Einführung in die Informatik 1
Wintersemester 2005/2006

Prof. Dr. Alois Knoll

TU München

Lehrstuhl VI Robotics and Embedded Systems

Informatik 1: Organisation

Bestandteile der Vorlesung

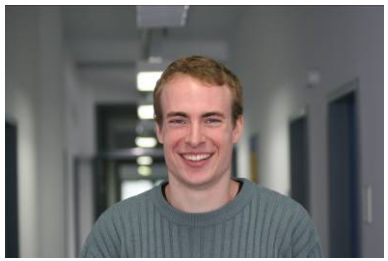
- Vorlesung
  - Donnerstag 10:15-11:45 MW 0001
  - Freitag 10:15-11:45 MW 0001
  - 6 ECTS Punkte
  - am 27.10.05, 03.11.05 und 01.12.05 (Dies Academicus) entfällt die Vorlesung
- Praktikum
  - Eigenständige Veranstaltung (aber eng mit der Vorlesung verbunden)
  - 6 ECTS Punkte
- Semestralklausur
  - 11.02.2005 13:00 Uhr
  - Voraussetzung für Schein
  - DVP-Klausur für Diplominformatiker

Team

Prof. Dr. Alois Knoll

Dr. Markus Schneider

Praktikumsleitung

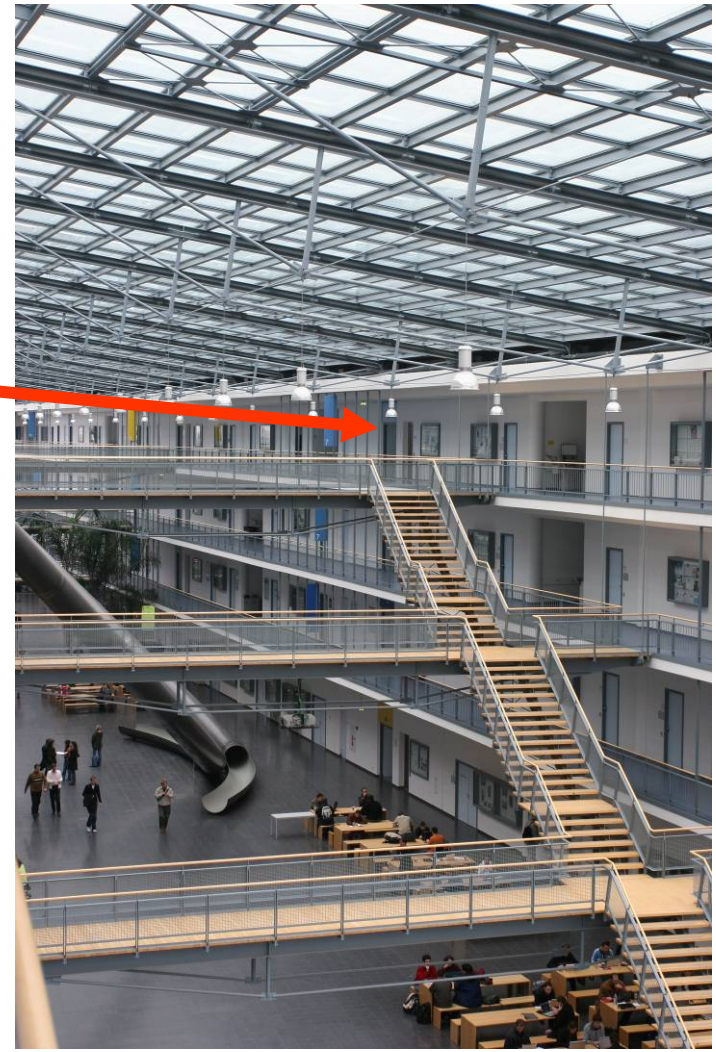


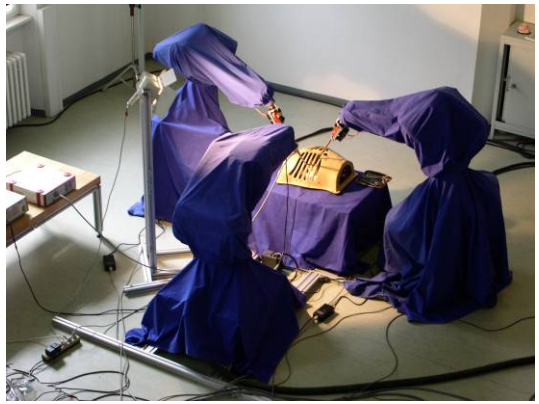
Dipl.-Inf. Christian Buckl

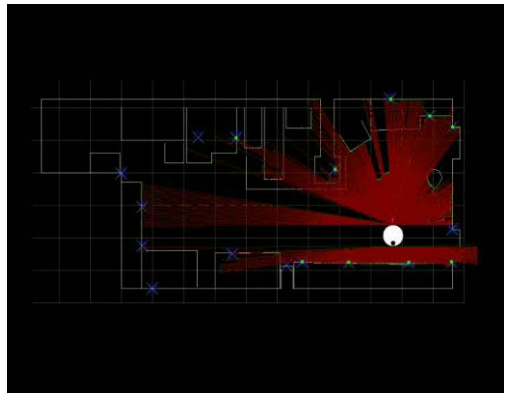
Technik,
Praktikumsleitung

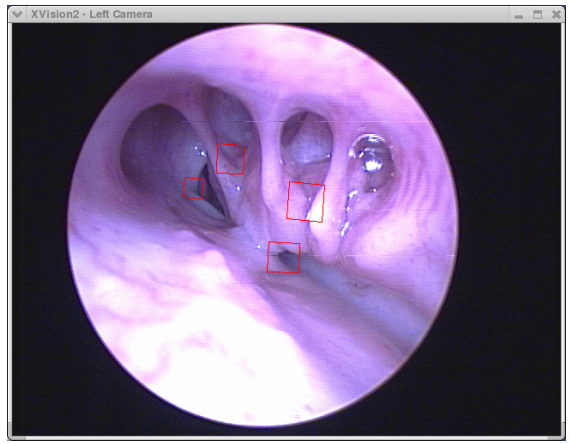
Lehrstuhl

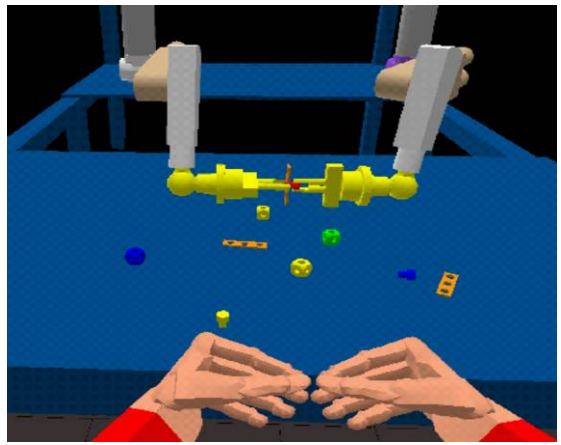
- Lehrstuhl VI: Eingebettete Systeme und Robotik
- Professoren: Prof. Knoll, Prof. Burschka, Prof. Schmidhuber, Prof. Hirzinger (DLR Robotik)
- 3.Stock, Gang 7
- Themengebiete:
  - Robotik
    - Mobile Roboter
    - Industrieroboter
  - Zuverlässige Echtzeitsysteme
  - Maschinelles Lernen
  - Interaktion Mensch-Maschine
  - Neuronale Netze
  - Medizinische Informatik











→Tag der offenen Tür 2005, Samstag, 22.10.2005, Garching

Praktikum I

- 4-stündige Veranstaltung zur praktischen Umsetzung der theoretischen Inhalte dieser Veranstaltung
- Die Veranstaltung ist unterteilt in eine Übung, sowie die praktische Umsetzung am Rechner
- Termine

Block	Tag	Uhrzeit	Gruppen	Spezialgruppen
A:	Montag	09:00-12:00	4	2 Gruppen Wirtschaftsinfo
B:	Montag	14:00-17:00	3	
C:	Dienstag	14:00-17:00	3	
D:	Mittwoch	09:00-12:00	2	
E:	Donnerstag	14:00-17:00	4	
F:	Freitag	13:00-16:00	4	2 Gruppen Wirtschaftsinfo

Praktikum II

- Erster Termin in der Woche vom 31.10.05 ... 4.11.05
- Zur Teilnahme ist eine Anmeldung erforderlich:
  - ab Freitag 28.10.05 11:00 Uhr möglich
  - Einteilung nach dem First-Come-First-Serve Prinzip
  - zur Anmeldung wird ein Zertifikat der RBG benötigt
  - Bitte bei der Meldung angeben, ob ein Notebook vorhanden ist
  - Falls sie ein Notebook besitzen, melden Sie sich bitte in einer Gruppe mit hoher Notebookrate an
- Alles Weitere zum Praktikum auf dem Merkblatt Nr.1
- Alles Weitere zur Praktikumsanmeldung auf Merkblatt Nr.2

Rechnerarbeitsplätze

In der Rechnerhalle befinden sich ca. 200 Rechner vom Typ Sun, Betriebssystem Solaris

Weitere 100 Rechner befinden sich in der Bibliothek! ... In wesentlicher schönerer Arbeitsatmosphäre ...

Kennungen für die Informatikhalle

- Studenten des Studiengangs Informatik-Bachelor haben die Kennungen bereits schriftlich erhalten bzw. erhalten ihre Kennung im Raum 00.010.013
- Studenten mit Nebenfach Informatik können die Kennung nach der Vorlesung in der Rechnerhalle beantragen:
  - Anmeldung an einem beliebigen Rechner mit dem Login „info1“ und einem leeren Passwort
  - Füllen Sie das Formular mit ihren persönlichen Daten aus.
  - Die Kennungen können Sie ab dem 02.11.05 von 10:00-13:00 Uhr im Raum 00.10.013 abholen (aus technischen Gründen vorher nicht möglich!)

Weitere Informationen

- Homepage der Vorlesung:
  - http://atknoll1.informatik.tu-muenchen.de:8080/tum6/lectures/courses/ws0506/info1
- Homepage des Praktikums
  - http://atknoll1.informatik.tu-muenchen.de:8080/tum6/lectures/courses/ws0506/infop1

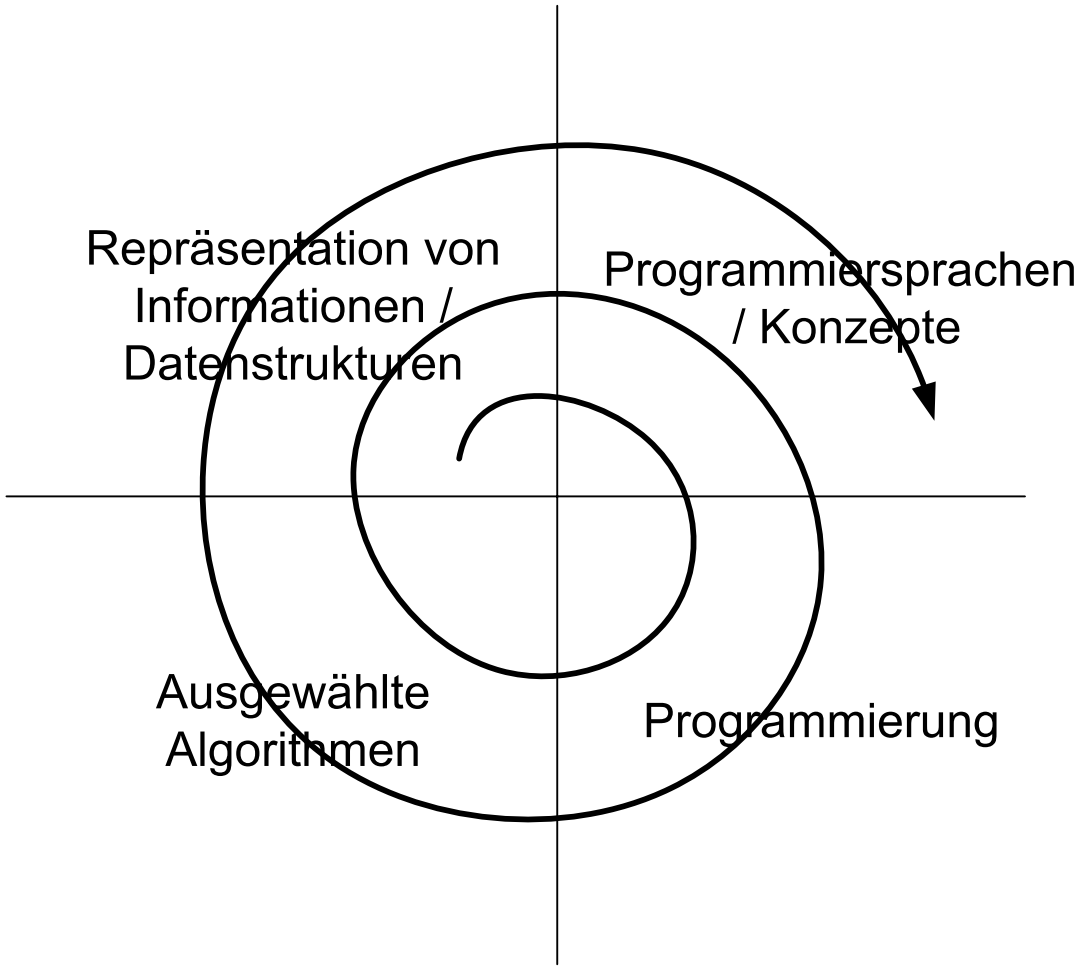
- Beim jeweiligen Praktikumsbetreuer
- Praktikumsleitung
  - Email: info1@in.tum.de
- Newsgroup:
  - tum.info.info12

Informatik 1: Vorlesungsstruktur und -inhalt

Ziel der Veranstaltung

- Beherrschung der Aufgaben eines Informatikers:
  - Modellierung von Problemen aus der Wirklichkeit
  - Konstruktion von Lösungen mit Hilfe der Informatik
  - Systematisches Entwickeln einer Lösung in Form eines Informatik-Systems mit Hilfe von Programmiersprachen
- Beherrschung der theoretischen Grundlagen
  - Algorithmus, Algebra, Termersetzungssysteme, Verifikation, Validation von Programmen, Verständnis der Effizienz eines Programms, ...
- Aktive Beherrschung von verschiedenen Programmierparadigmen
  - Funktionale, imperative und objektorientierte Programmierung

Didaktischer Ansatz der Vorlesung



Vorlesungsstruktur und -inhalt I

- Allgemeine Vorlesungsmodalitäten
  - Ablauf der Vorlesung, Praktikum, Anmeldung zum Praktikum, Wesentliche Werkzeuge: OCAML, Acrobat, StarOffice, Emacs
- „Info 1 in a nutshell“
  - Definition: Algorithmus, Datenflussdiagram, Algebra/Rechenstruktur und Signaturen
  - Einfache Rechenstrukturen: Natürliche Zahlen, Gleitpunktzahlen, Sequenzen, Bäume, Arrays
  - Ocaml/F# auf den verschiedenen Plattformen
  - Funktionen
  - einfache Rekursion über natürlichen Zahlen
  - Typdeklarationen
  - Rekursion auf selbstdeklarierten Typen

Vorlesungsstruktur und -inhalt II

- Zentrale Datenstrukturen
  - Listen, Keller, Warteschlangen, Bäume
- Information und Codierung
  - Unterschiedliche Annäherung an den Begriff
    - Repräsentation von Information
    - Interpretation von Repräsentation
  - Shannonsche Informationstheorie
    - Grundlagen: Codierung, Entropie
    - Codierungsverfahren: Huffman, LZW
  - Kryptologie
    - Historie, Konzepte
    - Zahlentheoretische Techniken, z.B. RSA-Algorithmus
    - Anwendungen: ssh, pgp

Vorlesungsstruktur und -inhalt III

- Applikative/funktionale Programmierung
  - Definition, Eigenschaften
    - Rekursive Sorten
    - Syntaktische Beschreibung
    - Fixpunktdeutung
  - Rekursive Funktionen
    - Syntaktische Beschreibung
    - Kategorisierung: Linear, Nichtlinear
    - Semantik mit Fixpunktdeutung
    - Terminierung, Korrektheit, Spezifikation
- Grundlegende Algorithmen
  - Sortierverfahren, Mustersuche, Orthographische Ähnlichkeit
  - Algorithmen auf rekursiven Datenstrukturen

Vorlesungsstruktur und -inhalt IV

- Zuweisungsorientierte Programmierung
  - Sprachkonstrukte: Bedingte Anweisung, Iteration, Sequenz
  - Datenstrukturen: Felder, Records
  - Referenzen, Zeiger
- Objektorientiertes Modellieren und Implementieren
  - Klassen- und Objektbegriff
  - Graphische Modellierung
  - Subtyppolymorphismus, Vererbung
  - Schnittstellen

Vorlesungsstruktur und -inhalt V

- Verifikation zuweisungsorientierter Modelle
- Algorithmen und dynamische Datenstrukturen im zuweisungsorientierten Paradigma:
  - Algorithmische Prinzipien: Divide and Conquer, Greedy,…
  - Verkettete Listen, Bäume, Hashing, einfache Graphen
- Ausblick: Deklarative und logische Programmierung

- **Jeden Freitag:** Wichtige Informatiker (5min)

Vorlesungsstruktur und -inhalt VI

voraussichtlich im Sommersemester:

- Modelle der Informatik
  - Definition: Formale Sprachen und BNF
  - Beschreibung formaler Sprachen (Chomsky-2) durch BNF
  - Fixpunktdeutung von BNF-Ausdrücken
  - Chomsky-Hierarchie
  - Automaten und reguläre Ausdrücke
  - Graphen
  - Turingmaschinen

Vorlesungsstruktur und -inhalt VII

- Grundlegende Konzepte des nebenläufigen Programmierens
  - Schwer- und leichtgewichtige Prozesse, Prozesszustände und Operationen auf Prozessen
  - Aktionen und Ereignisse
  - Praktische Umsetzung der Konzepte
  - Spezifikation des zeitlichen Verhaltens und Einplanungen: Synchrone und asynchrone Ablaufmodelle
- Synchronisation und Kommunikation
  - Semaphore, Monitore
  - Erzeuger-Verbraucher, Leser-Schreiber Probleme
  - Synchrone/asynchrone Kommunikation
  - Rendezvous als Beispiel für aktionsorientierte Kommunikation

Literatur

- Goos: Vorlesung über Informatik
  - Band 1: Grundlagen und funktionales Programmieren, Springer Verlag, ISBN 3540672702
  - Band 2: Objektorientiertes Programmieren und Algorithmen, Springer Verlag, ISBN 3540415114
- Broy: Informatik: Eine grundlegende Einführung
  - Band 1: Programmierung und Rechnerstrukturen, Springer Verlag, ISBN 3540632344
- V. Claus, A. Schwill: Duden Informatik
  - Dudenverlag, ISBN 3411100230
- B. Blöchl, C. Meyberg: Repetitorium der Informatik
  - Oldenbourg Verlag, ISBN 3486272160

Literatur II

- Peter Pepper: Funktionale Programmierung in OPAL, ML, HASKELL und GOFER
  - Springer Verlag, ISBN 3540436219
- M. Erwig: Grundlagen funktionaler Programmierung
  - Oldenburg Verlag, ISBN 3486251007
- Peter Thiemann: Grundlagen der funktionalen Programmierung
  - Teubner Verlag
- C. Reade: Elements of Functional Programming
  - Addison-Wesley, ISBN 0201129159
- P.Hudak: The Haskell School of Expression
  - Cambridge University Press, ISBN 0521644089

Literatur III

- P. Rechenberg, G. Pomberger: Informatik Handbuch
  - Hanser Fachbuch, ISBN 3446218424
- U. Schöning: Algorithmik
  - Spektrum Verlag, ISBN 3827410924
- J. van Leeuwen: Handbook of Theoretical Computer Science
  - Volume A: Algorithms and Complexity, ELSEVIER, ISBN 0444880712
- T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen
  - Spektrum Verlag, ISBN 3827410290
- D.Harel: Algorithmics, The Spirit of Computing
  - Addison-Wesley, ISBN 0201504014

Programmiersprachen in Info 1

- Für die Illustration von Informatik-Konzepten: funktionale Programmiersprache Ocaml bzw. F# (MS-Implementierung)
- Zur Vorbereitung auf den harten Alltag des Informatikers wird im Praktikum Java/C# verwendet

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)
- 1990 Vorstellung von Caml-Light
- Heute Objective Caml
- http://caml.inria.fr/
- Implementierung von Standard-Algorithmen in OCaml heute vielfach schneller als beste C++-Compiler, siehe z.B. http://shootout.alioth.debian.org/

Technische Hinweise zum Gebrauch von OCaml

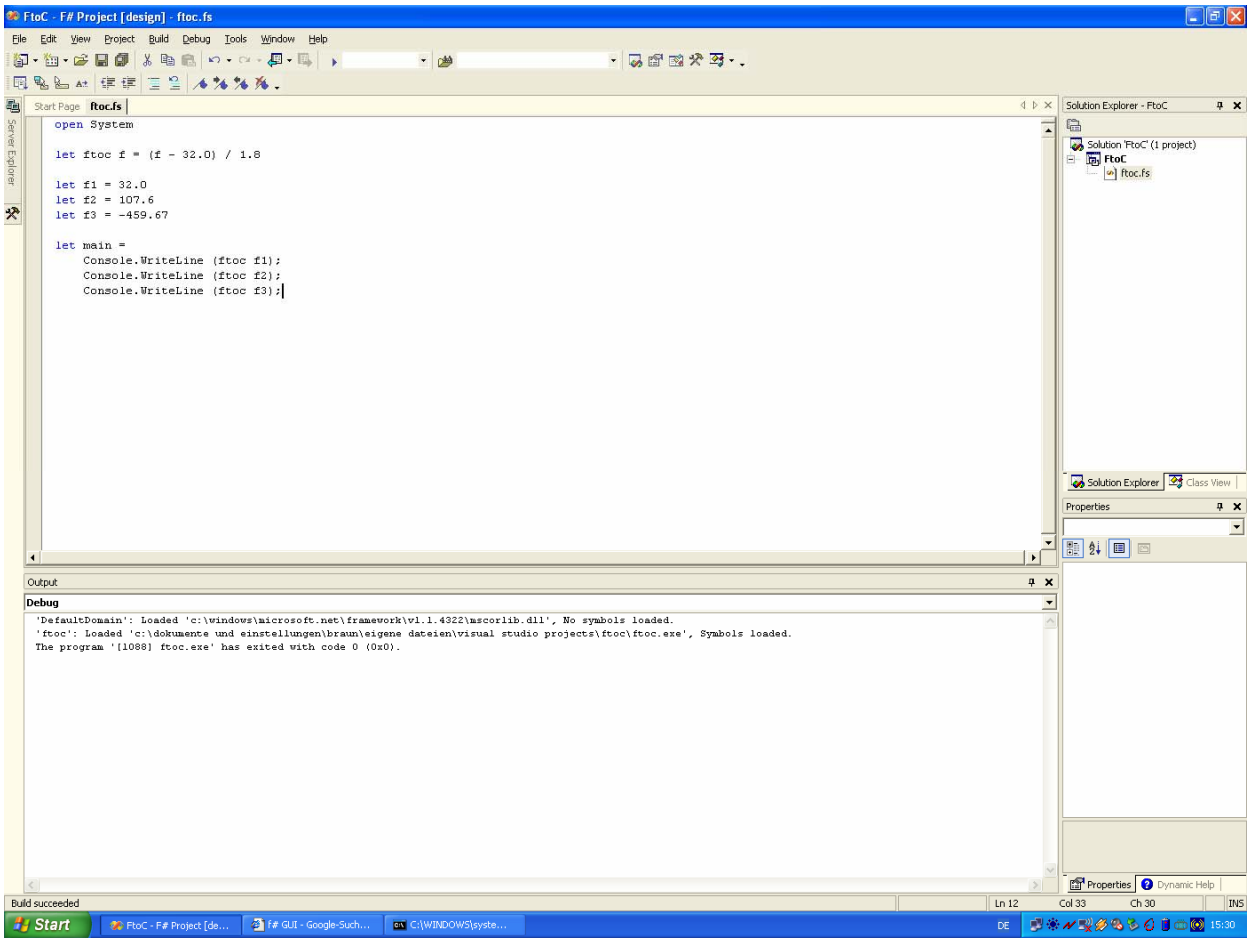
Editor emacs („Editor MACroS“):
http://www.gnu.org/software/emacs/windows/
http://ftp.gnu.org/gnu/windows/emacs/latest: emacs-21.2-fullbin-i386.tar.gz
Installation durch Entpacken (z.B. mit WinZip, erhältlich über http://www.shareware.com)

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.

F# : http://research.microsoft.com/projects/ilx/fsharp.aspx



- Microsoft-Variante von OCaml
- Integriert in Visual Studio & .net
- Studenten können Visual Studio .net unter http://www.maniac.tum.de/ kostenlos herunterladen
- Eine Installationsanleitung für F# ist auf der Info-1-Seite verfügbar

Info I in a nutshell

Klärung von Grundbegriffen

Informelle Illustration einiger Grundkonzepte

Was ist Informatik?

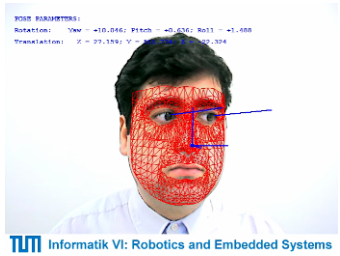
- Informatik ist die Wissenschaft der ***maschinellen Informationsverarbeitung***. Dies umfasst die Fragestellungen zur:
  - schematischen Darstellung (***Repräsentation***) von Informationen: ***Daten- und Objektstrukturen***, sowie deren Bezüge untereinander.
  - Regeln und Vorschriften zur Verarbeitung von Informationen (***Algorithmen, Rechenvorschriften***) und deren Darstellungen einschließlich der Beschreibung von Arbeitsabläufen (***Prozessen, kooperierende Systeme***)

Was ist Informatik?

Informatik ist die Wissenschaft von:

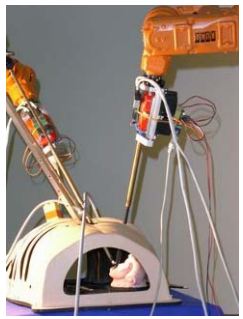
- Der theoretischen Analyse und Konzeption von Informatiksystemen (**Theoretische Informatik**)

Neuronale Netze



- Der organisatorischen und technischen Gestaltung von Informatiksystemen (**Systembezogene Informatik**)

- Der Realisierung von Informatiksystemen, insbesondere der technischen Komponenten (Hardware) (**Technische Informatik**)



Alternative Klassifikation der Informatik

Der "Informatik-Duden" unterteilt die Informatik in vier Bereiche:

- Theoretische Informatik: Formulierung und Untersuchung von Algorithmen, z.B. formale Sprachen, Berechenbarkeitstheorie, Komplexitätstheorie
- Praktische Informatik: Entwicklung von Methoden zur Programmerstellung, z.B. Programmiersprachen, Übersetzer, Betriebssysteme, Softwareengineering
- Technische Informatik: Beschäftigung mit dem funktionellen Aufbau von Computern und den dazugehörigen Geräten, z.B. Rechnerarchitektur, Rechnernetze, Rechnerorganisation
- Angewandte Informatik: Anwendung der Informatik in anderen Wissenschaften, z.B. CAD (computer-aided design)

Aufgabe das Informatikers

- Analyse: Das Verständnis von Problemen aus der Realität
  - Die Probleme können dabei aus vielen verschiedenen Bereichen kommen: soziale Systeme (z.B. Abläufe in Firmen), physikalische Systeme (z.B. Strömungssimulation) oder künstliche (Informatik-)Systeme (z.B. das Internet)
- Synthese (Design): Die systematische Entwicklung einer Lösung des Problems
  - Die Evaluierung und Verwendung von (existierenden/neuen) Hardware- und Softwarebausteinen
- Implementierung: Die Konstruktion der Lösung
  - Mit Hilfe eines Informatik-Systems (auch Datenverarbeitungssystem genannt)
- Wartung: Die Bereitstellung und Betreuung des Informatik-Systems

Welche Fähigkeiten benötigt der Informatiker?

- Analyse von Problemen aus der Wirklichkeit
- Modellierung mit statischen und dynamischen Strukturen (Subsysteme, Klassen, Datenstrukturen und Algorithmen, Automaten)
- Formale Spezifikation von solchen Modellen
- Wiederverwendung von Wissen (Entwurfsmuster)
- Analyse und Synthese von Datenstrukturen und Algorithmen
- Ein gutes Verständnis der theoretischen Grundlagen

- Fähigkeit zur Gruppenarbeit, Interaktion mit Kunden (Reden, Schreiben, Verhandeln, ... , Etikette!)

Eine Klassifizierung von Systemen

- Ziel ist immer die Konstruktion von künstlichen Systemen
- Systeme können aus zweierlei Sicht gesehen werden:
  - **statische Systeme**: Ein System ist eine komplexe Ansammlung von Elementen plus die Beziehung zwischen den Elementen.
  - **dynamische Systeme**: Ein System ist eine Folge von Zustandsübergängen
- Definition: Ein Informatik-System ist ein System, dass auf einem Rechner ausgeführt wird.
  - Informatik-Systeme werden oft auch Datenverarbeitungssysteme genannt
- Insgesamt können Informatik-Systeme z.B. in folgende vier Klassen unterteilt werden:

Systemklasse 1: Berechnen von Funktionen

- Ein System, dass eine Funktion $f:A \rightarrow B$ berechnet, die einen Wertebereich A in einen Wertebereich B abbildet.

  Beispiel: Berechung der Fläche eines Kreises mit Radius r.

Systemklasse 2: Interaktive Systeme

- Ein im Prinzip endlos laufendes System, das Eingangsdaten von anderen Systemen (Prozessen, Menschen) empfängt und Ausgangsdaten an solche Systeme sendet.
- Die Eingangsdaten können den Start und den Halt anderer Systeme (Prozesse, Menschen) bewirken.
- Die Ausgangsdaten sind eine Funktion der Eingabedaten und der Systemgeschichte.

Beispiele: Computerspiele, Prozeßleitwarten, Temperaturreglung, Ampelschaltung

Systemklasse 3: Eingebettete Systeme

- Ein System, das im Verbund mit Komponenten, die nicht der Datenverarbeitung dienen, eine Aufgabe löst.
- Die anderen Komponenten können technische Apparaturen, aber auch Menschen oder betriebliche Organisationen umfassen.

Beispiel: Auto

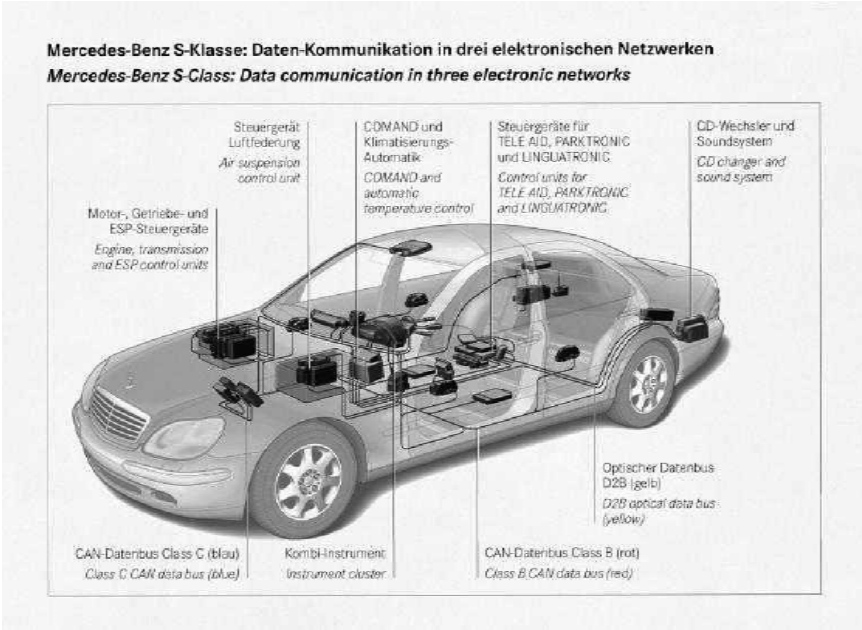
z.B. Mercedes S-Klasse:

3,5 km Kabel,

über 40 Prozessoren

z.B. 7er BMW mehr als

10 Mio. Zeilen Code



Systemklasse 4: Adaptives System

- **Ein (eingebettetes) System, das sich Veränderungen der Wirklichkeit anpasst, insbesondere auch solchen, die das System selbst hervorruft. Das ursprüngliche Modell ist nicht auf Dauer gültig.**

Beispiele: Regelkreis, Ökosysteme

Systeme

- In Informatik 1 werden nur Systeme der Typen *Berechnen von Funktionen* und (kleinste) *Interaktive Systeme* behandelt.
- Offenes vs. geschlossenes System:
  - **Geschlossenes System**: Die Komponenten des Systems haben keine Beziehung zu den Komponenten der Umgebung.
  - **Offenes System**: Die Komponenten des Systems haben eine Beziehung zu den Komponenten der Umgebung. Die Systemschnittstelle kann sich ggf. ändern.
- Die Systemschnittstelle bei offenen Systemen spezifiziert die Art und Weise, wie mit einem offenen System interagiert werden kann.

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 zu einer effizienten Lösung.

Algorithmus: Ursprünge

- Eine der ältesten Beschreibungstechniken für Abläufe:
  - Benannt nach dem Mathematiker Al-Khwarizmi (ca. 780...840), der am Hof der Kalifen von Bagdad wirkte.
- Bereits in der Antike kannte man informelle algorithmische Beschreibungen für Rechenverfahren, z.B. den berühmten Euklidischen Algorithmus:

  „**Nimmt man beim Vorliegen zweier ungleicher Zahlen abwechselnd immer die kleinere von der größeren weg, so müssen, wenn niemals ein Rest die vorangehende Zahl genau misst, bis die Einheit übrig bleibt, die ursprünglichen Zahlen gegeneinander prim (teilerfremd) sein**“

  aus Euklids Buch VII, §1, zitiert nach Gericke H.: Mathematik in Antike und Orient – Mathematik im Abendland. Fourier, Wiesbaden, 3. Aufl. 1994.

