceptional situations, but it can also be adopted as a programming style.

Objective Caml has imperative features: I/O, physical modification of values and iterative control structures are possible without having recourse to functional programming features. Mixture of the two styles is acceptable, and offers great development flexibility as well as the possibility of defining new data structures.

Objective Caml executes (threads): the principal tools for creation, synchronization, management of shared memory, and interthread communication are predefined.

Objective Caml communicates on the Internet: the support functions needed to open communication channels between different machines are predefined and permit the development of client-server applications.

FP: Eigenschaften von Ocaml III

Objective Caml communicates on the Internet: the support functions needed to open communication channels between different machines are predefined and permit the development of client-server applications.

Numerous libraries are available for Objective Caml: classic data structures, I/O, interfacing with system resources, lexical and syntactic analysis, computation with large numbers, persistent values, etc.

A programming environment is available for Objective Caml: including interactive toplevel, execution trace, dependency calculation and profiling.

Objective Caml interfaces with the C language: by calling C functions from an Objective Caml program and vice versa, thus permitting access to numerous C libraries.

Three execution modes are available for Objective Caml: interactive by means of an interactive toplevel, compilation to bytecodes interpreted by a virtual machine, compilation to native machine code.

Einige Technische Hinweise zum Gebrauch von OCaml

Editor emacs („Editor MACroS“):

http://www.gnu.org/software/emacs/windows/

http://ftp.gnu.org/gnu/windows/emacs/latest: emacs-21.2-fullbin-i386.tar.gz

Installation durch Entpacken (z.B. mit WinZip, erhältlich über http://www.shareware.com)

Emacs-Modus für OCaml:

http://www-rocq.inria.fr/~acohen/tuareg/mode/

Installation durch Einrichten einer Datei „`.emacs`“ oder durch Kopieren der files *.el in das Verzeichnis ...\emacs-21.2\bin\

Nach Aufruf von emacs: Alt-x -> „load-file“<cr> -> „tuareg“<cr> und dann Alt-x tuareg-run-ocaml <cr>

Mit <ctrl>c <ctrl>r kann dann eine markierte Region direkt evaluiert werden.

OCaml: Programmstruktur

- Im Mittelpunkt steht die Definition und die Anwendung von Funktionen, gleichzeitig gibt es Sprachkonstrukte für die Programmierung großer Systeme, wie etwa ein Modulkonzept oder flexible Ausnahmebehandlung.

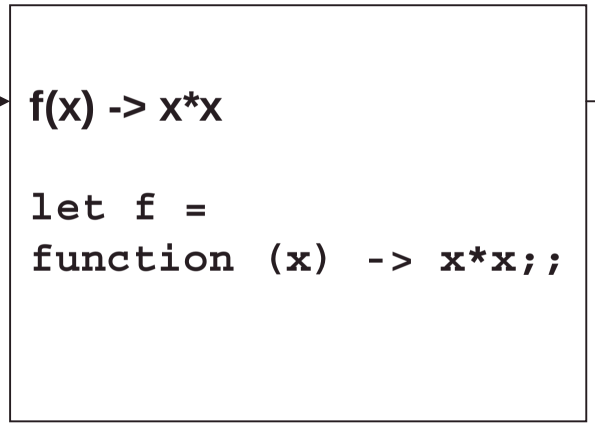
ocaml-2.04/stdlib/stack.ml

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with hd::tl → s.c <- tl; hd | [] → raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

Konkretisierung

Funktionsparameter

- sind typisiert und strukturiert.
- können selbst Funktionen sein.



Funktionsresultate

- sind typisiert und strukturiert.
- können selbst Funktionen sein.

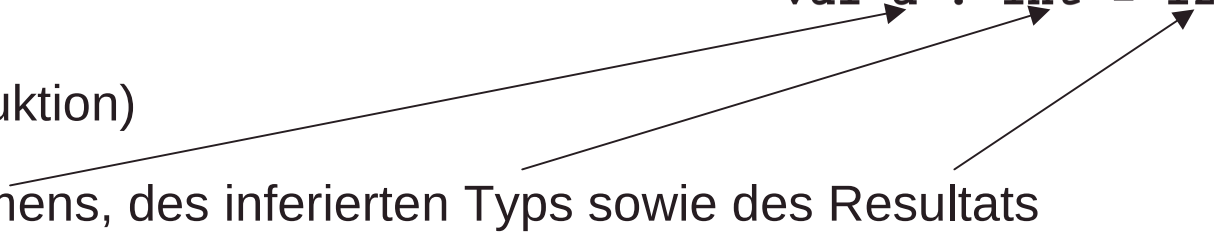
Interpretation von OCaml Ausdrücken

```
> ocaml

            Objective Caml version 3.06

# 1 - 2 + 3 * 4 / 5;;

- : int = 1
```

Schritte der Interpretation von Ausdrücken:

1. Syntaxcheck
2. Typinferenz
3. Ausführung (Reduktion)
4. Ausgabe des Namens, des inferierten Typs sowie des Resultats

```
# let a = 3*4;;

val a : int = 12
```



Basisdatentypen: Zahlen

In OCaml sind zwei Arten von Zahldarstellungen vordefiniert:

1. **int** für ganze Zahlen (*integer*)
2. **float** für reelle Zahlen (*Floating-Point-Numbers*, Gleitkommazahlen)

```
# 1.2 -. 3.4 +. 5.6 *. 7.9 /. 10.0;;
- : float 2.224
```

- Hierbei ist die „*gepunktete*“ Version der **Operatoren** zu beachten.
- Zudem ist zu unterscheiden: `10.0` ist float, `10` jedoch integer !

Basisdatentypen: Operatoren auf int und float

Operation	int	float
Addition	+	+.
Subtraktion	-	-.
Multiplikation	*	*.
Division	/	/.
Modulo	mod	
Potenzierung		**

Zu beachten: OCaml ist streng typisiert. Kein Typ-/Operator-"*Misch-Masch*" !

```
# 1 + 2.3;;
This expression has type float but is here used with type
int
```

Explizite Typumwandlung ist möglich:

```
# int_of_float 2.0;;      - : int = 2
# float_of_int 2;;        - : float = 2.
```

Basisdatentypen: Gleitkommafunktionen

Quadratwurzel `# sqrt(2.0);  - : float = 1.41421356237`

e-funktion `# exp(1.0);;  - : float = 2.71828182846`

Nat. Logarithmus `# log(exp(1.0));;  - : float = 1.`

Zehnerlogarithmus `# log10(100.0);;  - : float = 2.`

Aufrundung `# ceil(7.2);;  - : float = 8.`

Abrundung `# floor(7.2);;  - : float = 7.`

Kosinus `# cos(3.1415);; - : float = -0.999999995708`

Sinus `# sin(3.1415);; - : float = 9.26535896605e-005`

Sonst. Trigonometrische Funktionen: tan, acos, asin, atan

Basisdatentypen: Zeichen und Zeichenketten

In OCaml sind zwei Arten von Zeichen-Datentypen vordefiniert:

1. **char** für Einzelzeichen
2. **string** für Zeichenketten

```
# 'z';;
- : char = 'z'
# let s = "Dies ist ein Ocaml-String";;
val s : string = "Dies ist ein Ocaml-String"
```

Wichtigste Operat.:	`^`	Zusammenfügen (Konkatenation)
	`string_of_int`	Umwandlung `int` → `string`
	`int_of_string`	Umwandlung `string` → `int`

```
# "Die Vorlesung dauert" ^ string_of_int(90) ^ "Minuten.";;
- : string = "Die Vorlesung dauert 90 Minuten.„
```

```
# string_of_int (1234);;
- : string = "1234"
```

Basisdatentypen: Boolsche Werte

Mit **bool** steht in Ocaml ein Datentyp zur Repräsentation von Wahrheitswerten zur Verfügung:

Mögliche Werte sind :

true	für Wahr
false	für Falsch

Native Operatoren:

&&	**true**	**false**	
true	*true*	*false*	***„Und“***
false	*false*	*false*	

||	**true**	**false**	
true	*true*	*true*	***„oder“***
false	*true*	*false*	

not	**true**	**false**	
	false	*true*	***„nicht“***

Basisdatentypen: Vergleichsoperationen und boolsche Werte

Strukturelle Gleichheit	=	`# (sqrt (4.0)) = 2.0;; - : bool = true`
Physikalische Gleichheit	==	`# (sqrt (4.0)) == 2.0;; - : bool = false`
Negation von =	<>	
Negation von ==	!=	
Kleiner	<	
Größer	>	
Kleiner oder gleich	<=	
Größer oder gleich	>=	

Rückgabewerte sind Wahrheitswerte vom Typ **Bool** (true oder false)

Eingebaute Operatoren:	Und	# true && false	*- : bool = false*
	Oder	# true || false	*- : bool = true*
	Nicht	# not true	*- : bool = false*

Basisdatentypen: Tupel

Values of possibly different types can be gathered in pairs or more generally in tuples. The values making up a tuple are separated by commas. The type constructor * indicates a tuple. The type *int * string* is the type of pairs whose first element is an integer (of type *int*) and whose second is a character string (of type *string*).

- Werte gleicher und / oder unterschiedlicher Datentypen können zu **Tupeln** zusammengefaßt werden.
- Form: Durch Kommata getrennte Aufzählung in runden Klammern *(Wert1, Wert2, Wert3, ...)*

```
# (22,"Oktober",2002);;
- : int * string * int = (22, "Oktober", 2002)
```

Funktionsdefinition in OCaml

let *name* p_1 ... p_n **=** *expr*

let *name* **= function** p_1 -> ... -> **function** p_n -> *expr*

- **Beispiel A: Nachfolgerfunktion `succ (x)`**

```
let succ = function (x) -> x + 1;;
let succ x = x + 1;;
```

- **Beispiel B: Funktion `prod(x,y) = 3*x + 5*y`**

```
let prod1 = function (x,y) -> 3 * x + 5 * y;;
let prod1a (x,y) = 3 * x + 5 * y;;
let prod2a = function x -> function y -> 3 * x + 5 * y;;
let prod2b x y = 3 * x + 5 * y;;
let prod3 x y = 3.0 *. x +. 5.0 *. y;;
```

- **Parameter als Tupel an eine Funktion vs. Abfolge von Funktionsanwendungen**
- **Typisierung, Typ-Inferenz**

FP: Typisierung und Typ-Inferenz

- Frühe Sprachen wir Lisp waren „ungetypt“; d.h. der Typ eines Objekts war nicht festgelegt (mußte nicht festgeelgt werden)
- Es zeigt sich aber, daß eine „stark getypte“ Sprache bei der Entwicklung großer Softwaresysteme viele Vorteile bietet
  - Tippfehler und logische Fehler können schon zur Übersetzungs- bzw. „Definitionszeit“ von Funktionen abgefangen werden
  - Bei der Spezifikation von Schnittstellen spielen Datentypen eine hervorgehobene Rolle
- **Monomorphe** Typsysteme: Jedes Objekt der Sprache besitzt genau einen Typ. Problem: erhebliche Einschränkung des Programmierers. Beispiel: E/A-Statements in Modula-2: WriteInt, WriteReal, WriteString, ...
- **Polymorphe** Typsysteme: ein Objekt kann mehrere, ggf. sogar unendlich Typen haben. Es wird durch ausgefeilte Typ-Checker gewährleistet , daß zur Laufzeit keine Typfehler auftreten können

FP: Typisierung und Typ-Inferenz

- Typen von Funktionen werden nur soweit nötig eingeschränkt
- Funktionen, die auf beliebigen Typen arbeiten, bekommen eine „anonymen Typ“
- OCaml hat ein sehr leistungsfähiges Typ-Checksystem
  - Bei Bedarf können Typen „von Hand“ eingeschränkt werden
  - Anonymer Typ wird mit ’a, ’b, ’c, ... bezeichnet

Funktionsanwendung: Bindungsbereich für Variablen

- Die Rückgabe der Berechnung eines Funktionsausdrucks heißt *closure* und umfaßt (Namen der formalen Parameter, Funktionsrumpf, Umgebungsvariablen). Der Prozeß der Funktionsanwendung ist damit die Anwendung einer *closure* auf ein Argument, was zu einer neuen *closure* führt.
- Umgebungsvariablen sind innerhalb des Funktionsausdrucks „freie“ Variablen.
- Beispiel A:

```
function x -> x + m;;
let m = 3;;
(function x -> x + m) 4;;
```

- Beispiel B:

```
let m = 3;;
let z = function x -> x + m;;
z 30;;
let m = 3000;;
z 30;;
```

- OCaml hat statische Bindungen (Bindung `Name-Wert` zum Zeitpunkt der Funktionsdeklaration, nicht zum Zeitpunkt der Anwendung (dynamische Bindung)).

Funktionsanwendung: Funktionen höherer Ordnung I

- Funktionen, die Funktionen als Parameter tragen oder Funktionen zurückgeben, heißen **Funktionale** oder **Funktionen höherer Ordnung**
- **Beispiel:** Funktion zur Feststellung, ob zwei `float`-Werte innerhalb eines Wertes ε beieinander liegen

```
# let range epsilon x y =
        ((x > y) && (x -. y) < epsilon)
        || ((x < y) && (y -. x) < epsilon);;
val range : float -> float -> float -> bool = <fun>
# range 0.01  6.999 7.0;;
- : bool = true
```

- Daraus Erzeugung einer neuen Funktion `close`, die durch Spezialisierung der Funktion range für $\varepsilon = 0.1$ entsteht:

```
# let close x y = range 0.1 x y;;
val close : float -> float -> bool = <fun>
# close 6.5 7.0;;
- : bool = false
# close 6.95 7.0;;
- : bool = true
```

Funktionsanwendung: Funktionen höherer Ordnung II

Einschub:

- Werte gleichen Typs können zu **Listen** zusammengefaßt werden.
- Listen können während der Laufzeit wachsen und schrumpfen!
- Form: Durch Semikola getrennte Aufzählung in eckigen Klammern

```
[Wert1; Wert2; Wert3; ...]
```

- [] bezeichnet die **leere Liste**.
- **Beispiel**:

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

- Wichtige Operationen
  - `::` Anfügen eines Elementes an den Kopf einer Liste
  - `@` Konkatenation zweier Listen

```
# 1::2::3::[] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Funktionsanwendung: Funktionen höherer Ordnung III

Operationen auf Listen sind vielfach als Funktional definiert:

```
# #use "C:/Programme/Objective Caml/lib/list.ml";;
val length_aux : int -> 'a list -> int = <fun>
val length : 'a list -> int = <fun>
val hd : 'a list -> 'a = <fun>
val tl : 'a list -> 'a list = <fun>
val nth : 'a list -> int -> 'a = <fun>
val append : 'a list -> 'a list -> 'a list = <fun>
val rev_append : 'a list -> 'a list -> 'a list = <fun>
val rev : 'a list -> 'a list = <fun>
val flatten : 'a list list -> 'a list = <fun>
val concat : 'a list list -> 'a list = <fun>
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
val iter : ('a -> 'b) -> 'a list -> unit = <fun>
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

- Beispiel **`map`**: Wende eine Funktion **`('a -> 'b)`** auf eine Liste **`'a list`** an und erzeuge eine Liste **`'b list`** als Resultat

```
# let l_float = [1.;2.;3.;4.;5.;6.;7.;8.;9.;10.];;
val l_float : float list = [1.; 2.; 3.; 4.; 5.; 6.; 7.; 8.; 9.; 10.]

map sin l_float;;
- : float list =
[0.841470984808; 0.909297426826; 0.14112000806; -0.756802495308; -0.958924274663; -
0.279415498199; 0.656986598719; 0.989358246623; 0.412118485242; -0.544021110889]

# map sqrt l_float;;
- : float list =
[1.; 1.41421356237; 1.73205080757; 2.; 2.2360679775; 2.44948974278;
2.64575131106; 2.82842712475; 3.; 3.16227766017]
```

Funktionsanwendung: Funktionen höherer Ordnung IV

- Die Funktion `close` wurde aus `range` durch Spezialisierung gewonnen, d.h. der parameter `eps` wurde auf einen festen Wert (`0.1`) gesetzt. Ähnliches Beispiel:

```
let add x y = x + y ;;
let add4 x = add 4 x;;
```

- Die Funktionen haben einen Parameter weniger als die jeweilige Ausgangsfunktion, dies ist aber keineswegs immer notwendig. Strukturell anderes Beispiel:

```
let compose f g x = f (g x);;
let h x = 3 + x;;
let j y = 5 * y;;
      # compose h j 3;;
      - : int = 18
      # compose j h 3;;
      - : int = 30
```

Ein erstes OCaml-Programm

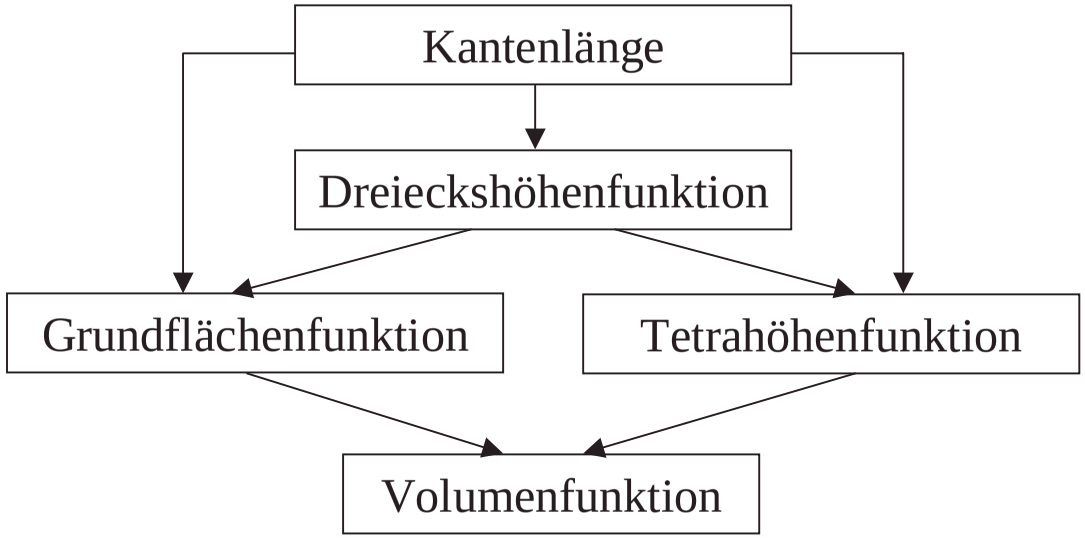
- **Gesucht:** Funktionales OCaml-Programm welches bestimmt ob ein Jahr ein Schaltjahr ist oder nicht
- **Gegeben:** Eingabejahr, Gregorianischer Kalender
- **Algorithmus:** Ein Jahr ist Schaltjahr, wenn das Eingabejahr durch 4 teilbar ist, nicht jedoch durch 100, es sei denn eine Teilung durch 400 ist möglich.
- **OCaml-Programm:**

```
# let isSchaltjahr jahr =
        (jahr mod 4 = 0)
    && not ((jahr mod 100 = 0)
    && (jahr mod 400 <> 0));;

# isSchaltjahr 2000;;
- : bool = true
```

Ocaml-Programm zum Tetraeder-Bespiel

Algorithmus (Wiederholung):

- Volumen V = (1/3) * Grundfläche * Tetraederhöhe
- Grundfläche A = (1/2) * Kantenlänge * Dreieckshöhe
- Tetraederhöhe H = sqrt(Kantenlänge^2 – ((2/3)*Dreieckshöhe)^2)
- Dreieckshöhe h = (Kantenlänge/2)*sqrt(3)



OCaml-Programm für Tetraeder

```
let square x = x *. x;;

let grundflaechenfkt a h =  0.5 *. a *. h;;

let tetrahoehenfkt a h =
      sqrt(square(a) -. square(2. /. 3. *. h));;

let dreieckshoehenfkt a = sqrt(3.) *. a /. 2.;;

let volumenfkt kantenlaenge =
        (1. /. 3.)
        *. grundflaechenfkt kantenlaenge
            (dreieckshoehenfkt(kantenlaenge))
        *. tetrahoehenfkt kantenlaenge
            (dreieckshoehenfkt(kantenlaenge))
```

Fallunterscheidungen / Wächter

Bisher nur eingeschränkte Möglichkeiten der Funktionsdefinition. Insbesondere Problemstellungen wie:

Gesucht: Maximum zweier ganzen Zahlen.

Gegeben: Zwei integer-Zahlen a, b

sind mit den bisherigen Sprachmitteln nicht lösbar, da etwa der

Algorithmus:

Ist a größer als b, dann gebe a zurück, sonst b

ein entsprechendes Sprachmittel erfordert.

Fallunterscheidung If-then-else

- Zur einfachen Fallunterscheidung existiert in OCaml die Struktur

```
if expr1 then expr2 else expr3
```

- Dabei gilt:
  expr1 muss vom Typ bool sein, *expr2* und *expr3* können beliebigen, müssen jedoch vom gleichen Typ sein.

- Der Wahrheitswert der *expr1* bestimmt somit, ob zur Bestimmung des Ausdruck-Wertes *expr2* (true-Fall) oder *expr3* (false-Fall) ausgewertet wird.

Beispiel zur Fallunterscheidung

Damit schließlich:

```
# let max a b = if a > b then a else b;;
val max : 'a -> 'a -> 'a = <fun>

# max 22 45;;
- : int = 45
```

- Zu beachten: Da in der Definition der **max**-Funktion nichts verwendet wird, was explizit auf ganze Zahlen zutrifft, wird mit dem Typ 'a (wesentlich ist das Präfix-Quote) angegeben, daß unter Nutzung des polymorphen Typsystems auch andere Datentypen verwendet werden können, z.B. float

```
# max 2.2 4.5;;
- : int = 4.5
```

Pattern-Matching

Eine der wesentlichen Stärken der OCaml-Programmierung besteht im sog. Pattern-Matching:

Pattern: Muster für Wertdarstellung, oder *Wildcard „_“*
Pattern-Matching: Gegenüberstellung von Wert und Muster

OCaml-Syntax:

```
match expr with
| p1 -> expr1
...
| pn -> exprn
```

Beispiel: Äquivalenz

=	true	false
true	*true*	*false*
false	*false*	*true*

```
# let eq v = match v with
| (true,true)       -> true
| (true,false)      -> false
| (false,true)      -> false
| (false,false)     -> true;;
val eq : bool * bool -> bool = <fun>
```

Pattern-Matching

Dies lässt sich über das Mittel der Wertzuweisung vereinfachen:

```
# let eq v = match v with
| (true, x)  -> x
| (false, x) -> not x
```

- Hier wird innerhalb der Funktion dem Bezeichner x der Wert des zweiten Tupelelementes zugewiesen und dieses ausgewertet.

- Damit wird die Anzahl der Pattern verringert, die Funktion damit übersichtlicher

Weitere Beispiele:

- Mit Wildcard

```
# let isZero n = match n with
  | 0 -> true
  | _ -> false;;
val isZero : int -> bool = <fun>
```

- Mit Wächter-Funktionalität und mit Fehlermeldung

```
# let isPositiv n = match n with
| n when n >= 0  -> true
| n when n < 0   -> false
| _              -> failwith ("????");;
val isPositiv : int -> bool = <fun>
```

Pattern-Matching

Weiteres Beispiel für elegante Programmstrukturen mit Wächtern:

```
let max3 v = match v with
   | (a,b,c) when (a >=b) && (a >= c) -> a
   | (a,b,c) when (b >=a) && (b >= c) -> b
   | (a,b,c) when (c >=a) && (c >= b) -> c
   | _ -> failwith("????");;
```

Fehlermeldung ohne die letzte Zeile wäre :

```
Warning: Bad style, all clauses in this pattern-matching
are guarded.
```

Alternative: geschachtelte `if-then-else` Ausdrücke

```
If a >= b then if a>=c then a
                       else c fi
          else if b>=c then b
                       else c fi
fi
```

Pattern-Matching

- Pattern-Matching auf Listen:

```
# let midof3 v = match v with
  | [_;a;_]   -> a
  | _         -> failwith("wrong nr of elements");;
val midof3 : 'a list -> 'a = <fun>
```

Zu beachten: Wildcard für uninteressante Elemente

```
# let head lst = match lst with
  | x :: tail       -> x
  | []        -> failwith("empty list");;
val head : 'a list -> 'a = <fun>

#head [2;3;4;5];;
- : int = 2
```

Weiteres Beispiel für nichtlinare Rekursion

Fibonacci: Modell für die Entwicklung einer Kaninchenpopulation

Regeln:

- Zum Zeitpunkt i gibt es A_i alte und J_i junge Paare
- Nach dem Übergang zum jeweils nächsten Zeitintervall hat jedes alte Paar ein junges Paar erzeugt. Gleichzeitig wird das junge zu einem alten Paar
- Kaninchen sterben nicht

Damit Modell:

$J_{i+1} = A_i$

$A_{i+1} = A_i + J_i$

Die Gesamt-Kaninchenzahl F_i ergibt sich (für n>1) damit aus der folgenden Beziehung:

$F_i = A_i + J_i$ (Summe aus alten und jungen)

$= A_i + A_{i-1}$

$= (A_{i-1} + J_{i-1}) + (A_{i-2} + J_{i-2})$

$= F_{i-1} + F_{i-2}$

Weiteres Beispiel für nichtlinare Rekursion

Die Rekursion steckt also direkt in der Aufgabenstellung.

- OCaml-Programm:

```
let rec fib n = match n with
    | n when (n = 0) -> 1
    | n when (n = 1) -> 1
    | n when (n >=2) -> fib (n-1) + fib (n-2)
    | _ -> failwith ("unzulaessig");;
```

```
# fib 2;;
- : int = 2
# fib 3;;
- : int = 3
# fib 4;;
- : int = 5
# fib 5;;
- : int = 8
# fib 6;;
- : int = 13
#
```

Geschachtelte Rekursion I

- **`mod`**-Funktion mit einfacher Endrekursion ineffizient:

```
let rec mod1 n d =
      print_string (">");
      if n < d then n
      else mod1 (n - d) d;;
```

- Linearer Aufwand in Größe des Dividenden
- Bessere Lösung unter Ausnutzung der Beziehung
  `n mod d = (n mod 2d) mod d :`

```
let rec mod2 x =
      print_string(">");
      match x with
      | (n,d) when (n < d) -> n;
      | (n,d) when (d <= n) && (n < 2*d) -> (n - d)
      | (n,d) when (2*d <= n) -> mod2 (mod2(n, 2*d),d)
      | _ -> failwith("???");;
```

- *Nur noch logarithmischer* Aufwand in Größe des Dividenden!

Geschachtelte Rekursion II

Definition der Ackermann-Funktion:

A(0, n) = n + 1
A(m+1, 0) = A(m, 1)
A(m+1, n+1) = A(m, A(m+1, n))

1. If x = 0 then A(x, y) = y + 1
2. If y = 0 then A(x, y) = A(x-1, 1)
3. Otherwise A(x, y) = A(x-1, A(x, y-1))

- Beliebter Anwendungszweck: Compiler-Benchmark, siehe z.B. http://www.bagley.org/~doug/shootout/bench/ackermann

```
let rec ack m n =
  if m = 0 then n + 1
  else if n = 0 then ack (m - 1) 1
  else ack (m - 1) (ack m (n - 1));;

let rec ack2 n =  match n with
    | (m, n) when m = 0 -> n + 1;
    | (m, n) when n = 0 -> ack2 (m -1 , 1);
    | (m, n) -> ack2 (m - 1, ack2 (m, n - 1));;

let rec ack3 n =  match n with
    | (0, p) -> (p + 1);
    | (q, 0) -> ack3 (q - 1 , 1);
    | (q, p) -> ack3 (q - 1, ack3 (q, p - 1));;
```

Einbettung rekursiver Funktionen

- Eine Implementierung rekursiver Funktionen, wie sie bisher bei der Fakultätsfunktion oder bei Listen durchgeführt wurde, ist häufig ineffizient: erst nach Erreichung der Terminierungsbedingung wird der Ausdruck berechnet. Wir zeigen dies an zwei Beispielen:

- 1. Beispiel: Ein Aufruf der Fakultätsfunktion fac(4) erzeugt sukzessive folgende Ausdrücke:

```
   fac(4)
=  4 * fac(3)
=  4 * 3 * fac(2)
=  4 * 3 * 2 * fac(1)      (**)
=  4 * 3 * 2 * 1
=  4 * 3 * 2
=  4 * 6
=  24
```

  Es zeigt sich: Es wird zunächst durch mehrfachen rekursiven Aufruf der Term (**) erzeugt; nach Erreichen der Terminierungsbedingung wird der verbleibende Term ausgewertet. Die Anzahl der für die abschließende Auswertung erforderlichen Schritte ist etwa gleich der Anzahl der rekursiven Aufrufe.