Eingabe: 12, 8
12 – 8 = 4
8 – 4 = 4
⇒ 12, 8 nicht prim

Eingabe; 13, 9
13 – 9 = 4
9 – 4 = 5
5 – 4 = 1
⇒ 13, 9 prim

Algorithmus: Ursprünge

Euklid, zweite Fassung:

- Gegeben: Zwei Variable *M* und *N* aus Nat
- Gesucht ggT(m,n): größte Zahl, die sowohl Teiler von *M* als auch von *N* ist
- S1: -- Berechnung Rest
  Dividiere *M* durch *N* und weise *R* den (ganzzahligen) Rest der Division zu. Damit gilt auch $0 \leq R < N$.
- S2: -- $R = 0$?
  Wenn $R = 0$, dann terminiere mit dem Ergebnis *N*.
- S3: -- Tausch
  Setze $M := N$ und $N := R$. Fahre bei S1 fort.

Algorithmus: Definitionen

- „Ein **Algorithmus** ist ein Verfahren
  - mit einer präzisen (d.h. in einer genau festgelegten Sprache abgefassten)
  - endlichen Beschreibung
  - unter Verwendung
    - effektiver (d.h. tatsächlich ausführbarer)
    - elementarer (Verarbeitungs-) Schritte.“

  (nach Broy, M.: Informatik: Eine grundlegende Einführung, Band 1. Springer-Verlag, Berlin, 2. Auflage, 1998)
- Algorithmus nach [Knuth 1973]:
  „Ein Algorithmus ist eine endliche Menge von Regeln, die eine endliche Folge von Operationen zur Lösung eines Problems beschreibt. “

Einfache Funktionen in Ocaml

- Umrechnung von Temperatureinheiten: Fahrenheit in Celsius
- Umrechnungsformel: $T_{Celsius} = 5/9 * (T_{Fahrenheit} - 32)$
- Funktionale Implementierung (Ocaml):

```
let temperature_Celsius (t_fahrenheit) =  (5.0 /. 9.0) *. ( t_fahrenheit -. 32.0);;
temperature_Celsius 30.0;;
```

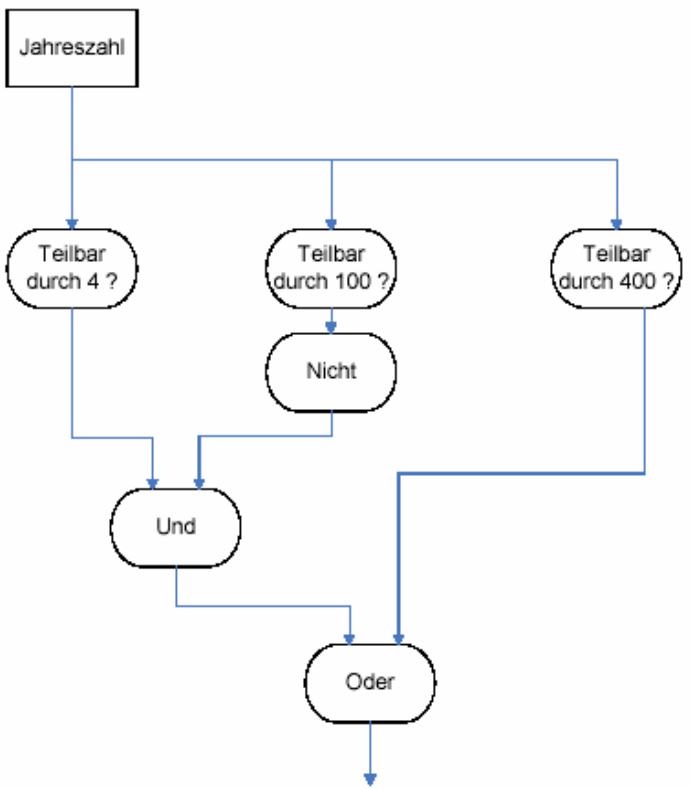
Funktionen über ganzen Zahlen und boolesche Operatoren: Schaltjahre

- Beispiel: Ein Umlauf der Erde um die Sonne, das so genannte tropische Jahr dauert genau 365,2422 Tage. Deshalb wurde schon zu Cäsars Zeiten jedes vierte Jahr ein Schalttag hinzugenommen. Dann geht jedoch nach circa 128 Jahren der Kalender einen Tag vor. Deshalb führte Papst Gregor im Jahre 1545 folgende Schaltjahresregel ein: Ein Jahr ist ein Schaltjahr genau dann, wenn die Jahreszahl durch 4 und nicht durch 100 oder aber durch 400 teilbar ist.

Offenbar werden Funktionen benötigt, die Folgendes leisten:

- Überprüfung, ob eine Zahl m durch eine Zahl k teilbar ist,
- Verknüpfung von Aussagen durch die Operationen **`und`**, **`oder`**, **`nicht`**

Funktionen über ganzen Zahlen und boolesche Operatoren: Schaltjahre: Implementierung




```
let schaltjahr (jahreszahl) =
    (jahreszahl mod 4 = 0) &&
    not (jahreszahl mod 100 = 0) ||
    (jahreszahl mod 400 = 0);;
```

Funktionen mit bedingten Ausdrücken

Der bedingte Ausdruck ist eine besondere Funktion mit drei Argumenten :

- Einem booleschen Ausdruck: Über den Wert dieses booleschen Ausdrucks wird der weitere Programmablauf gesteuert.
- Einem Ausdruck A; der Wert von A ist das Resultat des bedingten Ausdrucks, wenn der boolesche Ausdruck den Wert `wahr` annimmt.
- Einem Ausdruck B; der Wert von B ist das Resultat des bedingten Ausdrucks, wenn der boolesche Ausdruck den Wert `falsch` annimmt.

Verzweigungen: Der bedingte Ausdruck

- Beispiel: Postpaketgebühren (Daten und Werte sind frei erfunden, eine Übereinstimmung mit realen Postgebühren wäre rein zufällig)

$$\texttt{Preis (Gewicht)} = \begin{cases} 2{,}50\ \text{Euro}, & 0\ \text{kg} < \text{Gewicht} < 1\ \text{kg} \\ 3{,}50\ \text{Euro}, & 1\ \text{kg} < \text{Gewicht} < 2\ \text{kg} \\ 5{,}50\ \text{Euro}, & 2\ \text{kg} < \text{Gewicht} < 4\ \text{kg} \\ 7{,}50\ \text{Euro}, & 4\ \text{kg} < \text{Gewicht} < 8\ \text{kg} \\ \text{Ablehnung}, & 8\ \text{kg} < \text{Gewicht} \end{cases}$$

Verzweigungen: Der bedingte Ausdruck

- Postpaketgebühren: Implementierung

```
let preis x = if (0.0 < x && < 1.0) then 2.5 else
              if (1.0 <= x && x < 2.0) then 3.5 else
              if (2.0 <= x && x < 4.0) then 5.5 else
              if (4.0 <= x && x < 8.0) then 7.7
              else failwith "Ablehnung";;
```

Was ist bereits vorhanden?

- Beschränkung auf *Systemklasse 1*: Funktionsberechnung $f:A \rightarrow B$
- Funktion *f* kann einfache oder aufwendig Berechnungen enthalten, *A* und *B* sind Eingangs- und Rückgabewert.
- *A* und *B* haben jeweils einen *Typ*; dieser Datentyp wird bei der Funktionsdefinition festgelegt. Übersetzer leitet mit Hilfe eines sogenannten Typ-Inferenzalgorithmus den Typ des Rückgabewerts soweit wie möglich automatisch ab.

- Funktionsdefinition in Ocaml (Bsp.: Temperaturkonversion)
- Funktionen über ganzen Zahlen (Bsp.: Kalender)
- Funktionen mit bedingten Ausdrücken (Bsp.: Paketpreisberechnung)

Möglichkeiten zur Funktionsdefinition I

- Benannte Funktionsdefinition:
```
let f2 x = if x < 3 then "Less" else "Greater";;
```
- Weitere Möglichkeit – anonyme Funktion:
```
fun x -> x*x;;
```
Anwendung mit:
```
(fun x -> x*x) 2;;
```
- Funktionsdefinition mit mehreren Parametern:
```
let f2 x y = if x*y < 5 then "Less" else "Greater";;
```
- Anonyme Funktion mit mehreren Parametern:
```
(fun x y -> x*y);;
```
oder `fun x -> fun y -> x*y;;`
Anwendung mit:
```
(fun x -> fun y -> x*y) 2 3;;
```

Möglichkeiten zur Funktionsdefinition II

- Falls für Wiederverwendung erforderlich, können Funktionen mit „`fun`“-Syntax auch benannt werden:
```
let a_mal_b = fun a -> fun b -> a*b;;
```
Anwendung mit:
```
a_mal_b 3 4;;
```
- Besonders nützlich: `let ... in` – Konstrukt zur komfortablen Definition geschachtelter Funktionen:
```
let pow3 x =
    let sqr x = x * x in
    x * sqr x;;
```
oder:
```
let pow3 x =
    let y = x * x in
    x * y;;
```

Verschattungsregeln

- Variablen in Ocaml: Einmalzuweisung ("single assignment")
- Variablenwert wird zum *Zeitpunkt der Funktionsdefinition* festgelegt (`let` definiert "closure" für Funktionsanwendung (siehe später))
- Beispiel:

```
let p = 10;;
let f x = (x, p, x + p);;
let p = 1000;;

# f p;;
- : int * int * int = (1000, 10, 1010)
```

- Weiteres Beispiel:

```
let x=1 in
    let x=2 in
    let y=x+x in
    x+y;;
- : int = 6
```

Lokale Deklarationen mit `let … in`

- Bei `let name = expr1 in expr2;;` ist der Wert von `name` nur bei der Auswertung von `expr2` gültig (und hat den Wert von `expr1`)
- Beispiel:

```
let x = 2;;
let x = 3 in x * x;;

# let x = 2;;
val x : int = 2
# let x = 3 in x * x;;
- : int = 9
# x;;
- : int = 2
```

Wichtig bei rekursiven Funktionen (siehe später)

Lokale Wiederverwendung von Code

- Definition einer Funktion zur lokalen Verwendung

```
let ipow4 x = let sqrx = x*x in
                (sqrx)*(sqrx);;

val ipow4 : int -> int = <fun>
```

- Definition von zwei Funktionen in einem Paar:

```
# let(ipow3, ipow4)=let sqr x = x*x in
     ((fun x->x*(sqr x)), fun x->(sqr x)*(sqr x));;
val ipow3 : int -> int = <fun>
val ipow4 : int -> int = <fun>
```

Datentypen und ihre Spezifikation

- Die von uns verwendeten Programmiersprachen erlauben die Definition von Datentypen.
- Vorteil: Übersetzer kann automatisch prüfen, ob die Operation, die auf Werten bestimmten Typs ausgeführt werden sollen, zulässig sind. Beispiel: Multiplikation zweier Variablen, die jeweils einen Buchstaben (statt einer Zahl) repräsentieren, ist normalerweise nicht sinnvoll
- Der Begriff des Datentyps („Sorte“) beinhaltet die eigentliche Repräsentation von Werten („Bits und ihre jeweilige Bedeutung“) und die darauf definierten Operationen.
  - Beispiel: Natürliche Zahlen in 8-bit, 16-bit , …, N-bit-Darstellung und die darauf definierten Grundrechenarten (`+`, `-`, `*`, `DIV`).
  - Siehe dazu später die Konzepte der „Rechenstruktur“ und der „Algebra“ im mathematischen Sinne.

Datentypen und ihre Spezifikation

- Gelegentlich ist es nützlich:
  - die Repräsentation zu ändern (etwa mehr/weniger Bits),
  - Operationen zu verbieten, abzuändern oder neue hinzuzufügen,
  - Operationen zu definieren, die auf unterschiedlichen Datentypen zulässig sind.
- Für letzteres führt man das Konzept der *Polymorphie* („Vielgestaltigkeit“) ein, d.h. die Möglichkeit, Operationen für unterschiedliche Datentypen zu definieren, aber gleich zu benennen.
- Übersetzer erkennt dann vorliegenden Typ und wendet die darauf zulässige Operation an.
- In der Mathematik schon immer üblich, Beispiel: „*“ für natürliche, ganze, reelle, komplexe Zahlen, aber auch für Matrixmultiplikation und „und“-Verknüpfung in Schaltalgebra

Datentypen und ihre Spezifikation

- Beispiele für polymorphe Funktionen: Identität und Funktionskomposition

```
# let id x = x;;
val id : 'a -> 'a = <fun>
val id : 'a -> 'a = <fun>
# id 1234;;
- : int = 1234
# id 12.34;;
- : float = 12.34
```

- Komposition:

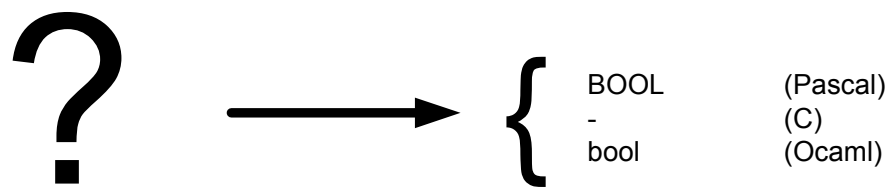
```
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in compose mul5 add1 9;;
- : int = 50
# let add1 x = x+1 and mul5 x = x*5 in compose add1 mul5 9;;
- : int = 46
```

Repräsentation von Daten auf dem Rechner

- Bei der Programmierung ist neben der geeigneten *Spezifikation von Rechenvorschriften* die andere wesentliche Aufgabenstellung die der *Repräsentation* von Informationen aus der Umwelt auf dem Rechner.
- Die einzelnen Programmiersprachen bieten verschiedene Datentypen zur Verwendung an.
- Dabei wird zunächst zwischen einfachen und komplexen (zusammengesetzten) Datentypen unterschieden.
- Im folgenden Darstellung der wichtigsten dieser Datentypen.

Wahrheitswerte

- Oftmals beschränkt sich der Informationsgehalt auf die Wahrheitswerte: **wahr** oder **falsch**?
- Beispiel: Ja-Nein-Fragen
- Programmiersprachen stellen hierfür vielfach einen „bit“-Datentyp zur Verfügung



Wahrheitswerte / Boolsche Algebra

- Zur Manipulation von Wahrheitswerten stehen diverse Operationen zur Verfügung:
  - $\wedge$ (Konjunktion/und)

$\wedge$	**true**	**false**
true	true	false
false	false	false

  - $\vee$ (Disjunktion/oder)

$\vee$	**true**	**false**
true	true	true
false	true	false

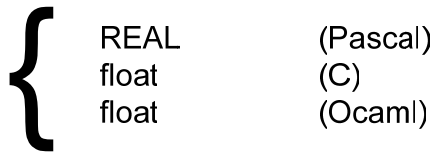
  - $\neg$ (Negation/nicht)

$\neg$	**true**	**false**
	false	true

- Weitere Operatoren: = (Äquivalenz), $\rightarrow$ (Implikation)

Kontinuierliche Größen

- Ein wichtiger Datentyp ist der für kontinuierliche Werte; diese kommen an vielen Stellen in unserer natürlichen Umwelt vor
- Im Alltag verwenden wir reelle Zahlen, um solche Werte auszudrücken
- Beispiele: Temperatur, Gewicht, Körpergröße, Ströme, …
- Auf dem Rechner stehen Gleitkommazahlen zur Repräsentation einer *Annäherung* an diese Werte zur Verfügung
- Wichtigster Standard für die Repräsentation von Gleitkommazahlen (32-bit, 64-bit und 128 bit) ist IEEE 754-1985



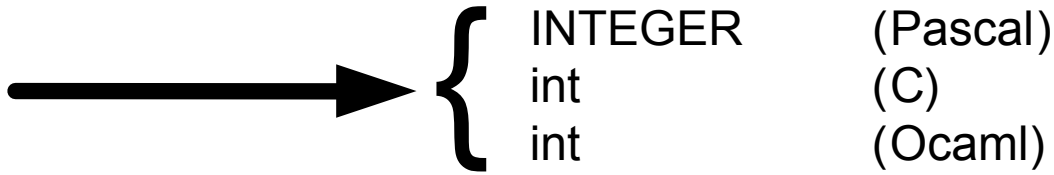
Operatoren: Gleitkommazahlen (FLOAT, REAL)

- Zur Darstellung von reellen Zahlen werden Gleitkommazahlen verwendet
- Gleitpunktzahlen *approximieren* reelle Zahlen mit vom System festgelegten Genauigkeit

Operation	**typischer Operator (bzw. in OCaml)**
Addition	+ bzw. +.
Subtraktion	- bzw -.
Multiplikation	* bzw. *.
Division	/ bzw. /.
Potenzierung	^,**, o.ä.