Einbettung rekursiver Funktionen

- 2. Beispiel: Es soll eine Funktion implementiert werden, die aus einer Zeichensequenz diejenigen Zeichen extrahiert, die (im Sinne der auf Zeichen geltenden lexikographischen Ordnung) kleiner als ein bestimmtes Zeichen c sind.
- Folgende Implementierung leistet das Gewünschte:

```
# let rec kleiner_als a = match a with
            |(c, []) -> []
            |(c, x::l) when (x < c) -> x::(kleiner_als (c, l))
            |(c, x::l) -> kleiner_als (c, l);;
```

- Für

```
# kleiner_als ('a', ['>';'@';'H';'b']);;
- : char list = ['>'; '@'; 'H']
```

erhalten wir die Folge von Ausdrücken:

```
  kleiner_als ('a', ['>';'@';'H';'b']);;
= `>` :: (kleiner_als ('a', ['@';'H';'b'])
= `>` :: `@`   :: (kleiner_als ('a', ['H';'b'])
= `>` :: `@`   :: `H` ::(kleiner_als ('a', ['b'])
= `>` :: `@`   :: `H` :: []
```

- Auch hier muß also nach Erreichen der Terminierungsbedingung die Liste wieder zusammengefügt werden.

Einbettung rekursiver Funktionen

- Durch Einbettung rekursiver Funktionen erhält man rekursive Funktionen, deren Resultat sofort nach Erreichen der Terminierungsbedingung feststeht.
- Trick: Mitführen der Zwischenergebnisse

Einbettung rekursiver Funktionen (Fakultät)

In den beiden genannten Beispielen erreicht man dies folgendermaßen:

Einbettung der Fakultätsfunktion:

Man implementiere eine Hilfsfunktion zur Berechnung von

```
f(m, n) = m * fac(n)
```

durch

```
f(m, 1) = m
f(m, n) = f(m*n, n-1)
```

und rufe die Fakultätsfunktion `fac(n)` durch `fac(n) = f(1, n)` auf.

Implementierung:

```
let rec f arg = match arg with
        |(m, 1) -> m
        |(m, n) -> f (m*n, n-1);;
let fac2 n = f(1, n);;
```

Dies erscheint zunächst komplizierter; betrachtet man jedoch die Folge der rekursiven Aufrufe, so erkennt man, dass die Berechnung mit dem letzten rekursiven Aufruf beendet ist:

Einbettung rekursiver Funktionen (Fakultät)

```
fac2 4  = f(1,4)
        = f(4,3)
        = f(12,2)
        = f(24,1)
        = 24
```

- Verallgemeinerung: Einbettung einer rekursiven Funktion erreicht man, indem man das zu lösende Problem verallgemeinert.
- Die gesuchte Funktion ist dann ein Spezialfall des allgemeinen Problems.
- Im Falle der Fakultätsfunktion bedeutet dies für die allgemeine Problemstellung: Multipliziere zu einer Zahl `m` das Produkt der Zahlen von `1` bis `n`. Die Fakultätsfunktion ergibt sich dann im Spezialfall `m = 1`
- Einbettung von Beispiel 2: Die Einbettung erreicht man durch folgende Verallgemeinerung:

  Löse das Problem, an eine gegebene Zeichensequenz `l1`, diejenigen Zeichen einer Liste `l2` anzufügen, die kleiner als ein gegebenes Zeichen `c` sind: `g(l1, c, l2)`

  Aufruf der gesuchten Funktion: `kleiner_als (c, l) = g([], c, l)`

Einbettung rekursiver Funktionen (Suche in Sequenzen)

Die Implementierung ergibt:

```
let rec g arg = match arg with
      | (l1, c, []) -> l1
      | (l1, c, (x :: l2)) when x < c -> g ((x::l1), c, l2)
      | (l1, c, (x :: l2)) -> g (l1, c, l2);;
let kleiner_als (c, l) = g ([], c, l);;
```

Durch Einbettung ist es häufig möglich, nichtlinear rekursive Funktionen zu linearisieren und daduch eine erheblich zeit- oder speichereffizientere Implementierung anzugeben.

Beispiel: Einbettung der nichtlinearen Fibonacci-Funktion (mit maximal $\mathbf{2^n}$ Aufrufen ergibt eine lineare Funktion mit maximal n Aufrufen).

Einbettung rekursiver Funktionen: Fibonacci-Funktion

Die Fibonacci-Funktion läßt sich durch die nichtlinear rekursive Rechenvorschrift:

```
let rec fib n =
        | n when (n = 1) || (n = 2) -> 1
        | n -> fib (n-1) + fib (n-2);;
```

Im allgemeinen vedoppelt sich die Anzahl der rekursiven Aufrufe bei jedem rekursiven Aufruf und es können sich (abgesehen von konstanten multiplikativen Faktoren) bis zu 2^n Aufrufe ergeben.

Durch Einbettung ist es nun möglich, die Fibonacci-funktion in linearer Zeit zu berechnen. Grundidee ist hierbei, bei `n = 1` zu beginnen und jeweils bekannte Resultate zu verwenden.

Wir definieren:

```
let fib_lin n = f (n, 1, 2, 1);;
let rec f (n, k, a, b) =
        |(n, k, a, b) when k = n -> a
        |(n, k, a, b) -> f(n, k+1, a+b, a);;
```

Einbettung rekursiver Funktionen: Fibonacci-Funktion

Erläuterung:

- der Parameter `n` der Funktion `f` entspricht dem Parameter `n` der rekursiven Variante
- der Parameter `k` ist ein Zähler, von `1` bis `n`
- die beiden Parameter `a` und `b` speichern jeweils den Wert `fib(k-1)` und `fib(k-2)` ab

Grundidee der Einbettung: Da die rekursive Rechenvorschrift zur Berechnung der Fibonacci-Funktion jeweils auf den „Vor-“ und „Vorvorgänger“ zurückgreift, benötigen wir, im Gegensatz zu den Einbettungen linearer Rekursionen, zwei HIlfsparameter `a` und `b`. Außerdem beginnt die eingebettete Lösung die Rekursion nicht mit `n`, sondern mit `1` und zählt dann hoch bis `n;` Grund: Für `n=1` und `n=2` sind die Resultate bekannt.

Übung: Einbettung der Binomialkoeffizienten `Bi(n,k):`

Lösungsidee: Verwendung eines Zählers `i` von `0` bis `n` und einer Sequenz, die die Binomialkoeffizienten `Bi(i-1,k),0 <= k <= (i-1),` enthält, als weitere Parameter.

Datentypen

Definition: Unter einem Datentyp versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit (Informatik-Duden)

- Die bisher benutzten OCaml-Datentypen **`bool, int, float, char`** (nicht jedoch Tupel und Listen) werden als *primitive Datentypen* bezeichnet. Sie sind auch in den meisten anderen Programmiersprachen verfügbar.

- Weitere Datentypen lassen unter zu Hilfenahme gewisser Konstruktionsvorschriften bilden: zusammengesetzte Datentypen, Datenstrukturen

- Zu beachten ist, daß bereits einige wenige Konstruktionsvorschriften ausreichen um beliebig umfangreiche Datenstrukturen aufzubauen

1. Aufzählungstypen

- ... sind Datentypen, welche durch eine endliche Aufzählung (Enumeration) aller zugehörigen Elemente erstellt werden.

- Schematisch: `dattype d = Enum_1 | Enum_2 | ... | Enum_n`

OCaml-Beispiel:

```
# type color = Red | Green | Blue;;
type color = Red | Green | Blue
```

Zu beachten:

- Schlüsselwort ***type*** für die Typdefinition
- Selbstdefinierte Aufzählungen müssen großen Anfangsbuchstaben besitzen.
- Datentyp-Bezeichner müssen kleinem Anfangsbuchstaben besitzen
- Datentyp bool ist Aufzählungstyp (type bool = true | false)

2. Produkttypen

- Zusammensetzung mehrerer (auch unterschiedlicher) Datentypen zu **einem** neuen Typ.
- Schematisch: `dattype t = (t1, ... , tn)`
- T ist das Produkt der Typen T1, ... , Tn dar (T1 * ... * Tn), wobei letztere als die **Komponenten** des neuen Datentyps bezeichnet werden.
- Formen:
  - Tupel
  - Records

Tupel

- Tupel treten in OCaml als Argumentlisten von Funktionen auf (siehe früher):

```
# let add (x,y) = x + y ;;
val add : int * int -> int = <fun>
```

können aber auch als Rückgabewert definiert werden:

```
# let swap (x,y) = (y,x);;
val swap : 'a * 'b -> 'b * 'a = <fun>
```

- Darüberhinaus können Tupel in OCaml auch explizit als Typ definiert werden:

```
# type complex = float*float;;
type complex = float * float
```

(Für komplexe Zahlen siehe http://pauillac.inria.fr/ocaml/htmlman/libref/Complex.html)

Tupel

Beachte: nach Deklaration erfolgt *keine* automatische Typzuordnung:

```
# (7.5, 8.0);;        - : float * float = (7.5, 8.)
```

Betrachte folgende Funktion:

```
# let add (x:complex) (y:complex) = (fst(x) +. fst(y), snd(x) +. snd(y));;
val add : complex -> complex -> float * float = <fun>
```

Zur Erklärung:

1. Durch Verwendung des „:“ wird der Argumenttyp explizit festgeschrieben:

   ```
   # let f x = x;;                  val f : 'a -> 'a = <fun>
   ```

   Ist polymorph verwendbar,

   ```
   # let f (x:int) = x;;            val f : int -> int = <fun>
   ```

   dagegen nur für **`int`**.

2. Mit **`fst`** und **`snd`** existieren zwei native Funktionen, die auf die Elemente eines Paars („2-Tupel“) explizit zugreifen zu können.

Tupel und Typkonstruktoren

Da der Typchecker hier nicht zwischen den *strukturell gleichen Typen* float*float und complex unterscheidet kann, sind Aufrufe der folgenden Form zulässig:

```
# add (3.4,5.6) (3.5,6.7);;          - : float * float = (6.9, 12.3)
```

Will man eine striktere Typverwendung, so ist der Typ anders zu definieren: besser, wenn auch aufwendiger, mittels **Konstruktordefinition**:

```
# type complex = Complex of float*float;;
type complex = Complex of float * float
```

Damit ist zur Typanwendung die Nutzung des Konstruktors obligatorisch:

```
# let x:complex = (4.5, 6.7);;
This expression has type float * float but is
here used with type complex

# let x:complex = Complex(4.5, 6.7);;
val x : complex = Complex (4.5, 6.7)
```

Tupel und Typkonstruktoren

Dies ist auch in der Definition der **`add`**-Funktion möglich:

```
# let add (Complex(a,b)) (Complex(c,d)) = Complex(a +. c, b +. d);;
val add : complex -> complex -> complex = <fun>
```

Was bewirkt, daß

```
# add (3.4,5.6) (3.5,6.7);;
This expression has type float * float but is here used with type
complex
```

Nur noch in der Form:

```
# add (Complex(3.4, 5.6)) (Complex(3.5, 6.7));;
- : complex = Complex (6.9, 12.3)
```

möglich ist.

Zu beachten: **Konstruktoren** müssen mit **G**roßbuchstaben beginnen.

Tupel und Typkonstruktoren

- Auf die Komponenten von Tupeln kann nicht ohne weiteres zugegriffen werden.
- Zwar existieren mit den Funktionen **`fst`** und **`snd`** Zugriffsmöglichkeiten für Paare, für Tupel mit mehr Elementen gibt es solche **Selektoren** jedoch nicht mehr.
- Möglich ist hier der Zugriff über Pattern-Matching:

```
# type student = int * string * string;;
type student = int * string * string

#  let vorname d = match d with (_,_,v) -> v;;
val vorname : 'a * 'b * 'c -> 'c = <fun>

# vorname (5443453, "Mustermann", "Hans");;
- : string = "Hans"
```

Records

- In OCaml ist es möglich, den Komponenten eines Tupels Namen zu geben. Dies führt zum Begriff des **Record**s:

- Definition: Ein Record ist eine Datenstruktur, die aus einem Tupel besteht, dessen einzelne Komponenten mit Namen markiert sind.

- Schematisch:

```
dattype d = {s1:d1; s2:d2; ... ; sn:dn}
```

- Die `s1, ... , sn` bezeichnen dabei als Selektoren die einzelnen Komponenten von `d`.
- In OCaml gilt: Kleiner Anfangsbuchstabe für Typ und Bezeichner.
- Ein wesentlicher Vorteil der Records gegenüber den Tupeln ist, dass die Komponenten damit über ihren Bezeichner zugreifbar werden.

Records

- Beispiel:

```
# type student = {matrikel : int; nachname : string;
                  vorname : string};;
type student = { matrikel : int; nachname : string;
vorname : string; }

# let stud1 : student = {matrikel = 5443454;
      nachname ="Musterfrau"; vorname = "Angela"};;
val stud1 : student = {matrikel = 5443454; nachname =
"Musterfrau"; vorname = "Angela"}
```

- Fehlende Elemente werden vom Interpreter angemahnt:

```
# let stud2 : student = {matrikel = 4242424};;
Some record field labels are undefined: nachname
vorname
```

3. Summenbildung

- Bei den vorher beschriebenen Datentypen sind stets alle Komponenten gleichzeitig vorhanden. Der gesamte Wertebereich entsteht durch *Konjunktion* der einzelnen Komponentenbereiche (Produktbildung).
- Doch auch die *Disjunktion* von Komponentenbereichen ist sinnvoll: Summenbildung, wenn es innerhalb eines Types mehrere Alternativen für die Komponenten gibt.
- Schematisch:

```
dattype d = c1 of d1 | c2 of d2 | ... | cn of dn
```

- Die durch "|" getrennten Teile werden als **Varianten** bezeichnet.
- `c1, ... , cn` bezeichnen die einzelnen Konstruktoren von `d` und sind frei wählbar (**großer** Anfangsbuchstabe in OCaml)
- Hinweis: Die Aufzählungstypen sind Spezialfälle (einelementig, nullstellige Konstruktoren) der Summenbildung.

Summenbildung

- Beispiel: Typbeschreibung für zweidimensionale geometrische Objekte:

```
# type point = {x:float;y:float};;
type point = { x : float; y : float; }

# type obj2D = Point of point | Line of point*point |
                Circle of point * float;;
type obj2D = Point of point | Line of point * point |
Circle of point * float
```

- Anwendung:

```
#   let x = Circle({x=2.0;y=3.0},4.0);;
val x : obj2D = Circle ({x = 2.; y = 3.}, 4.)
```

Rekursiv definierte Datentypen

- Eine wichtige Anwendung der Summenbildung besteht in der Definition von *rekursiven Datentypen*.
- Ähnlich der Beschreibung der rekursiven Funktionen zeichnen sich rekursive Datentypen darin aus, daß der definierte Datentyp auf der Konstruktorseite zur Definition benützt wird.

Schematisch:

```
dattype d = c0 of d' | ... | cn of d''
```

- `d'` bezeichnet den terminalen Datentyp mit Konstruktor `c0` und `d''` einen Datentyp mit Konstruktor `cn`, in dessen Definition wiederum `d` vorkommt.
- Mit der Rekursion wird insbesondere die Definition von Datentypen mit abzählbar unendlichen Wertebereichen möglich

Rekursiv definierte Datentypen

- Beispiel: Definition eines Datentyps für natürliche Zahlen

```
# type nat = Null | Succ of nat;;
type nat = Null | Succ of nat
```

- Anwendung:

```
# let i = Succ;;                val i : nat = Null
# let i = Succ(Null);;          val i : nat = Succ Null
```

- Interessant: Definition einer Funktion, welche `int` in `nat` umwandelt:

```
# let rec nat_of_int x = match x with
    | 0 -> Null
    | n -> Succ(nat_of_int(n-1));;
val nat_of_int : int -> nat = <fun>

# nat_of_int 10;;
- : nat =
Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ Null))))))))))
```

Rekursiv definierte Datentypen

- Weiteres Beispiel: Definition einer Liste über floats:

```
# type floatlist = Empty | Front of float*floatlist;;
type floatlist = Empty | Front of float * floatlist
```

- Anwendung:

```
# Empty;;
- : floatlist = Empty

# let lst = Front(1.1,Front(2.2,Front(3.3,Empty)));;
val lst : floatlist = Front (1.1, Front (2.2, Front (3.3,
Empty)))

(* Funktion die Anzahl der Elemente zählt: *)
# let rec count x = match x with
        | Empty -> 0
        | Front(x,tl) -> 1 + count(tl);;
val count : floatlist -> int = <fun>

# count lst;;
- : int = 3
```

Rekursiv definierte Datentypen

Parametrisierte Typen:

- Die vorab vorgestellte Listendefinition ist mit der Beschränkung auf float sehr *eng*. Stattdessen wäre es oftmals nützlich, dieselbe Typdeklaration für unterschiedliche (wählbare) Datentypen verwenden zu können. Dies lässt sich durch die Verwendung von Typvariablen erreichen:

```
# type 'a list = Empty | Front of 'a * ('a list);;
type 'a list = Empty | Front of 'a * 'a list

# let lst = Empty;;
val lst : 'a list = Empty

# let rec count x = match x with
  | Empty -> 0 | Front(x,tl) -> 1 + count(tl);;
val count : floatlist -> int = <fun>

# let lst = Front(1.1,Front(2.2,Front(3.3,Empty)));;
val lst : float list = Front (1.1, Front (2.2, Front (3.3, Empty)))
```

Rekursiv definierte Datentypen

Sehr wichtige Listen sind die sogenannten LIFO-(Last In First Out) bzw. FIFO-(First In First Out)-Listen. Erstere finden z.B. bei bei Hemdenstapeln oder *Stacks* Verwendung, letztere bei Druckerwarteschlangen oder Bushaltestellen.

- LIFO-Liste: Durch die Typdeklaration

```
# type 'a liFo = Empty | Prepend of 'a * ('a liFo);;
```

  wird eine rekursive Struktur definiert. Wählen wir als Trägermenge von `a` die natürlichen Zahlen `int`, so hat eine typische Listen folgendes Aussehen:

```
# let l1 = Prepend(1, Prepend(2, Empty));;
```

- Nur auf das zuletzt angefügte Element dieser Liste kann, etwa durch Pattern-matching direkt zugegriffen werden:

```
# let first ls = match ls with
        | Empty -> failwith ("undefined")
        | Prepend (x, rest) -> x;;
val first : 'a liFo -> 'a = <fun>
# first l1;;
- : int = 1
```

Von LiFo zu FiFo

- Wie **`first`** das erste Element der Sequenz direkt zugreifen kann, so kann durch eine analoge Funktion **`rest`** leicht auf die Liste ohne das erste Element gegriffen werden.
- FIFO-Liste: Die wesentlichen Operationen dieser Struktur lassen sich aus der LIFO-Struktur konstruieren, indem man eine rekursive Funktion **`last`** definiert, die auf das letzte Element der Liste zugreift:

```
# let rec last ls = match ls with
        | Empty -> failwith ("undefined")
        | Prepend (x, Empty) -> x
        | Prepend (x, ll) -> (last ll);;
val last : 'a liFo -> 'a = <fun>
```

- Funktion **`lrest`** gibt die Liste ohne das letzte Element zurück:

```
# let rec lrest ls = match ls with
        | Empty -> failwith ("undefined")
        | Prepend (x, Empty) -> Empty
        | Prepend (x, ll) -> Prepend (x, lrest ll);;
val lrest : 'a liFo -> 'a liFo = <fun>
```

Baumartige Strukturen

Literaturhinweise:

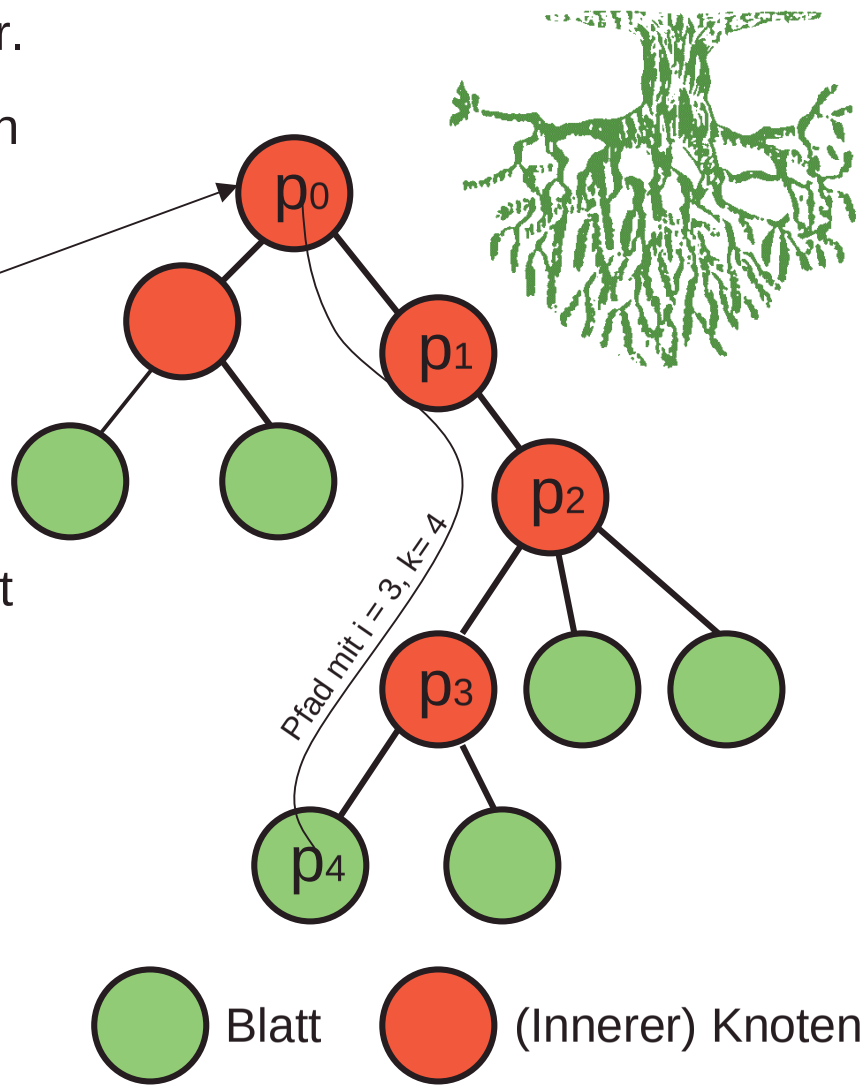
Ottmann/Widmayer: Algorithmen und Datenstrukturen, Band 70 der Reihe Informatik, BI-Wissenschaftsverlag, Mannheim, 4. Auflage, 2002

Wirth: Algorithmen und Datenstrukturen – Pascal-Version, Teubner Verlag, 2000

- Bäume gehören zu den wichtigsten Datenstrukturen in der Informatik: Codebäume, Suchbäume, Syntaxbäume, Entscheidungsbäume, Dateibäume ...
- Bäume bestehen aus Knoten unterschiedlichen Typs; die Verbindung zwischen zwei Knoten wird als *Kante* bezeichnet.

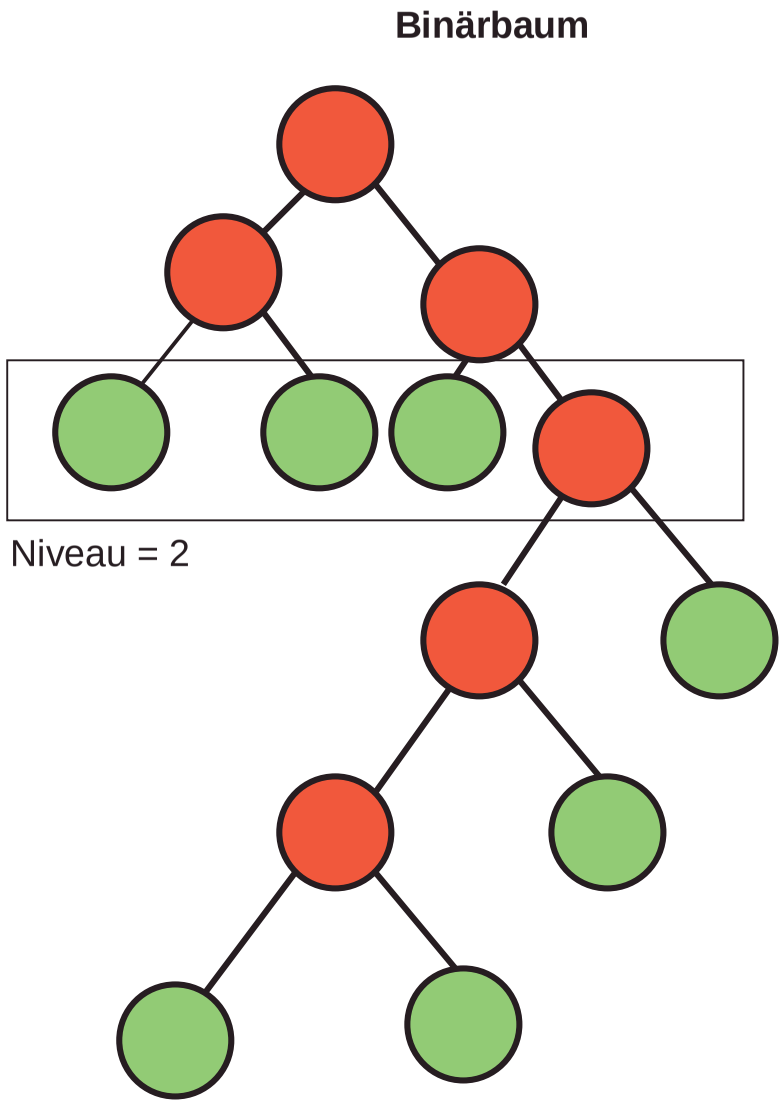
Definitionen:

- Baum ist verallgemeinerte Listenstruktur.
- Ein Element (Knoten) hat nicht nur einen Nachfolger (wie im Fall der Liste), sondern eine begrenzte Anzahl von *Söhnen*.
- „Oberster“ Knoten ist die Wurzel des Baums (kein Vorgänger oder *Vater*)
- Folge $p_0 ... p_k$ von Knoten für die gilt: p_{i+1} ist Sohn von p_i, $0 <= i < k$ heißt Pfad mit der Länge k
- Ist unter den Söhnen eine Ordnung definiert (so daß man vom 1., 2., 3., ... Sohn sprechen kann), so nennt man Baum *geordnet*.
- *Ordnung des Baums*: Maximale Anzahl von Söhnen eines Knotens.



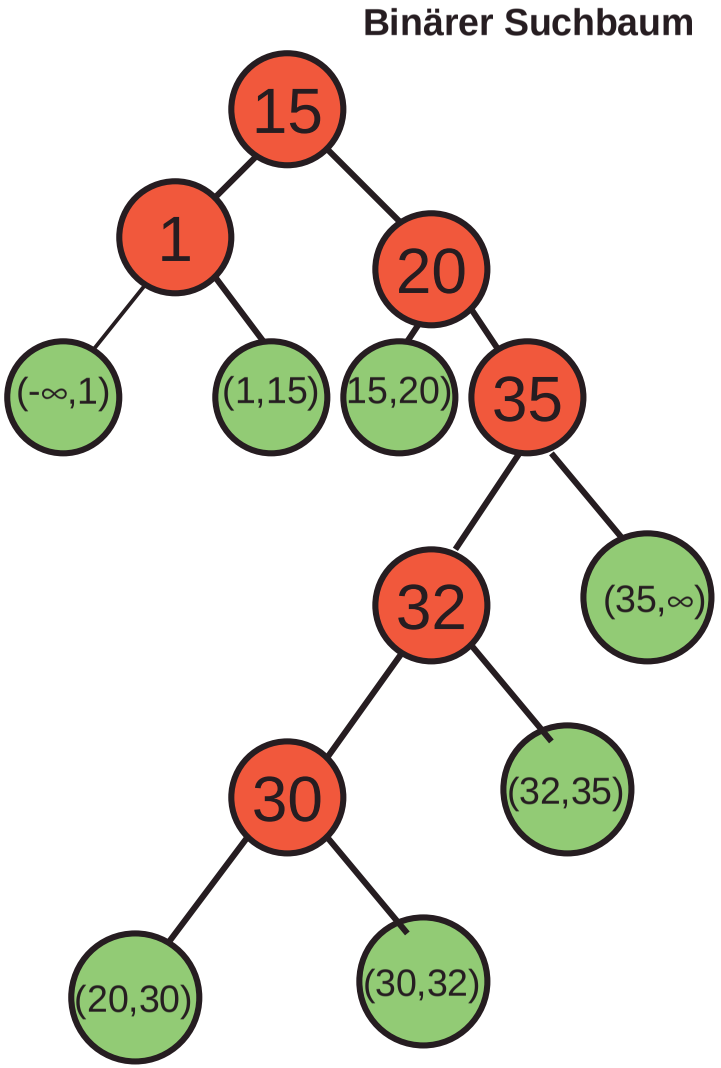
Definitionen:

- Geordnete Bäume der Ordnung 2 heißen *Binärbäume,* typischerweise verlangt man, daß ein Knoten zwei Söhne oder keinen Sohn haben soll (Blatt)
- Beim Binärbaum spricht man vom *linken* und *rechten* Sohn
- Höhe *h* des Baums: Maximaler Abstand eines Blatts von der Wurzel
- Tiefe *t* eines Knotens ist Abstand zur Wurzel, also Anzahl der Kanten auf dem Weg von Wurzel zu Knoten
- Knoten eines Baums gleicher Tiefe bilden ein *Niveau*



Eigenschaften:

- Bäume erhalten ihre Bedeutung für die Informatik, weil sie *Schlüssel* speichern – in den Knoten (Suchbäume) oder in den Blättern (Blattsuchbäume)
- Schlüssel werden so gespeichert, daß sie sich leicht wiederfinden lassen
- Wichtigste drei Operationen: *Suchen*, *Einfügen* und *Entfernen*
- Beim Suchbaum gilt für jeden Knoten Schlüssel im *linken Teilbaum* von p sind sämtlich *kleiner* als der Schlüssel von p und dieser ist kleiner als sämtliche Schlüssel im rechten Teilbaum von p

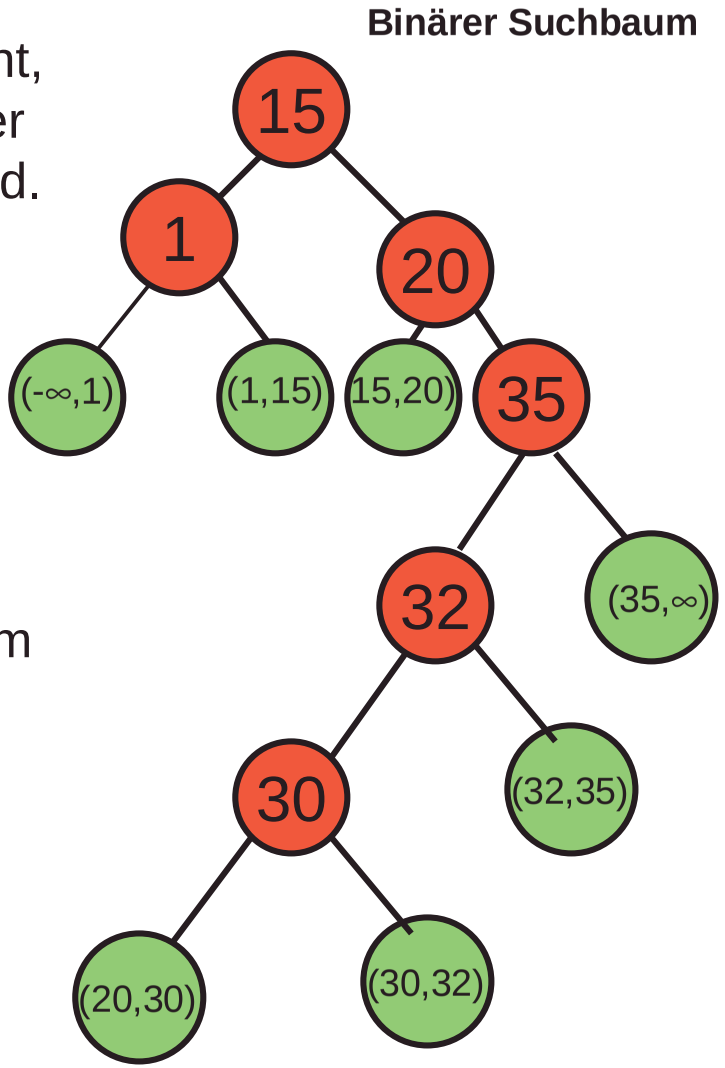


Eigenschaften:

- Suchalgorithmus: Ein Element *x* wird gesucht, indem bei der Wurzel begonnen wird und der dort gespeicherte Schlüssel *s* verglichen wird.

1. `Ist x = s dann Ende`
2. `Ist x < s dann Suche im linken Teilbaum`
3. `Ist x > s dann Suche im rechten Teilbaum`

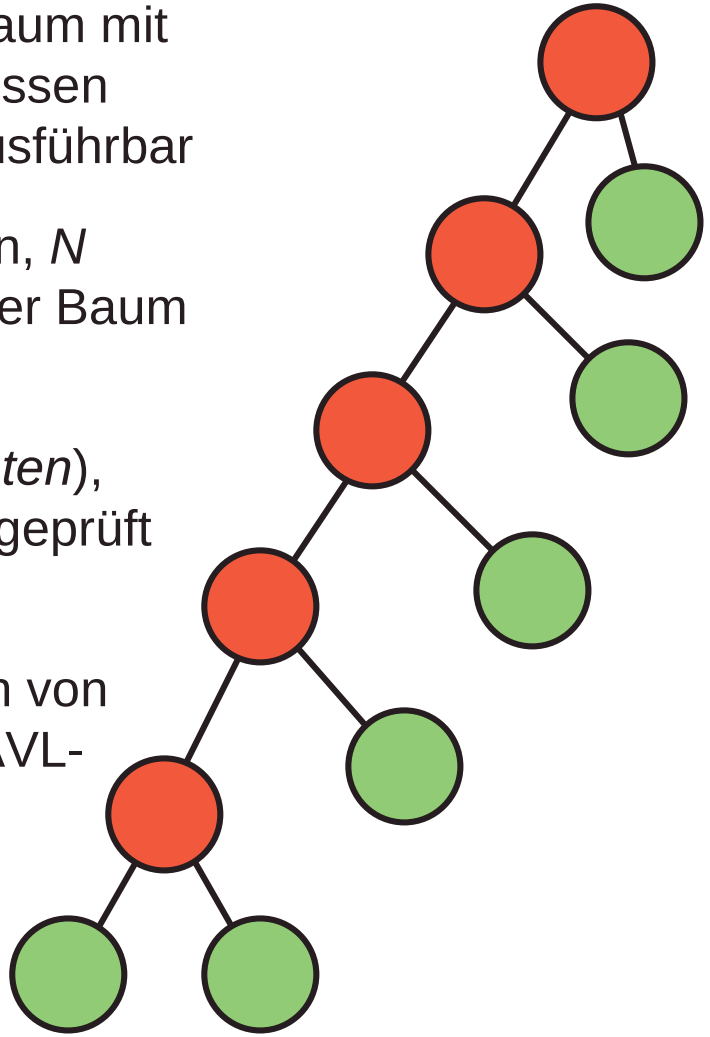
- Ist *x* nicht im Baum, dann „Landung“ in einem Blatt, das das *Intervall* beschreibt, welches den Schlüssel enthält



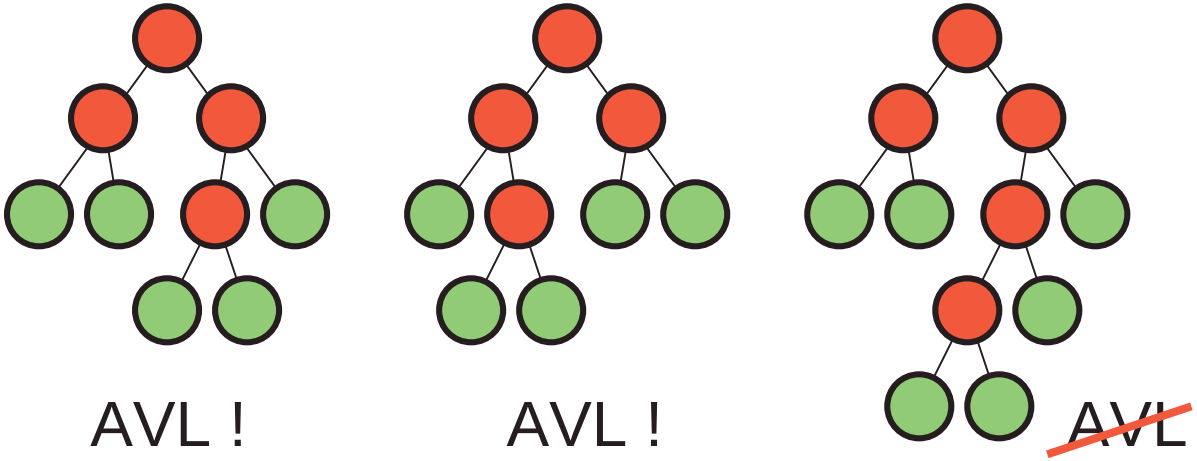
Eigenschaften

- Suchen, Einfügen, Löschen in einen binären Baum mit N Schlüsseln ist im statistischen Mittel (= gemessen über alle solche Bäume) in O(ld *N*) Schritten ausführbar
- Im „worst case“ kann es jedoch erforderlich sein, *N* Schritte ausführen zu müssen, wenn nämlich der Baum zu einer linearen Liste degeneriert ist
- Deshalb: Aufstellen von Bedingungen (*Invarianten*), deren Einhaltung bei den Schlüssel-Operation geprüft wird und die ein Degenerieren verhindern
- Erster Vorschlag zur Balancierung von Bäumen von Adelson, Velskij & Landis: AVL-Bäume. Beim AVL-Baum gilt für jeden Knoten *p*:
  - **Höhe des linken Teilbaums unterscheidet sich von der Höhe des rechten Teilbaums höchstens um 1**

Degenerierter Binärbaum
= lineare Liste mit N Elementen



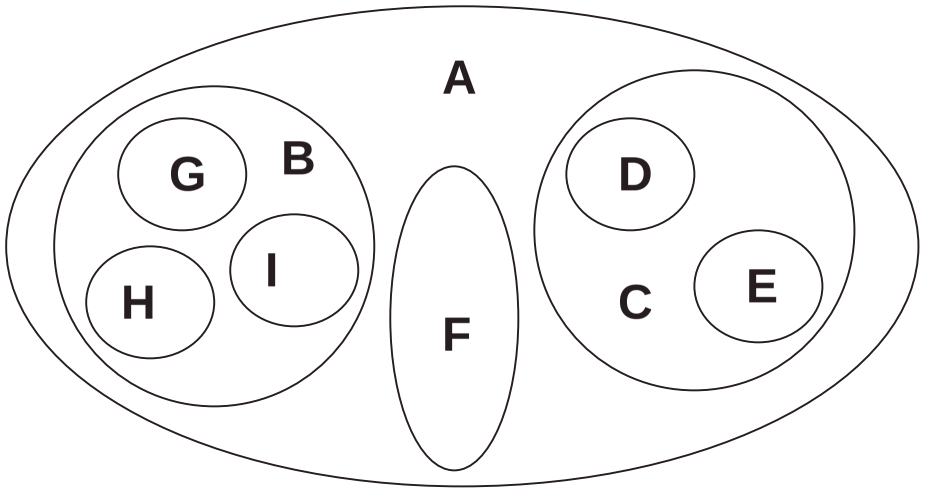
Balancierte Bäume: AVL



- AVL-Höhenbedingung stellt sicher, daß Bäume mit *N* inneren Knoten und *N* + 1 Blättern eine Höhe von O(log *N*) haben
- Umgekehrt kann man zeigen: ein AVL-Baum mit Höhe *h* hat mindestens fib(*h* + 1) Blätter!

Darstellung von Baumstrukturen

- Als Graph (wie vor)
- Als geschachtelte Mengen
- Als geschachtelter Klammerausdruck
  (A (B (G, H, I),C (D, E), F))



Darstellung von baumartigen Strukturen in OCaml

- Offensichtlich: Ein Baum kann als rekursive Struktur aufgefaßt werden, weil die Zweige des Baumes selbst wieder als („kleinere“) Bäume angesehen werden können.
- In funktionalen Sprachen lassen sich baumartige Strukturen z.B. über folgende Typdefinition definieren:

```
# type 'a tree = Empty | Tree of 'a tree * 'a  * 'a tree;;
type 'a tree = Empty | Tree of 'a tree * 'a * 'a tree
```

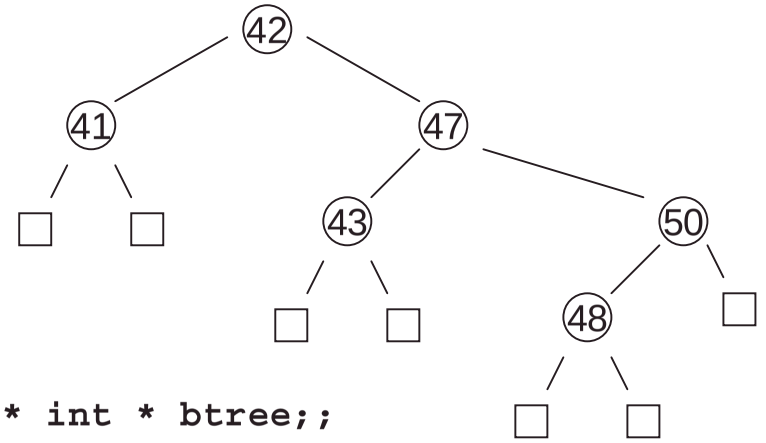
- Oder z.B.

```
# type 'a tree = Node of 'a * 'a tree * 'a tree | Leaf;;
type 'a tree = Node of 'a * 'a tree * 'a tree | Leaf
```

- Im ersten Fall hat der Konstruktor `Tree` hierbei drei Argumente: den linken und rechten Teilbaum vom Typ `tree` und das Knotenelement `'a`. Die Elemente dieser baumartigen Struktur haben dann beispielsweise die Gestalt:

```
# let tree1 = Tree (Empty, 2, Empty);;
val tree1 : int tree = Tree (Empty, 2, Empty)
# let tree2 = Tree (tree1, 3, Tree (Empty, 1, tree1));;
val tree2 : int tree =
Tree (Tree (Empty, 2, Empty), 3, Tree (Empty, 1, Tree
                                       (Empty, 2, Empty)))
```

OCaml-Typdeklaration für Binärbaume



Beispielbaum mit Schlüsseln vom Typ `int`:

```
# type btree = Empty | Node of btree * int * btree;;
type btree = Empty | Node of btree * int * btree
```

Und damit:

```
# let binbaum = Node(Node(Empty, 41, Empty), 42,
                     Node(Node(Empty, 43, Empty),
                     47, Node(Node(Empty,48,Empty),50,Empty)));;
val binbaum : btree =
  Node (Node (Empty, 41, Empty), 42,
   Node (Node (Empty, 43, Empty), 47,
    Node (Node (Empty, 48, Empty), 50, Empty)))
```

OCaml-Typdeklaration für Binärbaume

- Durch die Verwendung einer Typvariablen kann ein Binärbaum auch über wählbare (aber einheitliche) Datentypen definiert werden:

```
# type 'a btree = Empty | Node of 'a btree * 'a * 'a btree;;
type 'a btree = Empty | Node of 'a btree * 'a * 'a btree
```