Ganzzahlige Werte (INTEGER, CARDINAL)

- Häufig werden auch *ganzzahlige* Werte benötigt
- Beispiele: Anzahl Studenten, Hausnummer
- Unterschiedliche, rechnerspezifische interne Repräsentationen für diesen Datentyp möglich und üblich



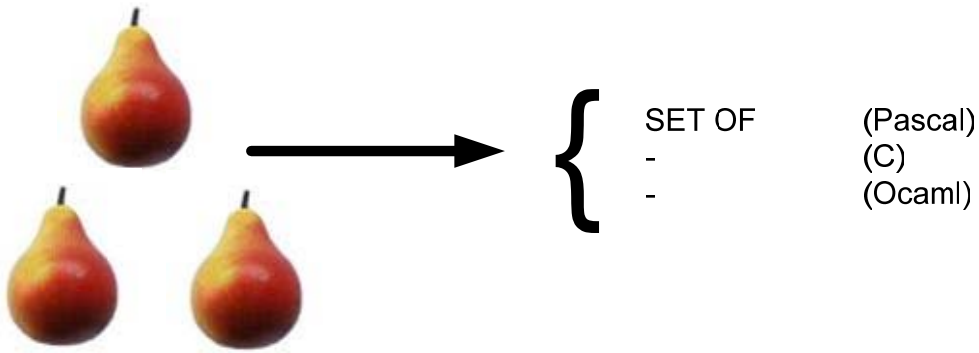
Operatoren für Ganzzahlen

- Bei der Division von Ganzzahlen werden die Ergebnisse abgerundet
- Die Modulo-Operation `mod` dient zur Berechnung des Restes bei einer Division
- Zusätzlich in OCaml: Operatoren zur Verschiebung des Bitmusters der Repräsentation:
  `z lsl n`: "logical shift left" von `z` um `n` Bitstellen
  und weitere Schieboperatoren `asl` sowie `asr`
- Außerdem: bitweise logische Verknüpfungen zwischen Repräsentationen zweier Ganzzahlen:
  `i land j`, `i lor j`, `i lxor j`.

Operation	**typischer Operator**
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Modulo	mod, %
Potenzierung	^,**

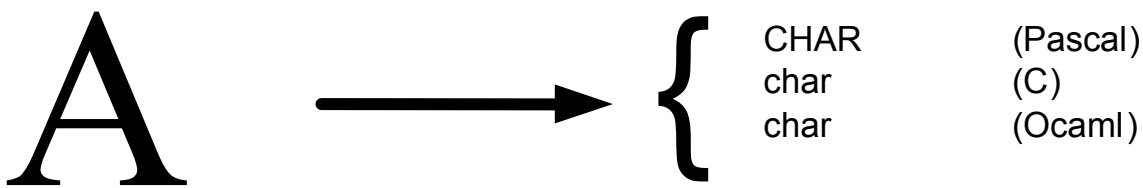
Mengen (Sets)

- Oftmals sollen mehrere Objekte des gleichen Typs gespeichert werden
- Einige Programmiersprachen bieten deshalb den komplexen Datentyp *set* an
- Typischerweise werden für diesen Datentyp Operatoren zur Vereinigung (*add*, *union*), zur Bildung der Schnittmenge (*intersection*), und Operatoren zum Hinzufügen und Löschen von Elementen angeboten



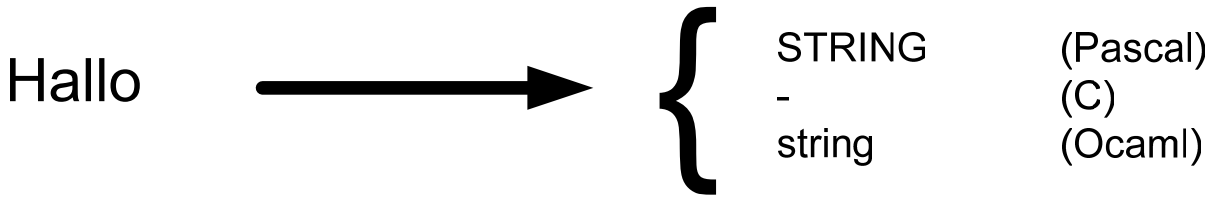
Zeichen (Characters)

- Auch Buchstaben und sonstige Zeichen können repräsentiert werden.
- Dazu wird ein Datentyp "Character" bereitgestellt.
- Es existieren unterschiedliche Konventionen für die internen Repräsentationen diese Datentyps, z.B. nur für Großbuchstaben, für Groß- und Kleinbuchstaben, für chinesische Schriftzeichen, … (CDC 6bit, EBCDIC, ASCII, ISO-Latin1, Unicode, ….)



Zeichenketten

- Zur Repräsentation von Wörtern und Texten werden Zeichen*ketten* benötigt.
- Zur Darstellung von Zeichenketten bieten die meisten Programmier-sprachen einen eigenen Datentyp *string* an.



Zeichen und Zeichenketten

- Zeichen werden beispielsweise durch einfache Anführungszeichen markiert: z.B. `‘f‘,‘4‘,‘/‘`
- Zeichenketten werden meistens durch Anführungszeichen markiert: z.B.`“Hallo“`
- Wichtigste Operation auf Zeichenketten ist die Zusammenfügung ("Konkatenation"), typischer Operator ist + oder ^
- Zumeist bieten die Programmiersprachen noch Funktionen zur Umwandlung von Zeichenketten in Zahlen und umgekehrt, sowie zur Ermittlung der Anzahl der Zeichen an, gelegentlich auch zur Einfügung von Zeichen in Zeichenketten an einer bestimmten Stelle.

Definition eines eigenen Datentyps „Zeichenkette“

- Wir versuchen, einen eigenen Datentyp „Zeichenkette“ auf der Basis des Datentyps „Zeichen“ zu implementieren.
- Ansatz: Eine Zeichenkette besteht aus einzelnen Zeichen. Eine Zeichenkette kann deshalb als ein Zeichen plus den Rest der Zeichenkette (dies ist evtl. eine leere Zeichenkette) aufgefaßt werden.
- Eine Zeichenkette kann somit leer sein oder sie besteht aus einem ersten Zeichen gefolgt von einer Zeichenkette
- Funktionale Implementierung:

```
type myString = Empty | CharListCons of char * myString;;
```

(Verwendung siehe später, `Empty` und `CharListCons` sind Konstruktoren; siehe später)

- Pascal-Implementierung:

```
TYPE ownString = ^elemOfString;
elemOfString = RECORD
      sign : CHAR;
      succ : ownString;
END;
```

- Einen solche Datentyp nennt man **rekursiv**, da in der Definition der eigene Datentyp verwendet wird (er erscheint auf beiden Seiten der Dekalration)

Tupel

- Oftmals werden in Objekten Daten verschiedener Datentypen zusammen-gefaßt
- Die Anzahl und die Typen der Elemente sind dabei zumeist im voraus bekannt
- Die Programmiersprachen bieten hierzu „Tupel“ an.
- Beispiel: Ausweis mit dem Namen, dem Geburtsdatum, …

RECORD (Pascal)
struct (C)
{} (Ocaml)

Tupel

- Bereits in vorhergehenden Beispielen haben wir Funktionen mit mehreren Argumenten kennen gelernt:
- `let monatslaenge (monat, jahr) = ….`
- Das Argument dieser Funktion ist kein elementarer Datentyp, sondern ein Tupel
- Mathematische Definition: Gegeben seien die Mengen A_1, …, A_n sowie das Kreuzprodukt $A_1 \times A_2 \times$, …., $\times A_n$. Dann nennt man ein Element dieses Kreuzproduktes ein Tupel (a_1, a_2, …., a_n)
- Die Reihenfolge ist hier wesentlich (im Unterschied zu Mengen)
- In der Informatik entsprechen den Mengen A_1, …, A_n Datentypen (Zahlen, Wahrheitswerten, selbstdefinierten Datentypen, usw.)
- Beispiel: (1, Januar, 2006) ist ein Element der Menge integer $\times$ Monat $\times$ integer

Beispiel für Tupel

- Eine Datumsangabe setzt sich aus Tages-, Monats und Jahres-angabe zusammen. Man könnte einen entsprechenden Datentyp folgendermaßen definieren:

```
# type monat = Januar | Februar | März | April
       | Mai Juni | Juli | August | September
       | Oktober | November | Dezember;;

# type datum = Datum of (int * monat * int);;

# let heute = Datum (1, Januar, 2005);;

val heute : datum = Datum (1, Januar, 2005)
```

Beispieldefinitionen: Tupel

Erläuterungen:

- Das Kreuzprodukt wird in Ocaml durch „*“ dargestellt
- Operatoren zur Konstruktion von neuen Datentypen aus bekannten heißen Konstruktoren. Konstruktoren können Argumente haben (hier ist „Datum“ ein derartiger Konstruktor)
- Die Argumente von Konstruktoren sind Typ-Ausdrücke, d.h. Ausdrücke, die aus Typvariablen aufgebaut sind; in obigem Beispiel ist (int * monat * int) ein derartiger Typausdruck
- Typausdrücke setzen sich aus Typnamen („int“, „float“, selbstdefinierte Typnamen) und Typoperatoren (hier „*“, oder weiteren Konstruktoren) zusammen
- Ein Datentyp wird erzeugt, indem der Konstruktor mit geeigneten Argumenten aufgerufen wird

Records: Benannte Tupel

- Alternativ zur Verwendung eines Tupels könnte man den Datentyp „`Datum`“ auch mittels eines Records, (d.h. einesTupels bei dem die einzelnen Elemente einen Namen tragen) definieren:

```
type month = Januar | Februar | März | April | Mai
           | Juni | Juli | August | September
           | Oktober | November | Dezember;;

type date = {tag : int; monat : month; jahr : int};;

let heute = {tag = 1; monat = Januar; jahr = 2005};;
```

Erläuterung:

Im Gegensatz zur vorherigen Typdefinition haben nun die einzelnen Elemente des Datentyps Namen

Diese Art der Typdefinition nennt man **Record**

Pattern-matching mit Tupeln/Records

- Tupel und Records werden komfortabel über Pattern-matching manipuliert
- Beispiel: Aus obigen Datumsangaben soll der Wochentag extrahiert werden (Selektorfunktion)

```
let selectDatum d = match d with
       | Datum (tag, monat, jahr) -> tag;;

let selectDate d = match d with
       | { tag = t; monat = m; jahr = j} -> t;;
```

- Erläuterungen
  - Pattern-matching mit Tupeln ist eine sehr häufige Operation in OCaml
  - Die Klammern und Kommata / Semikola bilden die Konstruktoren von Tupeln und Records

Weiteres zum Pattern-matching

- Problem: Der Anwender der Funktion `selectdate d` muss ein bestimmtes Format verwenden; die Funktion soll nun derart erweitert werden, dass mehrere Formate verwendbar sind.
- Dazu definieren wir zunächst folgende Datentypen:

```
type datum = (int * month * int);;
type date = {tag : int; monat : month; jahr : int};;

type variantDatum = Datum of datum | Date of date;;
```

- Folgende Funktion leistet dann das Gewünschte:

```
let selectTag d = match d with
      | Datum (t, m, j) -> t
      | Date ({ tag = t; monat = m; jahr = j}) -> t;;
```

- Aufruf z.B. auf folgenden Werten möglich:

```
let heute = Date ({tag = 1; monat = November; jahr
= 2005});;
let morgen = Datum (2, November, 2005);;
```

Beispiel: Geometrische Objekte

- Fläche geometrischer Figuren:

```
type punkt = {x: float; y:float};;

type figur  =  Punkt of punkt |
               Linie of (punkt * punkt) |
               Kreis of (punkt * float) |
               Rechteck of (punkt * punkt);;

let abs z = if (z < 0.0) then (-1.0 *. z) else z;;

let flaeche fig = match fig with
        | Punkt (_) | Linie (_) -> 0.0
        | Kreis (_, radius) -> 3.14 *. radius**(2.0)
        | Rechteck ({x = x1; y = y1}, {x = x2; y = y2}) ->
                              abs ((x2 -. x1) *. (y1 -. y2));;
```

Polymorphe Datentypen

- Bislang wurde stets präzise angegeben, welche elementaren Typen in Definition involviert sind
- Beispiel
  - Der Punkt wurde als Tupel über Fließkommazahlen definiert
- Bei ähnlichen Problemen könnte es jedoch auch genügen, nur ganze Zahlen zu verwenden
- Folge: Die entsprechenden Datentypdefinitionen und Funktionen müssten nochmals geschrieben werden
- Forderung: Es müsste eine Möglichkeit geben, Datentypen zu definieren, bei denen im Typausdruck der Typ variabel gestaltet werden kann

Polymorphe Datentypen

- Ein Datentyp, bei dem der Typ der Komponenten variabel ist, nennt man polymorphen Datentyp
- Beispiel: Polymorphe Definition des Datums

```
type 'a datum = Datum of (int * 'a * int);;
```

- Die Typvariable steht vor dem Typnamen und ist durch das „' “ - Zeichen als Typvariable zu erkennen
- Diese Definition hat gegenüber der ursprünglichen den Vorteil, daß der Typ des 2. Tupelelements frei ist
- Bei der Instantiierung wird durch so genannte Typinferenz (Ableitung des Typs) die Signatur des Typs ermittelt.

Polymorphe Datentypen

- Ein konkretes Datum könnte folgendermaßen instantiiert werden:

```
# let heute = Datum (13, "Mai", 2005);;
val heute : string datum = Datum (13, "Mai", 2005)
# let morgen = Datum (24, 10, 2005);;
val morgen : int datum = Datum (24, 10, 2005)
```

- Es können auch mehrere Typvariablen verwendet werden.
- Sollte beispielsweise in obiger Definition auch der Typ des Tages variabel bleiben (etwa um durch Verwendung von Fließkommazahlen auch die Uhrzeit mit zu erfassen), so würde man definieren:

```
type ('a, 'b) datum = Datum of ('b * 'a * int);;
```

- und instantiieren:

```
# let jetzt = Datum (24.438, Oktober, 2005);;
val jetzt : (month, float) datum = Datum (24.438, Oktober, 2005)
```

Polymorphe Funktionen

- Bei den bisherigen Funktionen konnte die Signatur (= Typ des Rückgabewertes) der Funktion stets eindeutig abgeleitet werden (Typinferenz)
- Beispiel: Allgemeine Selektorfunktion auf Tupel

```
# let selektor t = match t with
                  |(a, b) -> a;;
val selektor : 'a * 'b -> 'a = <fun>
```

- Die Signatur der Funktion kann hier nicht abgeleitet werden, da keine Operationen verwendet werden, die eine Typinferenz erlauben

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

- Die Definition der Funktion „`equality`“ enthält den Gleichheitsoperator, aus dem die Typgleichheit von a und b, sowie der Typ des Ergebnis inferiert werden kann.

Listenartige Datentypen

- Listen können verwendet werden, um Daten gleichen Typs zu sammeln
- Beispiel:
  - Liste ganzer Zahlen:
```
# [3; 4; -1; 5];;
- : int list = [3; 4; -1; 5]
```
  - Leere Liste:
```
# [];;
- : 'a list = []
```
  - Anhängen eines Elementes an eine Liste (von vorne!) mit Hilfe des Doppelpunktoperators:
```
# -10 :: [3; 4; -1; 5];;
- : int list = [-10; 3; 4; -1; 5]
```
  - Äquivalenz: `[3; -7; 5] = 3 :: (-7 :: (5 :: []))`

Polymorphe Operationen auf Listen

- Wir sehen:
  - Listen lassen sich aus der leeren Liste „`[]`“ und dem Operator „`::`“ aufbauen
  - „`[]`“ und „`::`“ sind die Konstruktoren von Listen
  - Über „`[]`“ und „`::`“ läßt sich Pattern-matching durchführen
- Erstes Listenelement:

```
let first list = match list with
        | x :: r -> x
        | [] -> failwith "Liste ist leer";;

val first : 'a list -> 'a = <fun>
```

- Die Funktion „`first`“ ist eine polymorphe Funktion, die eine Liste beliebigen Typs auf diesen Datentyp abbildet.

Polymorphe Operationen auf Listen

- Liste ohne das erste Element:

```
let rest list = match list with
        | x :: r -> r
        | [] -> failwith "Liste ist leer";;

val rest : 'a list -> 'a list = <fun>
```

- Bemerkung: Durch die Funktionen „`rest`“ und „`first`“ kann im Prinzip auf jedes Element einer Liste zugegriffen werden:

```
# let liste = [-10; 3; 4; -1; 5];;
val liste : int list = [-10; 3; 4; -1; 5]
# first (rest (rest (rest liste)));;
- : int = -1
```

Typdefinition für listenartige Datenstrukturen

- Die Liste ist in Ocaml ein vordefinierter Datentyp
- Wie könnte die entsprechende Typdefinition aussehen ?
- Analyse des Datentyps „Liste“
  - Datentyp besteht aus zwei Konstruktoren: (i) Leere Liste („[ ]“) und (ii) Doppelpunktoperator, der ein neues Element an eine schon bestehende Liste hängt
  - Der Doppelpunktoperator hat zwei Argumente:
    - Erstes Argument: Das neue Listenelement vom Typ „'a“
    - Zweites Argument: Die Liste, an die das Element gehängt werden soll vom Typ „'a `list`“
- Damit ergibt sich für listenartige Strukturen folgende abstrakte Typdefinition (= Datenstruktur und Zugriffsoperationen); s.a. Informatik-Duden: Unter einem Datentyp versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit

```
type 'a lifo = Nil | Prepend of ('a * 'a lifo);;

let rest list = match list with
        | Prepend (x, r) -> r
        | Nil -> failwith "Liste ist leer";;

let first list = match list with
        | Prepend (x, r) -> x
        | Nil -> failwith "Liste ist leer";;

let example = Prepend("dies", Prepend("ist", Prepend( "eine",
                                    Prepend( "Liste", Nil))));;
# rest example;;
- : string lifo = Prepend ("ist", Prepend ("eine", Prepend ("Liste", Nil)))
# first example;;
-: string = "dies"
# first (rest example);;
- : string = "ist"
```

- Unterschied zwischen der hier definierten Struktur und den vorimplementierten Listen: statt des Infixoperators „`::`“, der Präfixoperator (Konstruktor) „`Prepend`“ verwendet wird.