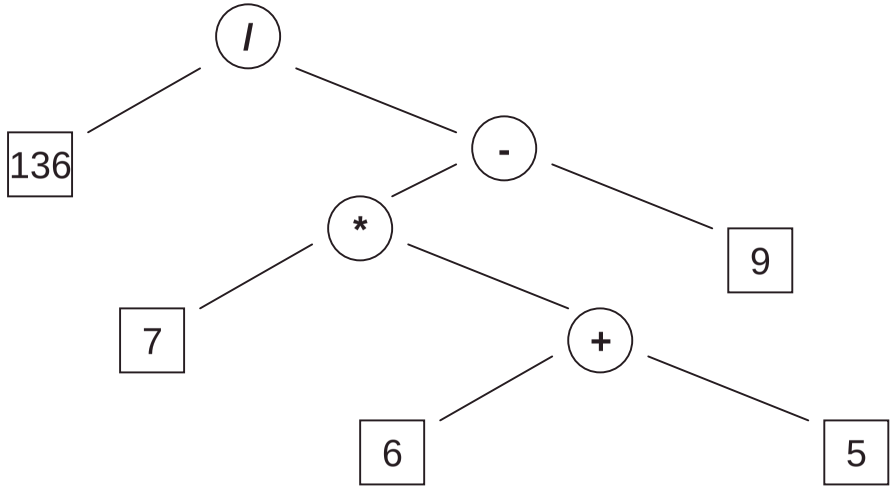
- Voriger Beispielbaum kann ohne Änderung verwendet werden:

```
# let binbaum = Node(Node(Empty, 41, Empty), 42,
                     Node(Node(Empty, 43, Empty), 47,
                     Node(Node(Empty,48,Empty),50,Empty)));;
val binbaum : int btree =
                     Node (Node (Empty, 41, Empty), 42,
                      Node (Node (Empty, 43, Empty), 47,
                       Node (Node (Empty, 48, Empty), 50, Empty)))
```

OCaml-Typdeklaration für Binärbaume

- Falls die Blätter des Baumes andere Informationen (abweichender Datentyp) tragen sollen, als die **inneren Knoten** (Nicht-Blätter), kann dies mit einer geeigneten Definition beschrieben werden:

```
# type ('a,'b) btree = Empty of 'a | Node of ('a,'b) btree
                                     * 'b * ('a,'b) btree;;
type ('a, 'b) btree =
                  Empty of 'a
                | Node of ('a, 'b) btree * 'b * ('a, 'b) btree
```



- Baum repräsentiert arithmetischen Ausdruck *136/(7*(6 + 5) – 9)*
- Beachte, daß **`Empty`** nun nichtleeres Blatt bezeichnet, besser wäre, hier **`Leaf`** zu schreiben (bzw. Operator und Operand) !

OCaml-Typdeklaration für Binärbaume

- Zur Umsetzung in Ocaml: Datentyp für die inneren Knoten....

```
# type op = Plus | Minus | Mult | Divide;;
type op = Plus | Minus | Mult | Divide
```

- ... und der Baum für den Ausdruck:

```
# let arithtree =
      Node(Empty 136, Divide,
      Node(Node(Empty 7, Mult,
      Node(Empty 6, Plus, Empty 5)), Minus ,
                                    Empty 9));;
 val arithtree : (int, op) btree =
      Node (Empty 136, Divide,
      Node (Node (Empty 7, Mult, Node
      (Empty 6, Plus, Empty 5)), Minus, Empty 9))
```

OCaml-Typdeklaration für Binärbaume

- Zur Auswertung des definierten arithmetrischen Ausdrucks lässt sich eine entsprechende Funktion zur Auswertung des Ausdrucks definieren:

```
# let rec eval v = match v with
          | Empty x -> x
          | Node(left, Plus, right)->(eval left) + (eval right)
          | Node(left,Minus,right)->(eval left) - (eval right)
          | Node(left, Mult, right)->(eval left) * (eval right)
          | Node (left, Divide, right)->(eval left) / (eval right)
val eval : (int, op) btree -> int = <fun>
```

- Für unseren Ausdruck ergibt sich damit:

```
# eval arithtree;;
- : int = 2
```

- *Übung*: Man baue analog der Funktion `set_of_list` (nächsten Folien) aus einem in einer Liste repräsentierten korrekten arithmetischen Ausdruck wie [136;/;(;7;*;(;6;+;5;);–;9;)] einen Baum und werte ihn aus!

Aufbau ungeordneter, unbalancierter Binärbäume

- **Grundlegende Operation: Aufbau von Bäumen aus einer Liste**

```
type 'a btree = Node of 'a * 'a btree * 'a btree | Leaf;;

(* Unbalancierte ungeordnete Binärbäume *)
        let empty = Leaf;;

        let insert_ub x t = Node (x, Leaf, t);;

        let rec set_of_list_ub = function
                []  -> empty
              | x :: l -> insert_ub x (set_of_list_ub l);;

        let l = [7;5;9;11;3];;

        # set_of_list_ub l;;
        - : int btree =
        Node (7, Leaf,
        Node (5, Leaf, Node (9, Leaf, Node (11, Leaf, Node
                                                (3, Leaf, Leaf)))))
```

Aufbau geordneter, unbalancierter Binärbäume

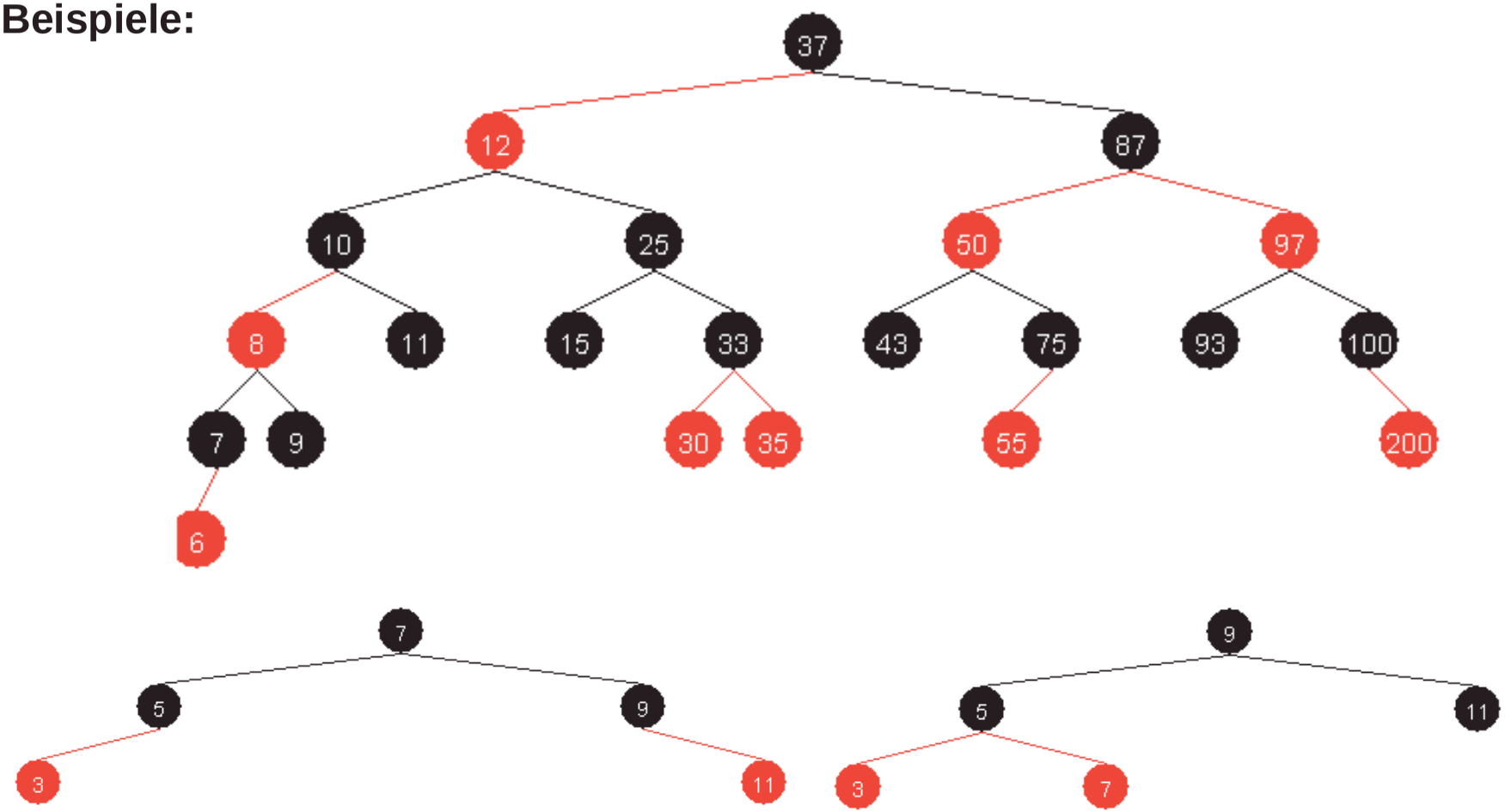
```
type 'a btree = Node of 'a * 'a btree * 'a btree | Leaf;;

(* Unbalancierte geordnete Binärbäume                    *)
(* Für alle Knoten (x, Links, Rechts) gilt:              *)
(* Alle Schlüssel der linken Söhne sind kleiner          *)
(* alle Schlüssel der rechten Söhne sind größer als x    *)

let empty = Leaf;;
let rec insert_ob x = function
  Leaf -> Node (x, Leaf, Leaf)
| Node (y, left, right) -> if x < y then Node (y, insert_ob x left, right)
                           else if x > y then Node (y, left, insert_ob x right)
                                else Node (y, left, right);;
 let rec set_of_list_ob = function
           [] -> empty
           | x :: l -> insert_ob x (set_of_list_ob l);;

#  set_of_list_ob l;;
- : int btree =
Node (3, Leaf,
   Node (11, Node (9, Node (5, Leaf, Node (7, Leaf, Leaf)), Leaf), Leaf))
```

Aufbau balancierter Rot-Schwarz-gefärbte Bäume I

- Binärbäume sind effiziente Datenstrukturen, sofern sie balanciert (und also nicht weitgehend zu Listen degeneriert) sind
- Die Balancierung z.B. zu AVL-Bäumen ist relativ aufwendig (siehe später)
- Einfacher zu balancieren sind sogenannte „Red-Black-trees“. RB-Bäume wurden 1978 publiziert
  (L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In 19th Annual Symposium on Foundations of Computer Science, pages 8--21, Long Beach, Ca., USA, October 1978. IEEE Computer Society Press.),
  basieren allerdings auf einem Spezialfall der von R. Bayer 1972 entwickelten B-Bäume.
- RB-Bäume mit n Knoten haben eine Höhe von O(log n). Die Balancierung nach Einfügen benötigt ebenfalls O(log n) Zeitschritte, das Entfernen eines Knotens braucht ebenso O(log n).

Aufbau balancierter Rot-Schwarz-gefärbte Bäume II

Beispiele:



resultiert aus Eingabe `7;5;9;11;3`

resultiert aus Eingabe `3;11;9;5;7`

http://yallara.cs.rmit.edu.au/~tgunning/CS544/RBTree-Applet.html

Aufbau balancierter Rot-Schwarz-gefärbte Bäume III

```
type color = Red | Black
type 'a btree = Node of color * 'a btree * 'a * 'a btree | Leaf

let empty = Leaf;;

let balance = function
  Black, Node (Red, Node (Red, a, x, b), y, c), z, d ->
        Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
| Black, Node (Red, a, x, Node (Red, b, y, c)), z, d ->
        Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
| Black, a, x, Node (Red, Node (Red, b, y, c), z, d) ->
        Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
| Black, a, x, Node (Red, b, y, Node (Red, c, z, d)) ->
        Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
| a, b, c, d -> Node (a, b, c, d)
```

Zum Vergleich:
http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/freenet/whiterose/rbtree.h?rev=1.3&content-type=text/vnd.viewcvs-markup

OCaml-Programm für Rot-Schwarz-gefärbte Bäume IV

```
let insert_rb x s = let rec ins = function
   Leaf -> Node (Red, Leaf, x, Leaf)
 | Node (color, a, y, b) as s ->
        if x < y then balance (color, ins a, y, b)
                 else if x > y then balance (color, a, y, ins b)
                               else s in match ins s with
                                Node (_, a, y, b) -> Node (Black, a, y, b)
                               | Leaf -> raise (Invalid "insert");;

let rec set_of_list_rb = function
                          [] -> empty
                        | x :: l -> insert_rb x (set_of_list_rb l)

let l = [7;5;9;11;3];;

# set_of_list_rb l;;
- : int btree =
Node (Black,
 Node (Red, Node (Black, Leaf, 3, Leaf), 5, Node (Black, Leaf, 7, Leaf)),
9, Node (Black, Leaf, 11, Leaf))
```

OCaml-Programm für Rot-Schwarz-gefärbte Bäume V

- *Übung*: Extrahieren Sie die *Invarianten* für diesen Baumtyp aus dem vorstehenden Programm!