- Stellt man nur die Funktion „`first`“ zur Verfügung, kann nur auf das jeweils zuletzt an die Liste gehängte Element zugegriffen werden: **L**ast **i**n **F**irst **o**ut. Frage: wie implementiert man „konsumierenden“ Zugriff?

```
type 'a lifo = Nil | Prepend of ('a * 'a lifo);;

let first list = match list with
         | Prepend (x, r) -> x
         | Nil -> failwith "Liste ist leer";;

let example = Prepend("dies", Prepend("ist", Prepend( "eine",
                                    Prepend( "Liste", Nil))));;
# first example;;
-: string = "dies"
# first example;;
-: string = "dies"
```

Von LiFo zu FiFo

- First in First out (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
      | Nil -> failwith ("undefined")
      | Prepend (x, Nil) -> 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
              | Nil -> failwith ("undefined")
              | Prepend (x, Nil) -> Nil
              | Prepend (x, ll) -> Prepend (x, lrest ll);;
val lrest : 'a liFo -> 'a liFo = <fun>
```

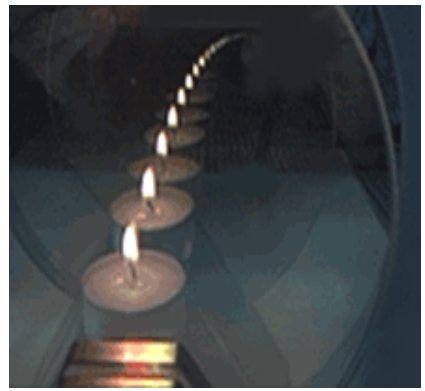
Rekursion

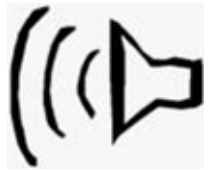
Rekursion

- **Definition:** Als *Rekursion* wird der Aufruf oder die Definition einer Funktion durch sich selbst bezeichnet.
- Hinweis: *Iteration* bedeutet die wiederholte Anwendung einer Rechenvorschrift (häufig auf dem Ergebnis der vorigen Anwendung)
- Rekursion ist wesentliches Konstrukt in funktionalen Programmier-sprachen und ersetzt "Schleifen" imperativer Sprachen (siehe später).
- Um einen "endlosen Regreß" zu verhindern, muss der Programmierer garantieren, dass die Funktion terminiert.
- Erfolgt pro Funktionsausführung maximal ein erneuter Aufruf der Funktion, so spricht man von **linearer** Rekursion.
- Sowohl Funktionen als auch Datenstrukturen können rekursiv sein.

Rekursion kommt im Alltag vor

„visuelle Rekursion“: Bild im Bild

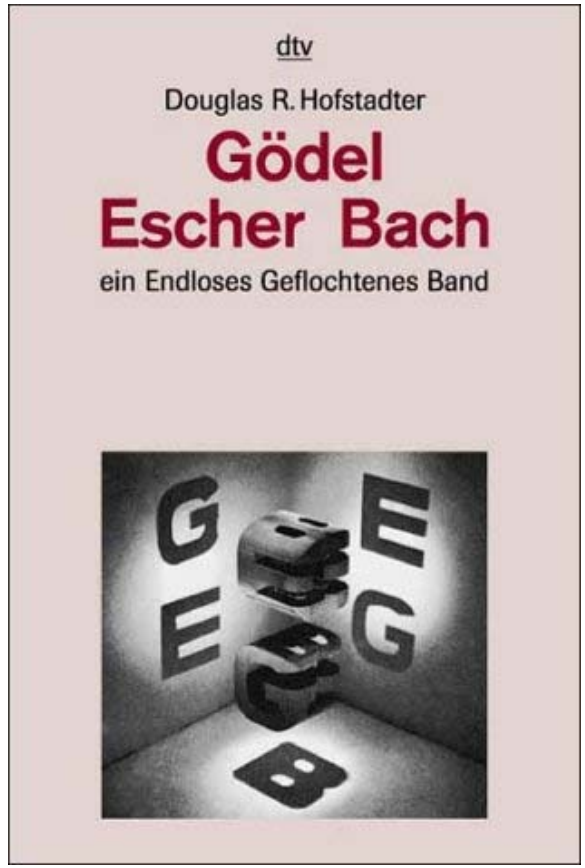




akustische Rückkopplung

Literatur

- Douglas R. Hofstadter: Gödel, Escher Bach, ein Endloses Geflochtenes Band, 1979
- Pulitzerpreis 1980

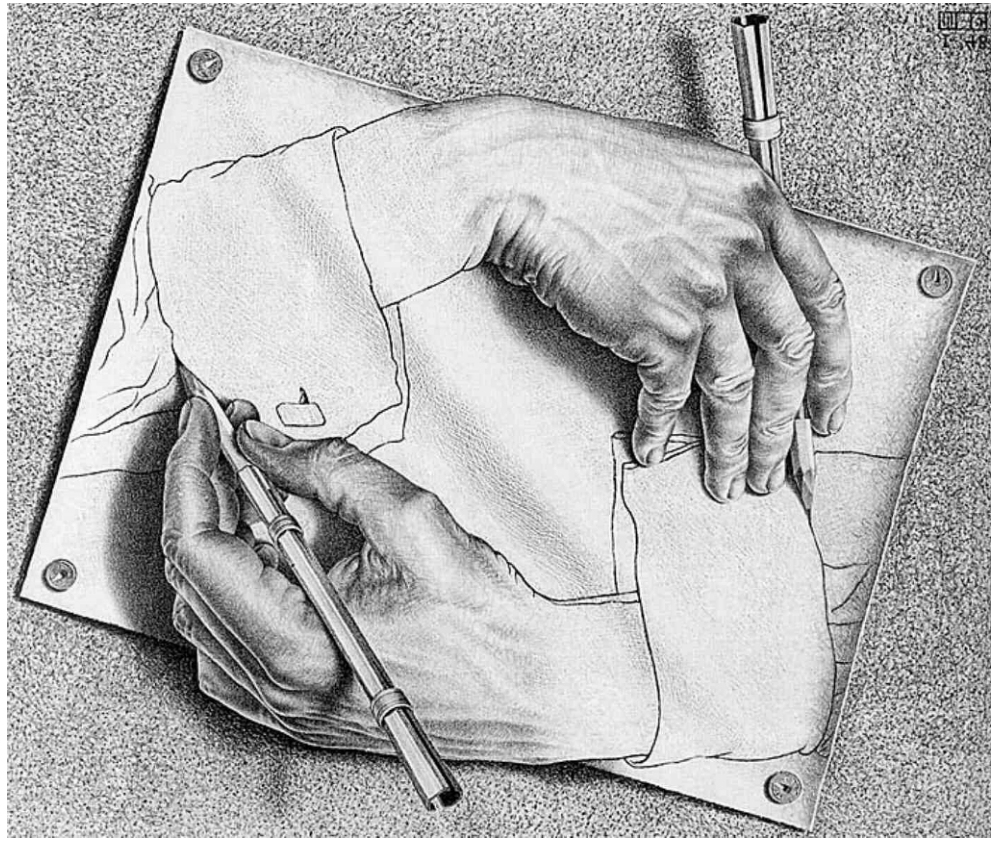


Rekursion in der Kunst

Maurits Cornelis Escher:

Drawing Hands, 1948

http://www.mcescher.com



Rekursion in der Kunst

- Alvin Lucier: I am sitting in a room.
- Technik: Sänger spricht Text in einem Raum, der aufgenommene Ton wird wiederholt im Raum abgespielt und gleichzeitig wieder aufgenommen, nach mehreren Durchläufen entsteht ein Musikstück, das die akustischen Charakteristika des Raumes widerspiegelt

initiale Aufnahme

nach 15 Durchläufen

nach 31 Durchläufen

http://www.archive.org/audio/audio-details-db.php?collection=opensource_audio&collectionid=residuum-i_am_sitting_in_a_room_mp3

Rekursion in der Kunst

- Alvin Lucier: I am sitting in a room.
- Technik: Sänger spricht Text in einem Raum, der aufgenommene Ton wird wiederholt im Raum abgespielt und gleichzeitig wieder aufgenommen, nach mehreren Durchläufen entsteht ein Musikstück, das die akustischen Charakteristika des Raumes widerspiegelt

initiale Aufnahme | nach 15 Durchläufen | nach 31 Durchläufen

http://www.archive.org/audio/audio-details-db.php?collection=opensource_audio&collectionid=residuum-i_am_sitting_in_a_room_mp3

Rekursion in der Kunst

- Alvin Lucier: I am sitting in a room.
- Technik: Sänger spricht Text in einem Raum, der aufgenommene Ton wird wiederholt im Raum abgespielt und gleichzeitig wieder aufgenommen, nach mehreren Durchläufen entsteht ein Musikstück, das die akustischen Charakteristika des Raumes widerspiegelt

initiale Aufnahme | nach 15 Durchläufen | nach 31 Durchläufen

http://www.archive.org/audio/audio-details-db.php?collection=opensource_audio&collectionid=residuum-i_am_sitting_in_a_room_mp3

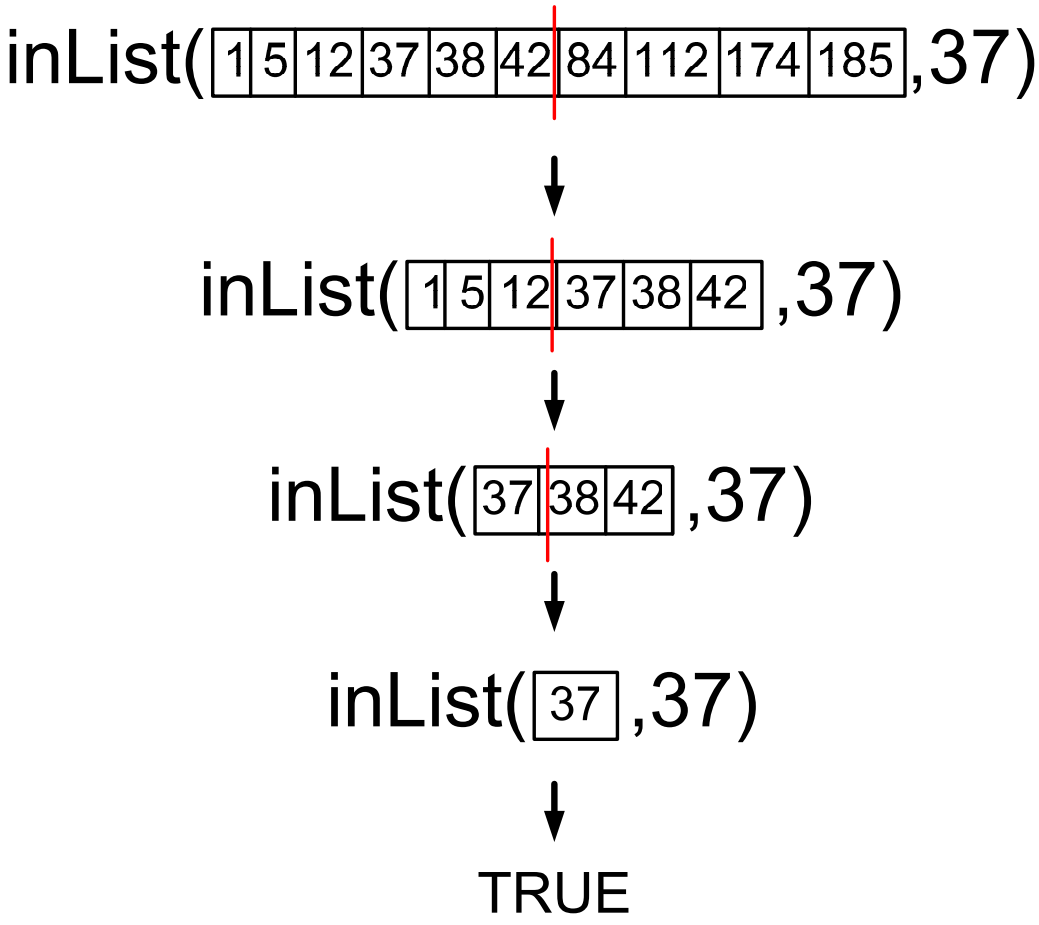
Beispiel: Suche eines Eintrags in sortierter Liste

- **Gegeben:** sortierte nicht-leere Liste (z.B. Telefonbuch), gesuchtes Element
- **Gesucht:** Ist Element in Liste oder nicht?
- Möglicher rekursiver Lösungsansatz:
  1. Hat die Liste nur ein Element, so vergleiche dieses Element mit dem gesuchten Element und gib das Ergebnis zurück.
  2. Besteht die Liste aus mehr als einem Element, so teile die Liste in zwei nicht-leere Listen l_1 und l_2 mit
     $$\forall\, x \in l_1, y \in l_2 : x \leq y$$
     Vergleiche das größte Element der ersten Hälfte mit dem gesuchten Element. Möglichkeiten:
     - gesuchtes Element ist größer: rufe den Algorithmus mit l_2 auf
     - sonst: rufe den Algorithmus mit l_1 auf
- Weitere Abwandlungen des Problems: Einfügen und Löschen von Elementen in sortierter Liste, Zusammenfügen zweier sortierter Listen

Terminierung

rekursiver Aufruf

Suche in sortierter Liste



Rekursive Programmierung: Lösung von Teilproblemen

- Grundgedanke der rekursiven Programmierung ist die intuitive Aufteilung des komplexen Problems in kleine, leicht lösbare Teilprobleme.
- Beispiel: Sortieren von Listen (nicht-lineare Rekursion)
  - Das (effektive) Sortieren einer Liste ist zunächst ein komplexer Algorithmus.
  - Eine leere Liste oder eine Liste mit einem Element ist per Definition sortiert.
  - Das Zusammenfügen zweier sortierter Listen kann sehr einfach implementiert werden.

  → **Idee:** Aufteilung der Liste in zwei Listen und Aufruf des Sortieralgorithmus für beide Listen, das Ergebnis sind zwei sortierte Listen, die einfach zusammengefügt werden können.

  Frage: wie macht man es?

Bereits bekannte rekursive Algorithmen

- Fakultät:

```
let rec fak n = match n with
        | 0 -> 1
        | n -> n*fak(n-1);;
```

- Summe:

```
let rec sum n = match n with
        | 0 -> 0
        | n -> n+sum(n-1);;
```

- Fibonacci-Zahlen

```
let rec fib n = match n with
        | 0 -> 0
        | 1 -> 1
        | n -> fib(n-1)+fib(n-2);;
```

Beispiel 1: Größter gemeinsamer Teiler (Euklid)

Wiederholung:

- Gegeben: Zwei Variable *a* und *b* aus Nat
- Gesucht *ggt(a,b):* größte Zahl, die sowohl Teiler von *a* als auch von *b* ist
- Algorithmus:
  - Unterscheidung zwischen folgenden Fällen:
  1. a mod b = 0: Das Ergebnis ist b.
  2. a>b: Rekursiver Aufruf von ggt(a mod b, b)
  3. a<b: Rekursiver Aufruf von ggt(a, b mod a)
- Hinweis: Vergleich des Algorithmus mit dem vorgeschlagenen Algorithmus auf Folie 43

Größter gemeinsamer Teiler (Euklid)

```
let rec ggt a b =
       if a mod b = 0 then b
       else ggt b (a mod b);;
```

oder

```
let rec ggt a b =
         let r = a mod b in
             if r = 0 then b else ggt b r;;
```

oder

```
let rec ggt a b =
         if (a mod b) = 0 then b
         else if (a>b) then ggt (a mod b) b
         else ggt a (b mod a);;
```

oder

```
let rec ggt a b =
         if a = 0 then b
         else if a >= b then ggt (a - b) b else ggt b a;;
```

Beispiel 2: verschränkte Rekursion (even, odd)

- **Aufgabe**: Bestimme, ob eine Liste gerade oder ungerade viele Elemente enthält
- **Gegeben**: Liste

Algorithmus:

Idee: Verwendung von zwei Funktionen *even* und *odd*

– Beide Funktionen prüfen, ob die Liste leer ist. Wenn ja, dann machen sie die entsprechende Ausgabe, ansonsten rufen sie die andere Funktion mit der um ein Element gekürzten Liste auf

Beispiel 2: verschränkte Rekursion (even,odd)

Code:

```
let rec odd list  = match list with
   | [] -> "odd"
   | head::tail -> even tail

and even list = match list with
   | [] -> "even"
   | head::tail -> odd tail;;

let evenOrOdd list = even list;;

let test1= evenOrOdd (21::41::227::[]);;
let test2= evenOrOdd (3::1::25::4::[]);;
```

Beispiel 3: Primfaktorenzerlegung

- **Aufgabe**: Algorithmus zur Zerlegung einer Zahl *n* in ihre Primfaktoren
- **Gegeben**: *n*
- **Gesucht:** Liste der Primfaktoren
- **Anwendungsgebiete**: z.B. Kryptographie

Algorithmus:

Starte die Suche beim Testen des Faktors f=2:

- Solange $n \geq f^2$ gilt:
  - Kann *n* durch den aktuellen Faktor geteilt werden
    - Ja → Rufe die Funktion mit $n=n/f$ und f rekursiv auf und hänge f an die Liste
    - Nein → Rufe die Funktion mit $n=n$ und $f=f+1$ rekursiv auf
- Falls n>1: Hänge *n* an die Liste

Beispiel 3: Primfaktorenzerlegung

Quellcode:

```
let rec prim_embedded n f = match n with
   | n when (n<f*f) -> n :: []
   | _ -> if (n mod f =0) then
              f::(prim_embedded (n/f) f)
      else
              prim_embedded n (f+1);;

let prim n = prim_embedded n 2;;
```

Beispiel 3: Primfaktorenzerlegung

Quellcode (als eingebettete Funktion):

```
let prim n =
   let rec prim_embedded n f = match n with
      | n when (n<f*f) -> n :: []
      | _ -> if (n mod f =0) then
                f::(prim_embedded (n/f) f)
       else
                prim_embedded n (f+1) in
   prim_embedded n 2;;
```

Beispiel 4: Zufallszahlen

- In der Informatik werden häufig zufällige Werte benötigt (z.B. Statistik, Kryptographie, numerische Mathematik (Monte-Carlo-Methoden), Analyse von Algorithmen)
- Fragestellung: Wie können Zufallszahlen (Reihen von Zufallszahlen) erzeugt werden?
- Typische Realisierung: Der multiplikative lineare Kongruenz-generator (MLKG)

  (http://www-user.tu-chemnitz.de/~jflo/Simulation/ZZ/mlkg.html)
- Formel:
  - s[k+1] := (a · s[k]) modulo m
  - Werte der Konstanten: z.B. m=2.147.483, $a=7^5=16.807$

Beispiel 4: Zufallszahlen

Code:

```
let nextSeed seed = UInt32.to_int32(((UInt32.of_int32(seed) *
   UInt32.of_int32(16807)) mod UInt32.of_int32(2147483647)));;

let rec random seed iter = match iter with
        | 0 -> []
        | n -> seed:: random (nextSeed seed) (iter - 1);;

let list= random 5 20;;

let rec listToString list = match list with
   |x::rest -> string_of_int(x)^","^listToString rest
   |_ -> "";;

do System.Console.WriteLine(listToString list);;
```

Beispiel 5: Binomialkoeffizienten

- Frage: Sollen Sie am Wochenende Lotto spielen?
- Der Binomialkoeffizient $\binom{n}{k}$ gibt die Anzahl der k-elementigen Teilmengen einer Menge von n Elementen an.

- Beispiel: $\binom{49}{6}$ gibt die Anzahl der Möglichkeiten an, die Sie beim Lotto haben (und damit ist der Kehrwert die Wahrscheinlichkeit, daß genau Ihre Zahlen gezogen werden).
- Eigenschaften des Binomialkoeffizienten:

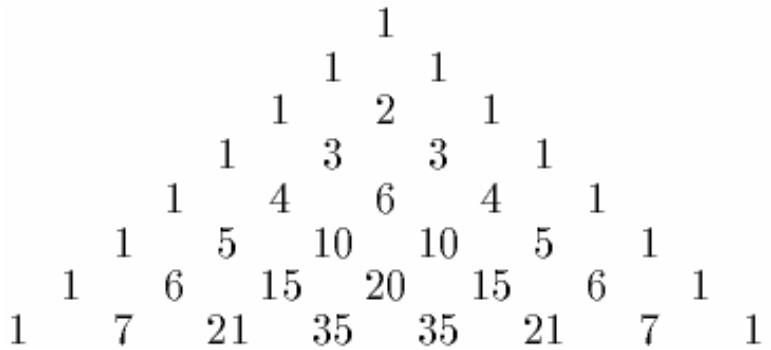
$$\binom{n}{k} = \frac{n!}{k!*(n-k)!}$$

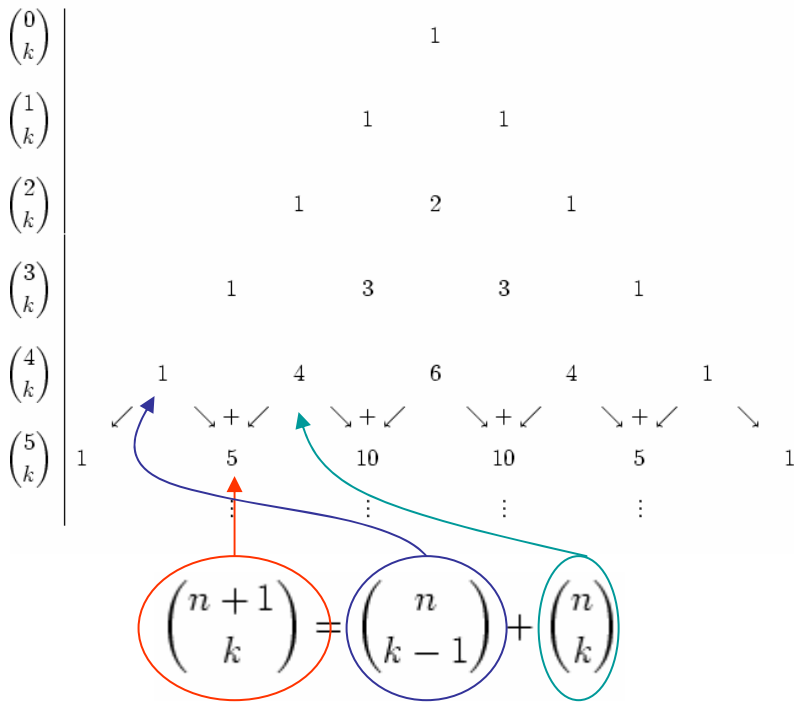
$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

Beispiel 5: Binomialkoeffizient

- Interpretation des Binomialkoeffizienten mit Hilfe des Pascalschen Dreiecks:





Beispiel 5: Binomialkoeffizient

Rekursive Berechung:

- Abbruchbedingungen: $\binom{n}{0} = \binom{n}{n} = 1$
- rekursiver Schritt: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Kombinatorischer Beweis:

Betrachtet man ein festes Element, so zerfallen alle k-elementigen Teilmengen in zwei Klassen:

- Die Klasse der Mengen, die das Element enthalten (für die restlichen Elemente bleiben noch $\binom{n-1}{k-1}$ Auswahlmöglichkeiten).
- Die Klasse der Mengen, die das Element nicht enthalten (somit verbleiben $\binom{n-1}{k}$ Auswahlmöglichkeiten).

Beispiel 5: Binomialkoeffizient

Code:

```
let rec binomial n k = match k with
   | 0 -> 1
   | k when (n=k) -> 1
   | _ -> binomial (n-1) (k-1) + binomial (n-1) k;;
```

Beispiel 6: Ackermannfunktion

- 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))

- Beliebter Anwendungszweck: Compiler-Benchmark

Beispiel 6: Ackermann-Funktion ohne/mit Pattern-Matching

```
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));;
# ack (2,8);;
- : int = 19
```

http://dada.perl.it/shootout/ackermann.html

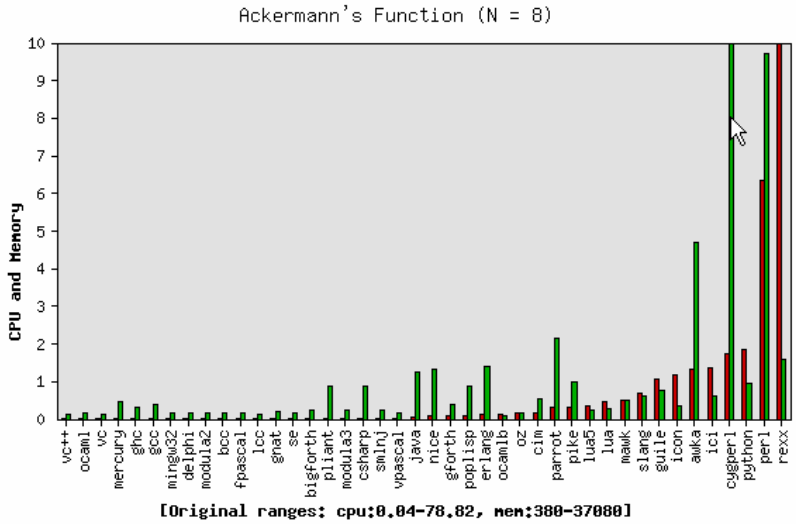
Erste Schritte Aktuelle Nachrichte...

Ackermann's Function

Back

[The Original Shootout] [NEWS] [FAQ] [Methodology] [Platform Details] [Acknowledgements] [Scorecard]

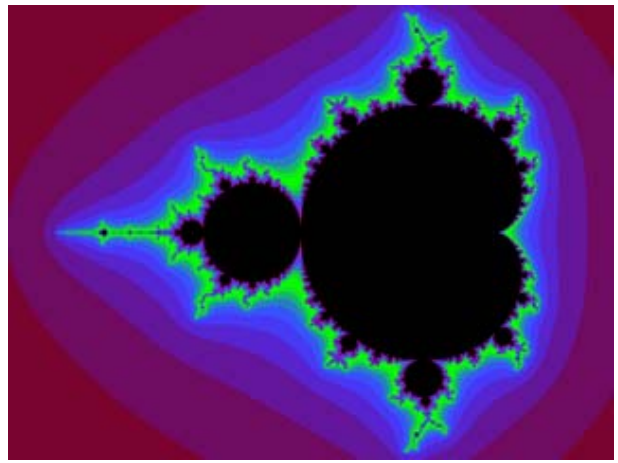
		[sort]	[sort]	
Source Code	CPU (sec)	Mem (KB)	Lines Code	Log
ocaml	0.04	664	9	log
vc++	0.04	540	13	log
vc	0.05	492	14	log
mercury	0.06	1728	36	log
gcc	0.07	1504	14	log
ghc	0.07	1224	9	log
mingw32	0.08	596	14	log
delphi	0.10	604	15	log
modula2	0.11	672	0	log
bcc	0.12	608	14	log
fpascal	0.13	556	18	log
lcc	0.13	548	14	log
gnat	0.14	792	0	log
se	0.14	592	30	log
bigforth	0.15	924	13	log
pliant	0.16	3228	14	log
csharp	0.18	3280	15	log
modula3	0.18	932	24	log
smlnj	0.22	940	20	log
vpascal	0.28	600	18	log
java	0.53	4628	11	log
nice	0.64	4940	0	log
gforth	0.67	1504	15	log
poplisp	0.78	3272	10	log
erlang	1.01	5268	10	log
ocamlb	1.09	380	9	log
oz	1.26	652	19	log
cim	1.32	2044	23	log
parrot	2.53	8036	35	log
pike	2.61	3620	9	log
lua5	2.71	840	11	log

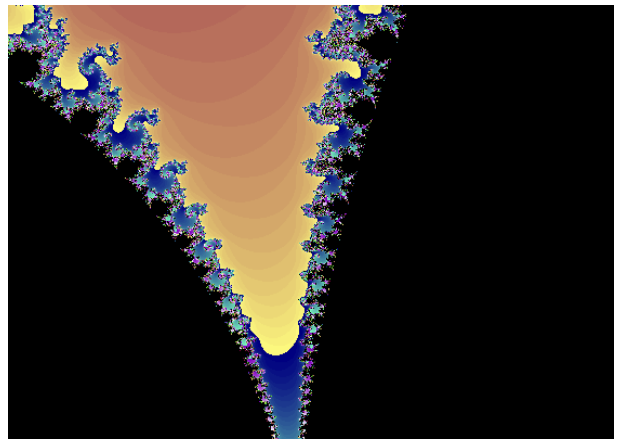


Beispiel 7: Mandelbrot-Menge

- Mandelbrot-Menge: Fraktal, definiert als die Menge der Punkte c $\in$ **C** (komplexe Zahlen), für die die iterierte Folge $z_0=0$

  $z_{n+1}=z_n^2+c$
  beschränkt bleibt ($z_i \in$ **C**); d.h. der *Betrag* der Folgenglieder für diesen Punkt wächst nicht über alle Grenzen
- *Iteration*: wiederholte Anwendung der gleichen Rechenvorschrift (auf den Ergebnissen des jeweils vorigen Rechenschritts)
- Anwendungsgebiet: Benchmarks für Graphikprozessoren
- 1905 von Pierre Fatou mathematisch erarbeitet, 1980 von Benoît Mandelbrot erstmals graphisch auf dem Computer dargestellt
- Für die Farbzuordnung nimmt man den Grad der Divergenz: schwarze Punkte konvergieren; die Anzahl der Schritte, nach denen der Betrag des Folgeglieds über einem Punkt einen Schwellwert überschreitet, wird einer Farbe zugeordnet.





Beispiel 7: Mandelbrot-Menge

Code:

```
type complex = int*int;;

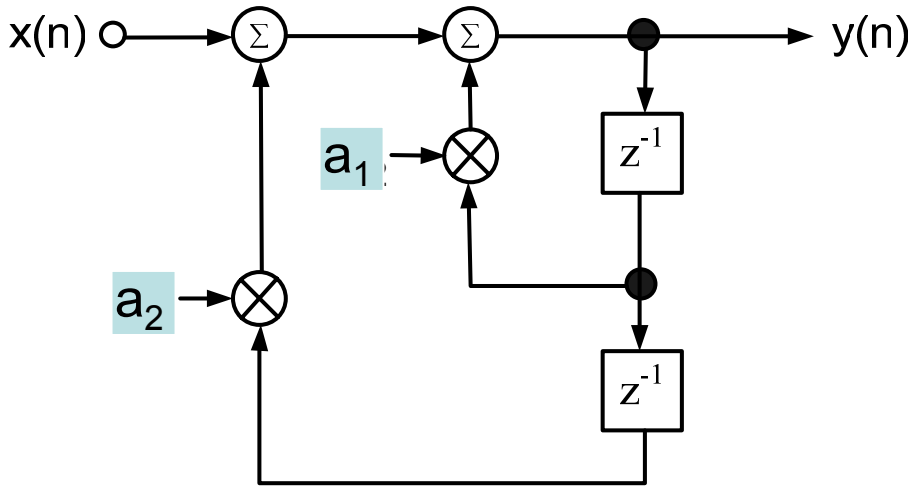
let addComplex a b = match (a,b) with
   | ((real_a,im_a),(real_b,im_b)) ->
       ((real_a+real_b),(im_a+im_b));;

let multComplex a b =  match (a,b) with
   | ((real_a,im_a),(real_b,im_b)) ->
       (((real_a*real_b)-
       (im_a*im_b)),((real_a*im_b)+(im_a*real_b)));;

let test_mandelbrot = ?
```

Beispiel 8: IIR-Filter

- Anwendungsgebiet:
  Digitale Signalverarbeitung
  (z.B. CD-Player)
- IIR: Infinite Impulse
  Response Filter
- Rekursionsgleichung:
  $y(n)=x(n)+a_1y(n-1)+a_2y(n-2)$
  $y(1)=x(1)+ a_1y(0)$
  $y(0)=x(0)$



Schaltbild

Beispiel 8: IIR-Filter

Code:

```
let rec iir x a1 a2 = match x with
   | x::[] -> x::[]
   | head::tail -> match iir tail a1 a2 with
        | x0::[] -> (head+(a1*x0))::x0::[]
        | x1::x2::rest -> (head+(a1*x1)+(a2*x2))::x1::x2::rest
        | _ -> []
   | _ -> [];;
```

```
# let test= iir (5::2::4::1::[]) 2 3;;
val test : int list = [57; 17; 6; 1]
```

Informatik in a nutshell

Exkurs: Induktion und Terminierung

Induktion

- **Definition:** Induktion bzw. induktives Schließen bezeichnet in der Logik und den Naturwissenschaften das Schließen „vom Besonderen auf das Allgemeine“ zum Zweck des Erkenntnisgewinns.
- *Induktion* und *Rekursion* sind somit stark miteinander verwandt: bei der Rekursion wird das Problem durch Abbildung auf Teilprobleme gelöst, während bei der Induktion der umgekehrte Weg vollzogen wird.
- Vollständige Induktion wird in der Mathematik angewandt, um Aussagen nicht nur für endliche Mengen, sondern auch für unendliche Mengen (z.B. die Menge der natürlichen Zahlen) treffen zu können.
- Prinzip: Zunächst wird gezeigt, dass die Aussage A für ein festes n (in der Regel n=0 bzw. n=1) gilt (**Induktionsanfang**). Nun muß noch gezeigt werden, daß wenn für alle Zahlen $x \leq n$ die Aussage $A(x)$ gilt, die Aussage auch für $A(n+1)$ gilt.

Induktive Definition

- Peano-Axiome zur induktiven Definition der Menge *NAT* der natürlichen Zahlen (s(*x*) bezeichnet die Nachfolgezahl von *x*):

P1 $0 \in NAT$.

P2 Wenn $n \in NAT$, dann auch $s(n) \in NAT$.

P3 Es gilt $0 \neq s(n)$ für alle n.

P4 Es ist $m = n$, falls $s(m) = s(n)$, d.h. s ist injektiv.

P5 Durch P1 und P2 sind alle Elemente von NAT gegeben.

Entsprechende Definition eines Datentyps

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

- Anwendung:

```
# 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))))))))))
```

Vollständige Induktion: Summe

Zu **beweisende** Aussage:

$$\forall n \in NAT : \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Induktionsanfang: n=0

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

Induktionsschritt

Induktionsschritt: $n \rightarrow n+1$:

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

Induktionsvoraussetzung

Induktion

Die Induktion zum Beweis einer Annahme besteht immer aus:

1. Induktionsanfang
2. Induktionsvoraussetzung
3. Induktionsschritt
4. Induktionsschluss

Vollständige Induktion: Summe der Quadratzahlen

- Zu **beweisende** Aussage: $\forall n \in NAT : \sum_{i=0}^{n} i^2 = \frac{n*(n+1)*(2n+1)}{6}$

- **Induktionsanfang**: n=0 $0 = \sum_{i=0}^{0} i = \frac{0*(0+1)*(2*0+1)}{6}$

- **Induktionsschritt**: $n \rightarrow n+1$:

$$\sum_{i=0}^{n+1} i^2 = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6} = \frac{(n+1)(n+2)(2n+3)}{6} =$$

$$= \frac{(n^2+3n+2)(2n+3)}{6} = \frac{2n^3+9n^2+13n+6}{6} =$$

$$= (n^2+2n+1) + \frac{2n^3+3n^2+1n}{6} = (n+1)^2 + \frac{n(2n^2+3n+1)}{6} = (n+1)^2 + \frac{n(n+1)(2n+1)}{6} =$$

$$= (n+1)^2 + \sum_{i=0}^{n} i^2$$

Vollständige Induktion: Binomialkoeffizient

- Zu **beweisende** Aussage: $\forall n \in NAT : \sum_{k=0}^{n} \binom{a+k}{k} = \binom{a+n+1}{n}$
- **Induktionsanfang**: n=0 $1 = \binom{a+0}{0} = \binom{a+0+1}{0} = 1$
- **Induktionsschritt**: $n \rightarrow n+1$:

$$\sum_{k=0}^{n+1} \binom{a+k}{k} = \sum_{k=0}^{n} \binom{a+k}{k} + \binom{a+n+1}{n+1} = \binom{a+n+1}{n} + \binom{a+n+1}{n+1} = \binom{a+n+2}{n+1}$$

Vollständige Induktion: Fibonacci-Zahlen

- Zu **beweisende** Aussage: $\forall n \in NAT: \; fib(n) \leq \left(\frac{7}{4}\right)^n$

- **Induktionsanfang**: n=0 und n=1 (!!!)

$$fib(0) = 0 \leq \left(\frac{7}{4}\right)^0 = 1$$

$$fib(1) = 1 \leq \left(\frac{7}{4}\right)^1 = 1{,}75$$

- **Induktionsschritt**: $n \rightarrow n+1$:

$$fib(n+1) = fib(n) + fib(n-1) \leq \left(\frac{7}{4}\right)^n + \left(\frac{7}{4}\right)^{n-1} = \left(1+\frac{4}{7}\right) * \left(\frac{7}{4}\right)^n \leq \left(\frac{7}{4}\right) * \left(\frac{7}{4}\right)^n \leq \left(\frac{7}{4}\right)^{n+1}$$

Vollständige Induktion: Binomischer Satz

- Zu **beweisende** Aussage: $\forall n \in NAT : (a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} * b^k$
- **Induktionsanfang**: $n=0$ $(a+b)^0 = 1 = \binom{0}{0} a^0 * b^0$
- **Induktionsschritt**: $n \rightarrow n+1$:

$$(a+b)^{n+1} = \sum_{k=0}^{n+1} \binom{n+1}{k} a^{n+1-k} * b^k = a^{n+1} + \sum_{k=1}^{n} \binom{n+1}{k} a^{n+1-k} * b^k + b^{n+1} = a^{n+1} + \sum_{k=1}^{n} \left[\binom{n}{k} + \binom{n}{k-1} \right] a^{n+1-k} * b^k + b^{n+1}$$

$$\binom{n}{0} a^{n+1} * b^0 + \sum_{k=1}^{n} \binom{n}{k} a^{n+1-k} * b^k + \sum_{k=1}^{n} \binom{n}{k-1} a^{n-k} * b^{k+1} + \binom{n}{0} a^{n-0} b^{k+1} =$$

$$= \sum_{k=0}^{n} \binom{n}{k} a^{n+1-k} * b^k + \sum_{k=0}^{n} \binom{n}{k-1} a^{n-k} * b^{k+1} = a * \sum_{k=0}^{n} \binom{n}{k} a^{n-k} * b^k + b * \sum_{k=0}^{n} \binom{n}{k} a^{n-k} * b^k = (a+b) * \sum_{k=0}^{n} \binom{n}{k} a^{n-k} * b^k =$$

$$(a+b) * (a+b)^n = (a+b)^{n+1}$$

Wichtige Anwendung von Induktion: Terminierung

- Definition Terminierung: Der Algorithmus muss für alle zulässigen Eingabeparametern in endlich vielen Schritten eine Lösung finden.
- Zum Beweis der Terminierung von rekursiven Funktionen kann die Induktion verwendet werden.
- Zum Beweis werden so genannte *Abstiegsfunktionen* verwendet.

Beweis der Terminierung durch Abstiegsfunktionen

- Eine Abstiegsfunktion bildet die Menge der Eingabeparameter einer vorgelegten Funktion auf eine natürliche Zahl ab.
- Die Abstiegsfunktion wird so gewählt, daß sie eine obere Grenze für die Anzahl der rekursiven Aufrufe (**Aufruftiefe**) angibt.
- Es muß gezeigt werden, daß die Funktion für alle Aufrufe mit Parametern, die auf einen Minimalwert (typischerweise 0 oder 1) abgebildet werden, sofort terminieren.
- Kann nun (für die gewählte Abstiegsfunktion) bewiesen werden, daß für jeden Aufruf der Wert der Abstiegsfunktion abnimmt und durch den Minimalwert nach unten begrenzt ist, so ist damit auch die Terminierung bewiesen.

Beweis der Terminierung durch Induktion

Vorgehensweise:

1. **Induktionsanfang**: Zu Beginn muss die Terminierung für das Minimum der Abstiegsfunktion bei Anwendung auf die zulässigen Argumente (typischerweise 0 oder 1) nachgewiesen werden.
2. **Induktionsannahme**: Wir gehen davon aus, daß wir die Terminierung für alle Aufrufe mit Argumenten, für die die Abstiegsfunktion einen Wert kleiner gleich *n* annimmt, nachgewiesen haben.
3. **Induktionsschritt**: Wir betrachten nun die Aufrufe mit Argumenten, für die die Abstiegsfunktion den Wert *n+1* annimmt. Wir müssen nachweisen, daß innerhalb dieser Aufrufe nur rekursive Aufrufe mit solchen Argumenten *arg'* auftreten, so daß gilt: $f(arg') \leq n$. Laut Induktionsannahme terminieren aber diese Aufrufe.
4. **Induktionsschluss**: Die Funktion terminiert auch für Aufrufe mit Argumenten bei denen die Abstiegsfunktion den Wert *n+1* annimmt. Somit terminiert die Funktion für alle Aufrufe.

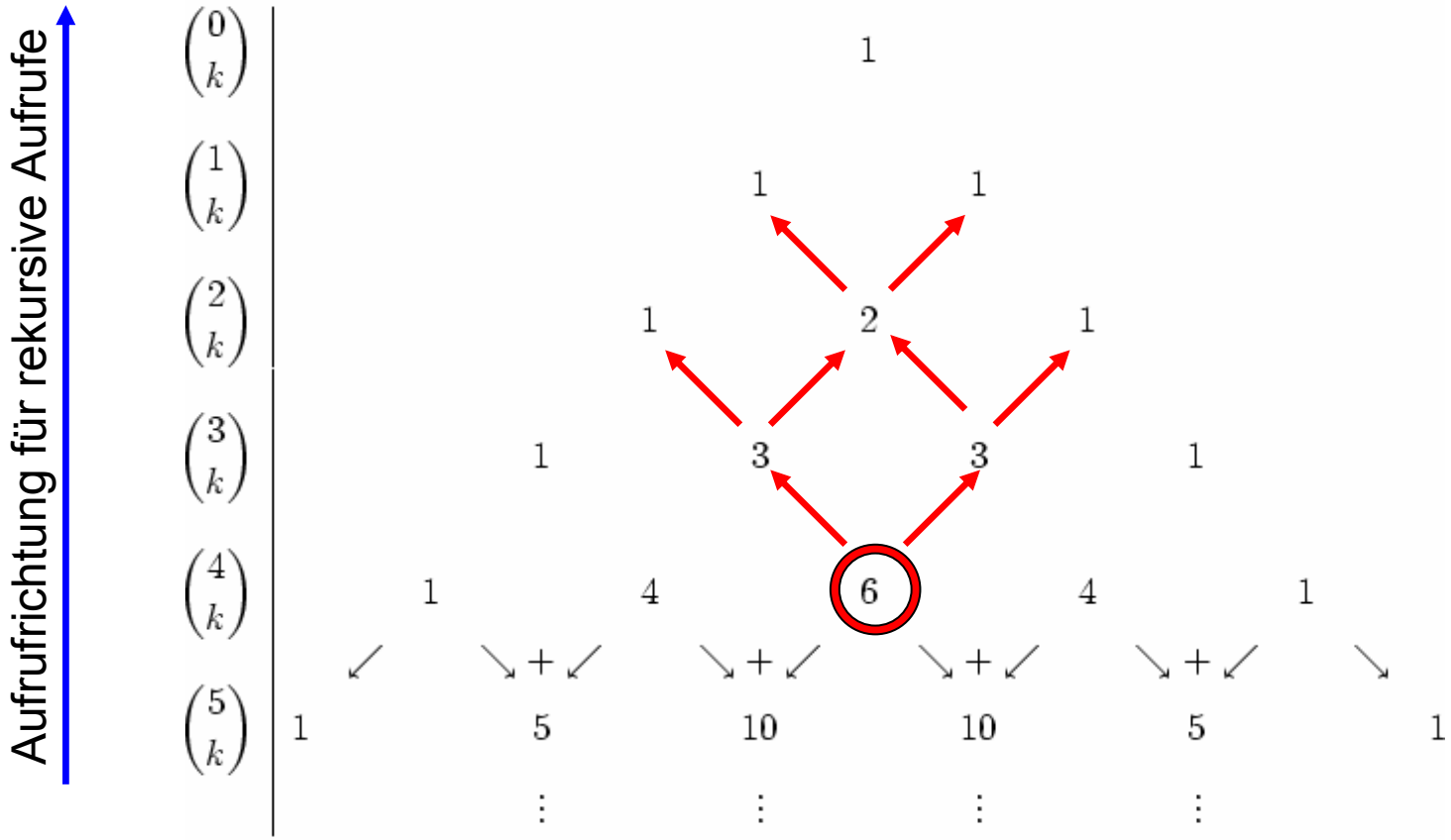
Triviales Beispiel: Terminierung der Summe

```
let rec sum n = match n with
        | 0 -> 0
        | n -> n + sum(n-1);;
```

- Abstiegsfunktion f(*n*) = *n*
- Beweis der Terminierung:
  - Zu zeigen: wird die Summen-Funktion mit dem Argument *arg* aufgerufen, so terminiert diese sofort, oder ab es gilt für sämtliche rekursive Unteraufrufe mit den Argumenten *arg'*:

    f(*arg*) > f(*arg'*)
  - Induktionsanfang: *n* = 0 → *arg=0* (f(0) = 0)
    - Der Algorithmus terminiert, da dies die Abbruchbedingung ist.
  - Induktionsschritt: *n* → *n*+1
    - Die Abstiegsfunktion *f(arg)* nimmt den Wert *n+1* an, falls arg=*n+1* (**trivial**)
    - Die Funktion ruft sich rekursiv mit *n* auf, f(*n*+1*)* > f(*n*)

Beispiel Terminierung: Binomialkoeffizienten



Beispiel Terminierung: Binomialkoeffizienten

```
let rec binomial n k = match k with
        | 0 -> 1
        | k when (n = k) -> 1
        | _ -> binomial (n-1) (k-1) +
               binomial (n-1) k;;
```

- Abstiegsfunktion f(*n,k*) = *n*
- Beweis der Terminierung:
  - Induktionsanfang: *n*=0 (f(0)=0)
    - Der Algorithmus terminiert, da dies die Abbruchbedingung ist.
  - Induktionsschritt: *n* → *n+1*
    - Die Abstiegsfunktion f(*arg*) nimmt den Wert *n+1* an, falls *arg*=(*n*+1,*k)* (**trivial**)
    - Im Algorithmus erfolgen zwei rekursive Unteraufrufe mit den Argumenten (*n,k*-1) sowie (*n,k*)

      ⇒ Es gilt wieder f(*n*+1,*k*) > f(*n,k-1*) = f(*n,k*)

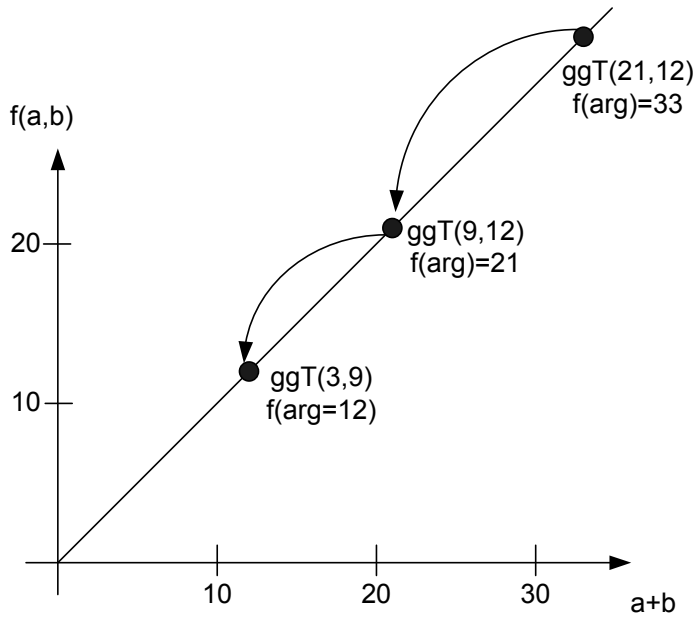
Beispiel Terminierung: ggT

- Abstiegsfunktion f(*a,b*) = *a* + *b*
- Beweis der Terminierung:
  - Induktionsanfang: f(*a*,*b*) = 0
    - Da der Algorithmus nur für positive Zahlen definiert ist, gilt *a*=*b*=0. Der Algorithmus terminiert, da dies die Abbruchbedingung ist.
  - Induktionsschritt: $n \rightarrow n+1$
    - Im Algorithmus erfolgt ein rekursiver Unteraufruf, entweder mit (*a*, *b* mod *a*) falls *b* > *a* oder (*a* mod *b*, *b*) falls *a* > *b*
    - Aus f(*arg*)=*n*+1 folgt *a*+*b*=*n*+1

    ⇒ Es gilt *n*+1=f(*arg*)=f(*a, b*) > f(*a, b* mod *a*) = *a*+*b* mod *a*≤*n,* falls *b*>*a*

    ⇒ Es gilt *n*+1=*a*+*b*=f(*a,b*)>f(*a* mod *b,b*)=*a* mod *b*+*b* ≤ *n,* falls *a*>*b*

```
let rec ggt a b =
        if (a mod b) = 0 then b
        else if (a>b) then ggt (a mod b) b
        else ggt a (b mod a);;
```



Beispiel Terminierung: Ackermannfunktion

- Ackermannfunktion:

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

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

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

- Hilfsmittel: Ordnung über NAT x NAT als Abstiegsfunktion:

  $(0,0) < (0,1) < (0,2) < (0,3) < \ldots < (1,0) < (1,1) < \ldots$

- Beweis per geschachtelter Induktion:
  - über *m* (äußere Induktion)
  - über *n* bei festen m (innere Induktion)

Beispiel: Terminierung Ackermannfunktion

- Induktionsanfang: m=0

  A(0,n) terminiert (Abbruchbedingung)

- Induktionsschritt:
  - Induktionsvoraussetzung: Die Ackermannfunktion $A(k,n)$ terminiert für $0 \leq k < m$ und beliebiges n. (**IV1**)
  - Innere Induktion:
    - Induktionsanfang: n=0

      A(m,0) terminiert, da der rekursive Unteraufruf mit A(m-1,1) terminiert
    - Induktionsvoraussetzung: Die Ackermannfunktion A(m,l) terminiert für festes m und $l < n$ (**IV2**)
    - Induktionsschluss auf n: $A(m,n)$ terminiert, da sowohl $x=A(m,n\text{-}1)$ (**IV2**) als auch $A(m\text{-}1,x)$ (**IV2**) terminiert.

(Funktionale) Programmierung

Funktion

- **Definition**: Gegeben sei eine endliche Zahl von Mengen $D_1,\ldots D_n$ sowie die Menge W. Eine Funktion f ist eine Abbildung, die einem Element des kartesischen Produktes $D_1 x D_2 x \ldots D_n$ eindeutig ein Element der Menge W zuordnet.
- Die Mengen $D_1,\ldots D_n$ beschränken sich dabei nicht auf Zahlen. In der Informatik können verschiedenste Daten (Wahrheitswerte, Zahlen, Zeichen oder komplexere Datenstrukturen) verwendet werden.
- Funktionale Abläufe sind dabei nicht auf die Anwendung einer einzigen Funktion beschränkt. Vielmehr können Funktionsresultate auch als Argumente für weitere Funktionen verwendet werden.

Bemerkungen zur funktionalen Programmierung

- 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 a functional style. (www.cs.nott.ac.uk/~gmh/faq.html)

- Damit gilt:

  Funktionale Programmierung = Definition von Funktionen
  (Funktionsdefinition; Abstraktion)

  Funktionale Programmausführung = Auswerten von Ausdrücken
  (Funktionsanwendung; Applikation)

Merkmale der funktionalen Programmierung I

1. Funktionsdefinitionen können ihrerseits auf weitere Funktionen zurückgreifen