Ausnahmen und ihre Behandlung in OCaml

- Normalerweise sollten Algorithmen so entworfen werden, daß sie für alle möglichen Eingaben korrekte Ausgaben produzieren.
- Dennoch existieren oft Ausnahmesituationen, welche z.B. dann eintreten, wenn eine Funktion vom Benutzer außerhalb des spezifizierten Definitionsbereichs verwendet wird.
- Typische Beispiele: Division durch 0, Speicherzugriff außerhalb zulässiger Grenzen, Ende einer Datei beim Lesen überschritten
- Zur Information des Benutzers bzw. zur *Behandlung* des Fehlers existiert in OCaml das Konzept der Ausnahmebehandlung (Exception-Handling)
- Sehr flexible Behandlung möglich: unterschiedliche Reaktion des Programms durch Definition eigener Behandler und Möglichkeit zur Definition eigener Ausnahmen
- Programmtechnische Vorteile insbesondere die *Separierung des Behandlers vom „regulären“ Code*, Bildung von Fehlerklassen
- Hinweis: Ausnahmen führen zu einer *Änderung des vorgesehenen Programmablaufs*. Im Unterschied zu *Unterbrechungen* (interrupts) treten sie aber nur an bekannten Punkten im Programm auf! Man spricht deshalb auch von synchronen vs. asynchronen Unterbrechungen

Ausnahmen und ihre Behandlung in OCaml

- Grundsätzliche Metapher: Bei Auftritt einer Ausnahmesituation „**wirft**“ eine Funktion eine Ausnahme, die von einem Ausnehmebehandler „**eingefangen**“ wird (*catch-and-throw*-Mechanismus).
- Um eine Ausnahme in OCaml zu werfen wurde bereits die OCaml-Standard-Funktion *failwith* benutzt.
- Allgemeiner werden Ausnahmen durch die Funktion *raise* geworfen, die als Argument ein Objekt des Typs *exn* erwarten:

```
# raise;;
- : exn -> 'a = <fun>
```

- Zur Deklaration eigener Ausnahmen steht in OCaml das Schlüsselwort **`exception`** zur Verfügung, die Syntax ist wie bei Typkonstruktoren (Ausnahme-Namen sind Konstruktoren):

```
# exception Ausnahme1 of string;;
exception Ausnahme1 of string
```

- Anwendung:

```
# raise (Ausnahme1 "Ich bin eine Ausnahme");;
Exception: Ausnahme1 "Ich bin eine Ausnahme".
```

Ausnahmen und ihre Behandlung in OCaml

- Zum Einfangen von Ausnahmen dienen Ausnahmebehandler mit folgender Syntax:

```
try expr with
| pattern_1    -> expr_1
| ...             ...
| pattern_n    -> expr_2
```

- Beispiel:

```
# exception EmptyList;;
# let head v = match v with
        | [] -> raise EmptyList
        | hd::tl -> hd;;
val head : 'a list -> 'a = <fun>

# head [1;2;3];;
- : int = 1
# head [];;
Exception: EmptyList.
```

Ausnahmen und ihre Behandlung in OCaml

- Nunmehr Einführung eines Behandlers in das Beispiel:

```
# let head2 g =
        try head g with EmptyList -> "Liste ist voellig leer";;
val head2 : string list -> string = <fun>

# head2 ["hallo";"studenten"];;
- : string = "hallo"

# head2 [];;
- : string = "Liste ist voellig leer"
```

- Zu beachten: Während head auf *'a list* arbeitet, ist head2 nur noch auf *string list* definiert, denn Ausnahmen sind *monomorph*

- Behandler-Stellung in den Programmen typischerweise (wie in Java): es können beliebig viele Funktionen eine bestimmte Ausnahme erzeugen, der/die Behandler befinden sich bei der aufrufenden Funktion (die ihrerseits aufgerufen werden kann von einer *umgebenden* Funktion, die weitere Behandler zur Verfügung stellt).

Ausnahmen und ihre Behandlung in OCaml

- Ausnahmen können auch dazu benutzt werden, die Berechnung von Ausdrücken zu steuern; in Abhängigkeit vom Funktionsargument kann eine Berechnung anders verlaufen.
- Damit wird der vollständig funktionale Bereich verlassen, denn die Auswertungsreihenfolge spielt jetzt u.U. doch eine Rolle!

```
# exception Found_one;;
# exception Found_two;;
# let rec multrec l = match l with
              [] -> 1
            | 1 :: _  -> raise Found_one
            | 2 :: _  -> raise Found_two
            | n :: x  -> n * (multrec x);;

# let multrec_with_handler l =
      try multrec l with
              Found_one -> 4711
            | Found_two -> 4712;;
```

Imperativer Programmierstil: Konzept des Zustands

- Einführung des Begriffs des *Zustands*, der im Speicher des Rechners gehalten wird und der sich gezielt durch *Sequenzen* von *Instruktionen* ändern läßt (durch *Zuweisungen* von Werten an Speicherstellen).
- In OCaml gibt es zwei Bereiche, die sich mit imperativen Sprachmitteln bearbeiten lassen:
  - Selektive Änderung einer Komponente bei strukturierten Datentypen: arrays und records, spart erheblichen Kopieraufwand
  - Ein-/Ausgabe, ist ohnehin normalerweise mit Zustandsänderungen verbunden
- Imperative Sprachkonstrukte sind (auch allgemein) Sequenz (Instruktionsblock), Iteration (Schleife) und Selektion (Fallunterscheidung/Alternativen).

Imperativer Programmierstil: Änderbare Datentypen I

- **Vektoren** ((schachtelbare) eindimensionale Felder). Syntax: **[|**_,_, ..., _**|]**.
  Initialisierung über direkte Zuweisung:

```
# let v = [|3.14; 6.28; 9.42|];;
val v : float array = [|3.14; 6.28; 9.42|]
```

- Oder mit Initialierungsfunktion mit Vorbesetzungsmöglichkeit:

```
# let v = Array.create 5 3.14;;
val v : float array = [|3.14; 3.14; 3.14; 3.14; 3.14|]
# let u = Array.create 3 v;;
val u : float array array=[| [|3.14;3.14;3.14;3.14;3.14|];
                             [|3.14;3.14;3.14;3.14;3.14|];
                             [|3.14;3.14;3.14;3.14;3.14|]|]
```

  – `create` wird importiert aus dem Modul `Array` (Standard-lib)

- Zugriff bzw. Zuweisung über Punktnotation:

```
# v.(1);;                  - : float = 3.14
# v.(2) <- 100.0;;
- : float array = [|3.14; 3.14; 100.; 3.14; 3.14|];;
# u.(1);;
- : float array = [|3.14; 3.14; 3.14; 3.14; 3.14|]
# (u.(1)).(1);;
- : float = 3.14
```

Imperativer Programmierstil: Änderbare Datentypen II

- **Records** mit änderbaren Komponenten

```
type name = {...; mutable name : t; ...}
```

```
# type student = {matrikel : int; mutable nachname :
                  string; vorname : string};;
# let stud1 : student = {matrikel = 5443454;
      nachname ="Musterfrau"; vorname = "Angela"};;
val stud1 : student = {matrikel = 5443454; nachname =
"Musterfrau"; vorname = "Angela"}
# stud1.nachname <- "Merkl";;        - : unit = ()
# stud1;;
- : student = {matrikel = 5443454; nachname =
"Merkl"; vorname = "Angela"}
# stud1.matrikel <- 123456;;
Characters 0-24:
stud1.matrikel <- 123456;;
^^^^^^^^^^^^^^^^^^^^^^^^
The record field label matrikel is not mutable
```

Imperativer Programmierstil: Kontrollstrukturen I

- Bereits bekannt: **Selektion** oder Fallunterscheidung,
  `if ... then ... else` **(oder** `case`**-Ausdruck bzw. Pattern-Matching)**
- Zusätzlich: **Sequenz,** d.h. Abfolge von Instruktionen
  `expr1; expr2; ... ; exprn` **bzw.**
  `begin expr1; expr2; ... ; exprn end`

```
# let rec hilo n = print_string "Status:";
           let i = (12 * 2) / 3 in
           if i = n then print_string "End Iteration\n\n"
                    else if i > n then
                    begin
                      print_string "INC n\n";
                      hilo (n + 1)
                    end;
           if i < n then print_string "Ueberschritten";;

# hilo 6;;
  Status:INC n
  Status:INC n
  Status:End Iteration
```

Imperativer Programmierstil: Kontrollstrukturen II

- **Und**: **Iteration** oder FOR-Schleife,

```
for name = expr1 to expr2 do expr3 done
for name = expr1 downto expr2 do expr3 done
```

- **Beispiel:**

```
# for i=1 to 10
    do
      print_int i;
      print_string " "
    done;
    print_newline();;

1 2 3 4 5 6 7 8 9 10
- : unit = ()
```

- Im Rumpf der Schleife darf kein anderer Wert als unit berechnet werden!

- **Und**: **Iteration** oder WHILE-Schleife (abweisende Schleife),
  `while` *`expr1`* `do` *`expr2`* `done`

- **Beispiel:**

```
# let r = ref 1 in
    while !r < 11 do
      print_int !r ;
      print_string "|";
      r := !r+1
    done;;

1|2|3|4|5|6|7|8|9|10|- : unit = ()
```

Imperativer Programmierstil: Ein-/Ausgabe I

- Die Standard-I/O-Bibliothek von OCaml ist sehr leistungsfähig, flexibel und umfangreich.
- Es gibt zwei Typen von Kommunikationskanälen: **`in_channel`** und **`out_channel`**.
- Vor Benutzung muß Kanal unter Angabe eines Namens geöffnet werden:

```
# open_out "stdout";; (* implizit bei startup des interpreters *)
# let meinkanal = open_in "C:/.emacs";;
```

- Danach Schreiben auf einen Ausgangskanal mit `output_string, output_char output` oder `output_value (* unsafe *)`:

```
# output_string stdout "hallo";;
hallo- : unit = ()
# output stdout "hallo" 3 2;;
lo- : unit = ()
```

- ... bzw. Lesen von einem Eingangskanal mit `input_line, input_char, input` oder `input_value`:

```
# input_line meinkanal;;
- : string = ";;;;;;;;;;;;;;;;;;;;;;;;; -*- Mode: Emacs-Lisp -*-;;;;;;;;;;;;;;;;;;"
# input_line meinkanal;;
- : string = ";; .emacs --- A simple startup file, enabling Auctex (and ispell.info)"
```

Imperativer Programmierstil: Ein-/Ausgabe II

- Mit Kanal-Funktionen wie **`seek_out`**, **`pos_out`** oder **`out_channel_length`** ist die Manipulation von Dateien einfach. Beachte: Physikalisch geschrieben wird erst beim Schließen eines Kanals oder mit Hilfe der Funktion **`flush out_channel`**.
- Speziell wichtig ist die Bibliothek Printf und die dort definierte Funktion **`fprintf`**.

```
# Printf.fprintf;;
- : out_channel -> ('a, out_channel, unit) format -> 'a = <fun>

# Printf.fprintf stdout "Artikel: %s, Nummer: %d SOWIE Preis: %f \n" "papier" 233 12.34;;
Artikel: papier, Nummer: 233 SOWIE Preis: 12.340000
- : unit = ()

# let preisdruck = Printf.fprintf stdout "Artikel: %s, Nummer: %d SOWIE Preis:
%f \n";;
Artikel: val preisdruck : string -> int -> float -> unit = <fun>
# preisdruck  "papier" 233 12.34;;
papier, Nummer: 233 SOWIE Preis: 12.340000
- : unit = ()
```

Im letzten Beispiel analysiert OCaml den Formatstring und kann damit die übergebenen Parameter auf Korrektheit prüfen

Imperativer Programmierstil: Ein-/Ausgabe III

- Beispielhafte Argumente für den Formatstring:
  - **`d`** oder **`i`** konvertieren integer to signed decimal
  - **`s`** fügt String-Argument ein
  - **`f`** konvertiert **`float`** in Dezimalnotation **`ddd.ddd`**
  - **`E`** oder **`e`** konvertieren float in „scientific notation“: **`m.mmmE±eee`**
  - **`g`** konvertiert **`float`** nach **`f`** oder **`e`**, je nachdem, was kompakter ist
  - **`b`** konvertiert Boolsche Argumente in die Strings **`true`** oder **`false`**
  - ...
- Schließlich gibt es auch vordefinierte Funktionen (einfach anzuwenden, weniger flexibel in der Ausgabe): **`print_int, print_char, print_string, print_float`**

Imperativer Programmierstil: Ein-/Ausgabe IV

- Grafik (mit Maus-Ereignissteuerung)

```
(* Initialisierung *)
#load "graphics.cma";;
Graphics.open_graph "";;
Graphics.set_window_title "Mein Graphik-Fenster";;

(* Einfache Operationen *)
Graphics.draw_rect 100 200 300 400;;
Graphics.draw_poly_line [|(0,0);(90,80);(70,60);(50,40)|];;
```

Imperativer Programmierstil: Ein-/Ausgabe V

- Grafik (mit Maus-Ereignissteuerung)

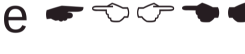
```
(* Komplexere Funktionen *)
let net_points (x,y) l n =
let a = 2. *. pi /. (float n) in
let rec aux (xa,ya) i =
    if i > n then []
    else
    let na = (float i) *. a in
    let x1 = xa + (int_of_float ( cos(na) *. l))
    and y1 = ya + (int_of_float ( sin(na) *. l)) in
    let np = (x1,y1) in
    np :: (aux np (i+1))
in Array.of_list (aux (x,y) 1);;
```

Zum Begriff der Information

Zum Begriff der Information

- Informatik ist die Wissenschaft **der systematischen Verarbeitung von Information**
- Zunächst zu klärende Begriffe: Signal, Nachricht, Repräsentation, Information.
  - **Signal**. Elementar feststellbare Veränderung (Lichtblitz, Tonhöhe, ...) eines physikalischen Parameters (Signalparameter)
  - **Datum (i)**. Signal, dargestellt durch digitale Zeichen (siehe DIN **ISO/IEC 2382** Text und Daten)
  - **Nachricht**. Folge von Signalen bestimmter (physikalischer) **Repräsentation** einschließlich zeit-räumlicher Anordnung
  - **Information**. Nachricht, die einen Sachverhalt ausdrückt, einem Ziel dient, Wissen erweitert oder: *einer vom Empfänger der Nachricht zugeordnete Bedeutung*
    - Nachricht wird durch eine **Interpretationsvorschrift** zur Information
    - Es gibt keine (abstrakte) Information ohne (reale) Nachrichten/Repräsentation
    - Information bedarf der Repräsentation, sonst kann sie nicht gespeichert oder verarbeitet werden
    - Dieselbe Information kann man durch verschiedene Nachrichten darstellen
    - Dieselbe Nachricht kann verschiedene Informationen vermitteln

Zum Begriff der Information

- **Codierung**. Umwandlung der Repräsentation einer Nachricht.
  - Beispiel: Strichzeichen II oder Wort „zwei“ steht für Information: Zahl 2.
- Informationsstruktur: (***I:*** *Repr* → *Info*); die Interpretationsvorschrift ***I*** ordnet jeder Repräsentation *Repr* eine Information *Inf* zu. Das Paar (*I*, *Info*) wird als **Semantik** von *Repr* bezeichnet. Interpretationsvorschriften sind ihrerseits Nachrichten!
  - Beispiel: Geometrische Figuren 𝒱𝒥𝒥 werden durch Interpretationsvorschrift ***I*** (**"**𝒱𝒥𝒥**"**) „römische Zahl“ zur Zahl 7. Zeichenfolge  wird durch Interpretationsvorschrift Zeichensatzwechsel zur Folge ABCDEFGH.
- Demgegenüber ist die **Syntax** einer Information die Vorschrift, wie die einzelnen Bausteine zusammengesetzt werden dürfen.
- Die **Pragmatik** beschreibt, welchen Zweck die Information hat bzw. welche Handlungen sich aus der Nachricht ergeben (sollen).
- **Semiotik** ist Syntax + Semantik + Pragmatik.
- **Datum (ii).** Eine konkrete Information zusammen mit ihrer Repräsentation.

Zum Begriff der Information

- **Datenverarbeitung**: **Bedeutungstreue** Manipulation oder **Umcodierung** von Repräsentationen, d.h. Manipulation muß in Übereinstimmung mit der Bedeutung vorgenommen werden.
  - Beispiel 1: Nachricht „**`hallo du`**“. Interpretation als Menge von Zeichen. Unter dieser Interpretationsvorschrift trägt die Verarbeitung zu einer lexikograpisch geordnete Zeichenfolge „ **`adhllou`**“ die gleiche Information. Unter der Interpretationsvorschrift des Deutschen ist eine solche Verarbeitung nicht bedeutungstreu.
  - Beispiel 2: Die bedeutungstreue Umcodierung von Temperaturangaben. $\boldsymbol{C}(f) = 5/9*(f-32)$. *f* ist die nach der Meßvorschrift „Fahrenheit“ gemessene Temperatur, $\boldsymbol{C}$(f) die Umcodierung auf die Temperaturskala „Celsius“. 100 ^{o}C entspricht 212 F und beide tragen die Information „kochendes Wasser“.
- **Problem**: Wie kann man die Interpretationsvorschriften formalisieren, wie kann man eindeutig beschreiben, was sie tun (etwa im Kopf des Empfängers?) Bedenke: die Interpretation ***I*** kommt letztlich dadurch zustande, daß sich alle Mitglieder einer Gesellschaft darauf geeinigt haben. Einfaches Beispiel: Rotes Licht (Ampel) bedeutet für Westeuropäer: Straßenrichtung gesperrt

Zum Begriff der Information

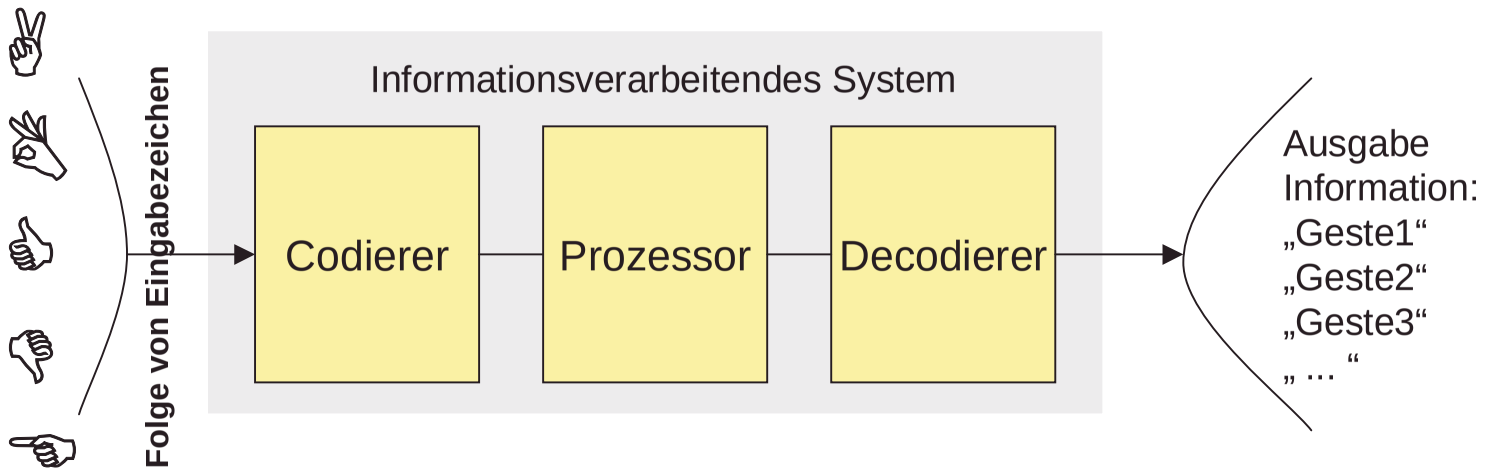
- In der Informatik hauptsächlich interessant: Semantik von Programmiersprachen bzw. den mit ihnen zu formenden Ausdrücken.
- Hier kann man nicht der Interpretation jedes Ausdrucks eine langwierige Diskussion im sozialen Umfeld vorangehen lassen. Außerdem muß man sich darauf verlassen können, daß alle unter einem Ausdruck dasselbe verstehen, also dieselbe Wirkung erwarten. Deshalb:
- **Formale** Semantikdefinition einer Informationsstruktur (***I****: Repr* $\rightarrow$ *Info*) wird vorgenommen durch Rückführung auf eine als bekannt vorausgesetzte Struktur (***I***$_0$*: Repr*$_0$ $\rightarrow$ *Inf*).
  - Beispiel: Strichzeichen **"IIIIIIIIIIIIIII"** wird als Zahl 15 interpretiert und (zumindest innerhalb einer Wissensdomäne) als allgemeinverbindlich angesehen. Für die Betrachtung komplexerer Systeme genügt es dann, die entsprechende Umcodierung angeben zu können: ***C***("01111") = **"IIIIIIIIIIIIIII"**

Zum Begriff der Information

- Es gibt vier wichtige Ansätze zur Beschreibung der Semantik von Programmiersprachen:
- **Übersetzersemantik**: Bedeutung von Konstrukten einer Programmiersprache **N** wird auf die Bedeutung der Konstrukte einer bekannten (einfacheren) Programmiersprache $\mathbf{N}_0$ zurückgeführt. Dazu gibt man einen Übersetzer $\mathbf{N} \rightarrow \mathbf{N}_0$ an. Problem wird allerdings in gewisser Weise nur verlagert (Semantik von $\mathbf{N}_0$ muß nach wie vor definiert werden).
- **Operationale Semantik**: Angabe eines Interpreters für **N**, der aus Eingabedaten durch schrittweise Abarbeitung von Programmen Ausgabedaten erzeugt. Interpreter wird als Automat angegeben (siehe dazu später).
- **Denotationale Semantik**: Beschreibung der Wirkung, die Anweisungen auf Zustände (also Variablenbelegungen) haben. Sei A_0 eine Anweisung, z ein Zustand aus der Menge aller möglichen Zustände Z und z' der auf z nach Ausführung der Anweisung folgende Zustand. Dann ist die Wirkung eine Abbildung $Z \rightarrow Z$ der Form: $\mathbf{F}\ [A_0]: z \rightarrow z'$ (semantische Funktion). Beispiel: gegeben zwei Variablen m und n mit $m = 5$ und $n = 6$. Dann ist die Wirkung der Anweisung `m := n + 1` wie folgt: $\mathbf{F}$ [`m:= n + 1`] (5,6) = (7,6).
- **Axiomatische Semantik**: Angegeben werden die Eigenschaften von Zuständen vor und nach der Ausführung einer Aktion als Prädikate („Zusicherungen“). {**P**} A_0 {**Q**}. **P** ist die Vorbedingung, **Q** die Nachbedingung (siehe später).

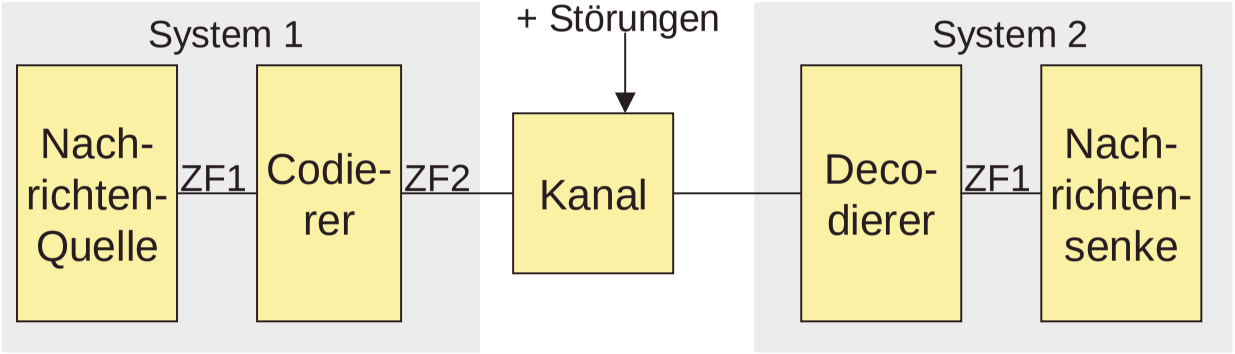
Grundlagen Codierungstheorie

- Bei der Codierung für technische Systeme ist man bestrebt, einfache, kompakte und ggf. auch störsichere Repräsentationen für Nachrichten zu finden
  - Kurze Codewörter
  - Redundanz (Weitschweifigkeit), um Fehler erkennen (oder korrigieren) zu können
- Zwei Klassen von Codierungen: (**a**) für interne Repräsentation und (**b**) für Informationsübertragung
- Fall **a**)



- Zeichenfolge → Zeichenfolge, die sich für interne Repräsentation gut eignet
- Prozessor arbeitet auf dieser Repr, Dekodierer wandelt Ergebnis in geeignete Form

- Fall **b**)



- Zeichenfolge $\rightarrow$ Zeichenfolge, die sich für Übertragung gut eignet
- Parallelübertragung: ein oder mehrere Zeichen werden parallel übertragen vs. serielle Übertragung: Zeichen werden in Untereinheiten zerlegt, die nacheinander übertragen werden

- Definitionen:
  - Zeichenmenge A: Zeichenvorrat, bei linearer Ordnung heißt A auch Alphabet
  - Elementarer Zeichnvorrat: **B** = {L, O}, zwei Binärzeichen oder **bi**nary digi**t**
  - Menge A* der Zeichenfolgen über Zeichenvorrat A enthält Wörter über A
  - Menge **B*** = {L,O}* ist Menge der Binärwörter, z.B. „LOLOLL“ ist 6-bit-Binärwort
  - Menge $\mathbf{B}^n = \{L,O\}^n$ ist Menge der n-bit Binärwörter
- Ein Code ist damit die Abbildungsvorschrift
  - c: A $\rightarrow$ B für einzelne Zeichen, oder
  - c: A* $\rightarrow$ B* für Abbildung Zeichenfolge $\rightarrow$ Zeichenfolge
- In der Rechentechnik zumeist Binärcodierungen für Alphabete, also Codierungen der Form: c: A $\rightarrow$ B*
- Beispiele für Alphabete: alle Großbuchstaben, alle Ziffern, alle chinesischen Schriftzeichen

- Beispiele für Codes: ASCII (American Standard Code for Information Interchange (7 bzw. 8 bit)), ISO-Latin1 (8bit), Unicode (16 bit)

7-bit-ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	00 0000 0000	01 0000 0001	02 0000 0010	03 0000 0011	04 0000 0100	05 0000 0101	06 0000 0110	07 0000 0111	08 0000 1000	09 0000 1001	10 0000 1010	11 0000 1011	12 0000 1100	13 0000 1101	14 0000 1110	15 0000 1111	
	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	
0	□	⌜	⊥	⌟	↘	⊠	✓	␣	↶	→	≡	↓	⇊	←	⊗	⊙	8
	16 0001 0000	17 0001 0001	18 0001 0010	19 0001 0011	20 0001 0100	21 0001 0101	22 0001 0110	23 0001 0111	24 0001 1000	25 0001 1001	26 0001 1010	27 0001 1011	28 0001 1100	29 0001 1101	30 0001 1110	31 0001 1111	
	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	
1	⊟	◷	◶	◵	◴	✓	⎍	⊣	⧖	↓	⎩	⊖	◰	◱	◳	◲	9
	32 0010 0000	33 0010 0001	34 0010 0010	35 0010 0011	36 0010 0100	37 0010 0101	38 0010 0110	39 0010 0111	40 0010 1000	41 0010 1001	42 0010 1010	43 0010 1011	44 0010 1100	45 0010 1101	46 0010 1110	47 0010 1111	
2	SP	!	"	#	$	%	&	′	(	)	*	+	,	−	.	/	A
	48 0011 0000	49 0011 0001	50 0011 0010	51 0011 0011	52 0011 0100	53 0011 0101	54 0011 0110	55 0011 0111	56 0011 1000	57 0011 1001	58 0011 1010	59 0011 1011	60 0011 1100	61 0011 1101	62 0011 1110	63 0011 1111	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	B
	64 0100 0000	65 0100 0001	66 0100 0010	67 0100 0011	68 0100 0100	69 0100 0101	70 0100 0110	71 0100 0111	72 0100 1000	73 0100 1001	74 0100 1010	75 0100 1011	76 0100 1100	77 0100 1101	78 0100 1110	79 0100 1111	
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	C
	80 0101 0000	81 0101 0001	82 0101 0010	83 0101 0011	84 0101 0100	85 0101 0101	86 0101 0110	87 0101 0111	88 0101 1000	89 0101 1001	90 0101 1010	91 0101 1011	92 0101 1100	93 0101 1101	94 0101 1110	95 0101 1111	
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	D
	96 0110 0000	97 0110 0001	98 0110 0010	99 0110 0011	100 0110 0100	101 0110 0101	102 0110 0110	103 0110 0111	104 0110 1000	105 0110 1001	106 0110 1010	107 0110 1011	108 0110 1100	109 0110 1101	110 0110 1110	111 0110 1111	
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	E
	112 0111 0000	113 0111 0001	114 0111 0010	115 0111 0011	116 0111 0100	117 0111 0101	118 0111 0110	119 0111 0111	120 0111 1000	121 0111 1001	122 0111 1010	123 0111 1011	124 0111 1100	125 0111 1101	126 0111 1110	127 0111 1111	
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL	F

Codierungstheorie: Codes fester Länge

- Typischerweise für rechnerinterne Repräsentationen von Zahlen.
- **Direkte** Codes: feste Gewichte g_i für jede Binärstelle. Mit den Multiplikator-funktionen $w(L) = 1$ und $w(O) = 0$ wird jedem Binärwort $d = <d_n\ d_{n-1}\ \ldots\ d_0>$ mit $d_i \in \{L,0\}$ eine Zahl zugeordnet durch

$$z = \sum_{i=0}^{n} g_i w(d_i)$$

- Beispiel: Direkter Code für 4-bit-Binärwörter zur Codierung von Dezimal- bzw. Hexadezimalzahlen mit den Gewichten: $g_i = 2^i$, $n = 3$

g_i =	8	4	2	1
	i=3	i=2	i=1	i=0

- Damit Zuordnung zu den ersten zehn Ziffern des Dezimalsystems c_D: $\{0...9\} \rightarrow B^4$

a	c_D(a)
0	0000
1	000L
2	00L0
3	00LL
4	0L00
5	0L0L
6	0LL0
7	0LLL
8	L000
9	L00L
A	L0L0
B	L0LL
C	LL00
D	LL0L
E	LLL0
F	LLLL