```
result1= function1 (function2(function3(),function4())));
```

Beispiel:

```
result=getNumberOfPersonsInList(getListOfInfo1StudentsWS2005());
```

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.

   Beispiel:

   `getTime()` ist keine zulässige Funktion in der funktionalen Programmierung

Merkmale der funktionalen Programmierung II

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

   Beispiel:

   Konstante: `numberOfSecondsPerMinute`

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

Mermale der funktionalen Programmierung III

Konsequenzen aus 4.:

- Das 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 Kontext 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

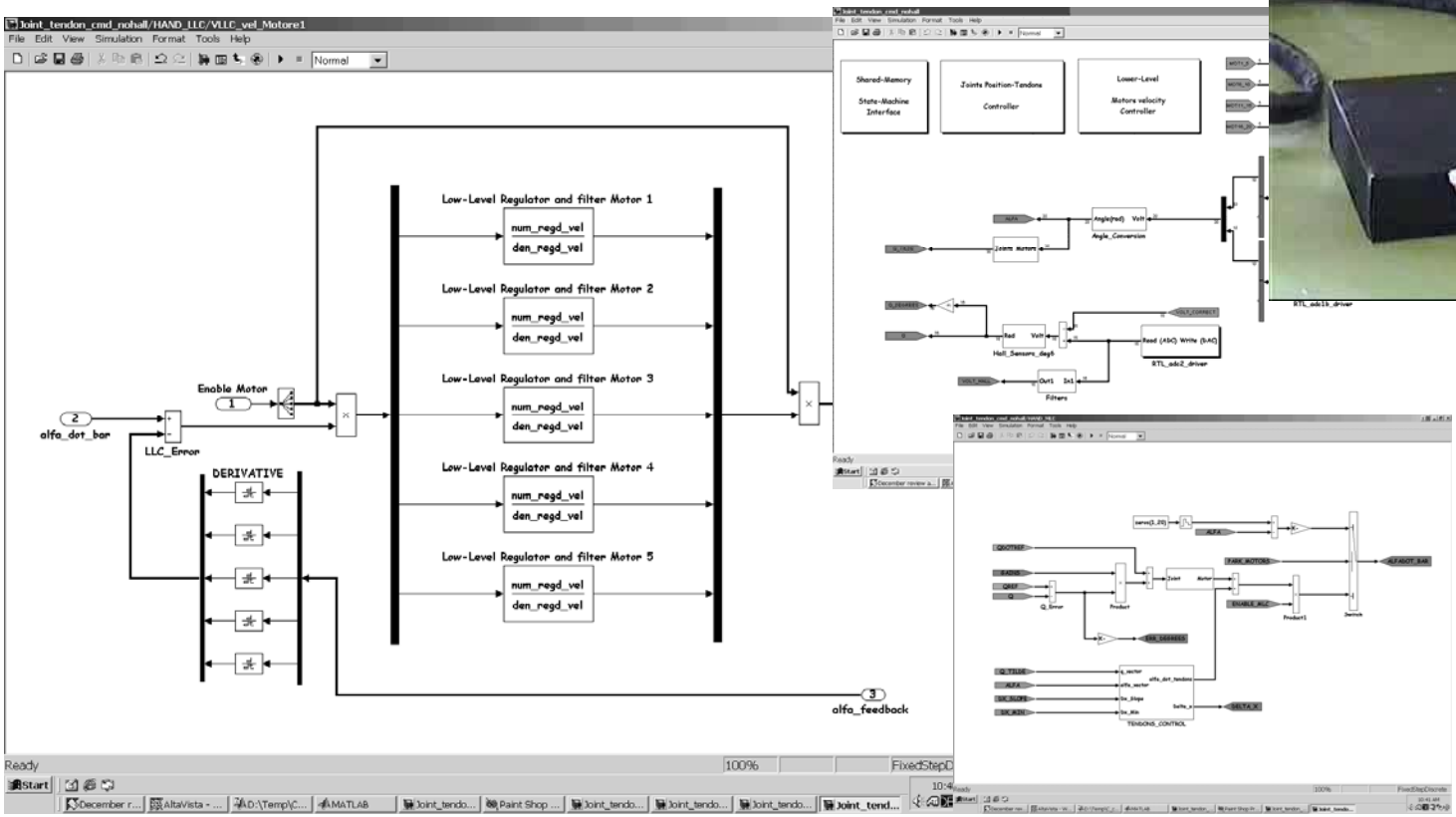
Vorteile der funktionalen Programmierung

- Was macht die funktionale Programmierung interessant?
  - Die Grundprinzipien sind sehr einfach
  - Funktionale Programme sind klar strukturiert (keine Seiteneffekte)
  - Die Ergebnisse sind nur von den Eingabeparametern abhängig; identische Parameter führen zu identischen Resultaten
  - Bei der Modellierung werden zunächst nur die Beziehungen zwischen den beteiligten Funktionen modelliert, d.h. die Datenflüsse zwischen den einzelnen Funktionen (Black-Box)
  - Bei den geeigneten Problemklassen sehr kurzer Weg vom Problem zum Programm

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

Vorteile der funktionalen Programmierung

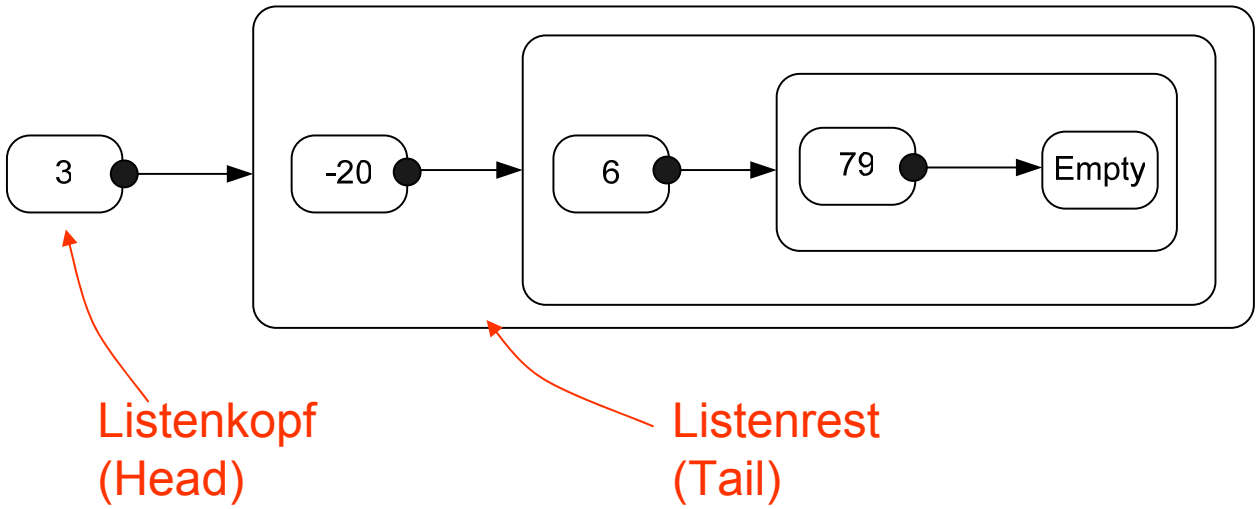
Direkte Umsetzbarkeit des black-box- bzw. Datenflussdenkens z.B. in den Ingenieurswissenschaften



Zentrale Datenstrukturen

Zentrale Datenstrukturen: Listen

- Sehr häufig werden Datenstrukturen benötigt, die mehrere Elemente des gleichen Datentyps speichern können. Listen deshalb zentraler Daten typ
- Listen sind Datentypen, die eine sich ändernde Zahl von Elementen eines gleichen Datentyps beinhalten kann.
- Listen sind rekursive Datenstrukturen:



Wiederholung: Realisierung in OCaml

Deklaration des polymorphen Datentyps Liste in OCaml

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

Deklaration der Liste von der letzten Folie:

```
let x=Prepend(3,Prepend(-
  20,Prepend(6,Prepend(79,Empty))));;
```

Pattern-Matching auf Listen

- Das Pattern-Matching in OCaml bietet die Möglichkeit, eine Liste auf Leerheit zu überprüfen und erlaubt zudem die Aufspaltung der Liste.
- Beispiel: Summe aller Elemente einer Liste von Ganzzahlen:

```
let rec sum list = match list with
   | Empty -> 0
   | Prepend(hd,tail) -> hd + sum tail;;

sum x;;
```

Operationen auf Listen

- Im Zusammenhang mit Datentypen werden Operationen zur Manipulation von Objekten dieses Datentyps angeboten
- Für Listen werden typischerweise folgende Operationen angeboten:
  - `isEmpty` *list*: Test, ob Liste leer ist
  - `length` *list*: Funktion zur Ermittlung der Anzahl der Elemente
  - `head` *list*: Funktion, die das erste Element einer Liste zurückgibt
  - `tail` *list*: Funktion, die von der Liste das erste Element entfernt
  - `isElement` *list_elem*: Funktion, die ermittelt ob ein Element in der Liste enthalten ist
  - `insert` *list_elem*: Funktion zum Einfügen eines Elements
  - `add` *list_elem*: Funktion zum Einfügen eines Elementes an den Kopf der Liste, häufige Syntax: *elem::list*
  - `delete` *list_elem*: Funktion zum Löschen eines Elements aus der Liste
  - `merge` *list1 list2*: Funktion zum Zusammenfügen zweier Listen, häufige Syntax: *list1@list2*

Spezialformen von Listen

Oft werden auch diverse Spezialformen von Listen verwendet, die wir teilweise schon kennen gelernt haben:

- Keller (Stack)/LIFO: Es kann jeweils nur auf das zuletzt eingefügte Element zugegriffen werden, Operationen: isEmpty, push (wie insert), top (wie head), pop (wie tail)
- Warteschlange/Queue/Fifo: Es kann immer nur auf das zuerst eingefügte Element zugegriffen werden, Operationen: isEmpty, insert (fügt Element hinten in Liste an), get (wie head)
- Sortierte Listen:
  - aufsteigend sortiert
  - absteigend sortiert

Gleich: Algorithmen zur Sortierung von Listen

Vorher Exkurs:
Algorithmischer Aufwand & Landausymbole

Bewertung von Algorithmen

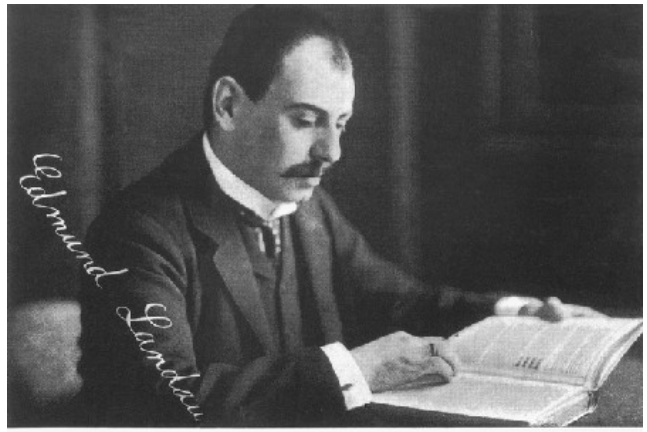
- Zur Bewertung von Algorithmen werden Abschätzungen zur Laufzeit und zum Speicherplatzverhalten benötigt
- Typischerweise ist der Aufwand abhängig von den Eingabeparametern
- Die Landau-Notation ermöglicht die Bestimmung des Zusammenhangs zwischen der Länge der Eingabe und der Laufzeit- bzw. Speicherplatzaufwand

Begriffe

- *Problemumfang/Problemgröße n*: Meist der Wert oder die Länge der Eingabe
- *Rechenaufwand*: Abstraktion von konkreten Rechnzeiten zu Anzahl der durchzuführenden Elementaroperationen (z.B. Addition, Vergleich)
- *Speicherbedarf*: Berechnung der Größe des zur Ausführung des Algorithmus benötigten Speicherplatzes
- Unterscheidung zwischen günstigstem (*best case*), durchschnittlichem (*average*) und schlechtestem Fall (*worst case*)
- Komplexität des Algorithmus: tatsächlich benötigter Aufwand für einen konkreten Algorithmus
- Komplexität des Problems: minimale Komplexität eines Algorithmus zur Lösung des Problems

Landau-Symbole

- Die Landau-Symbole erlauben Aussagen der Form: Die Funktion *f* wächst nicht schneller als die Funktion *g*.
- Informatiker sind vor allem an dem asymptotischen Verhalten der Komplexität für $n \to \infty$ interessiert, wobei *n* die Problemgröße (z.B. die Anzahl der Elemente einer Liste) repräsentiert.



Edmund Landau

(*1877, † 1938)

Landau-Symbole („*O*-Notation“)

$f(x) \in O(g(x)) \Rightarrow \exists c > 0 \exists x_0 \forall x > x_0 : |f(x)| \leq c \cdot |g(x)|$ f ist kleiner gleich g

$f(x) \in o(g(x)) \Rightarrow \exists c > 0 \forall x_0 \forall x > x_0 : |f(x)| < c \cdot |g(x)|$ f ist echt kleiner g

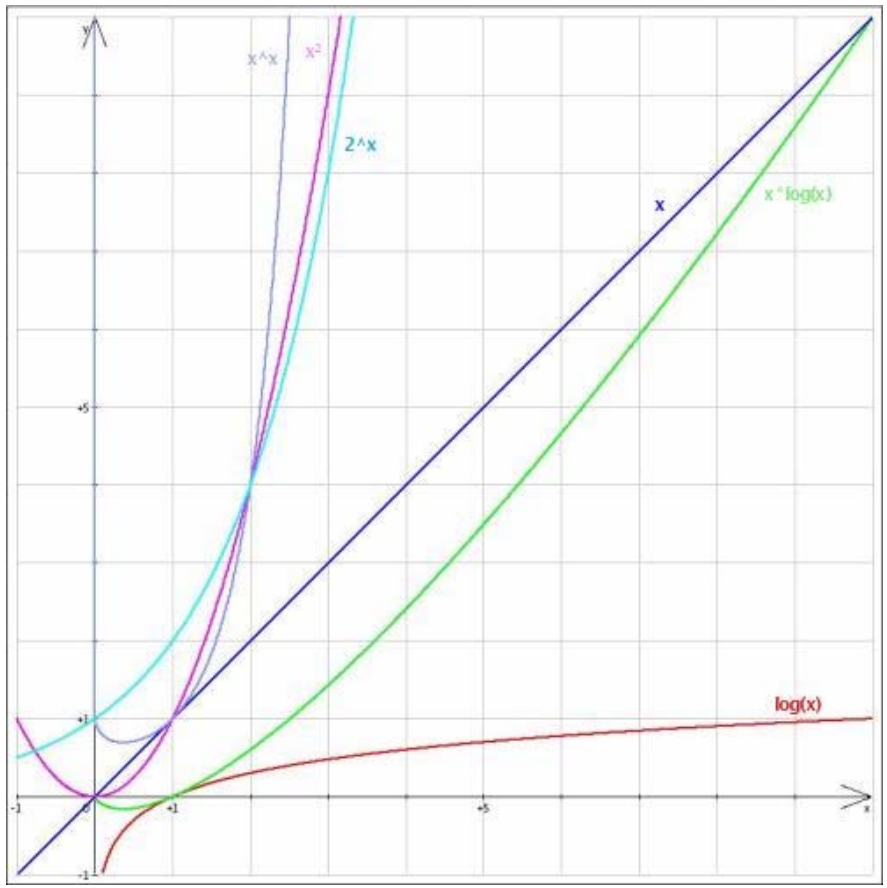
$f(x) \in \Omega(g(x)) \Rightarrow \exists c > 0 \exists x_0 \forall x > x_0 : |f(x)| \geq c \cdot |g(x)|$ f ist größer gleich g

$f(x) \in \omega(g(x)) \Rightarrow \exists c > 0 \forall x_0 \forall x > x_0 : |f(x)| > c \cdot |g(x)|$ f ist echt größer g

Mit Hilfe dieser Notationen können wir nun eine Aussage treffen, wie effizient ein Algorithmus ist.

- $O(1)$: Solche Algorithmen liefern unabhängig von der Problemgröße in konstanter Zeit ein Ergebnis, Beispiel: Entfernen des ersten Elementes einer Liste
- $O(\log n)$: Die benötigte Zeit ist proportional zum Logarithmus der Problemgröße, Beispiel: Suche in einer sortierten Liste.
- $O(n)$: Der Aufwand für einen Algorithmus ist proportional zur Problemgröße, Beispiel Suche in einer unsortierten Liste.
- $O(n * \log n)$: Beispiel: vergleichsbasiertes Sortieren einer Liste (siehe später)
- $O(n^2)$: Beispiel: Multiplikation einer Matrix mit einem Vektor
- $O(n^3)$: Beispiel: Multiplikation von Matrizen
- $O(2^n)$: Bestimmung aller möglichen Teilmengen einer Menge

Typisches Wachstumsverhalten



O-Notation: Konstante *c*

- Bisher haben wir nur die Komplexitätsklassen ($O(1)$,$O(\log n)$, $O(n)$,…) betrachtet und eine Größe ignoriert: die Konstante c.
- Die Konstante c erlaubt uns den Vergleich von verschiedenen Funktionen und ist im asymptotischen Verhalten für $n \rightarrow \infty$ vernachlässigbar.
- Dies gilt jedoch nicht für kleine Problemgrößen! Unter Umständen kann es deshalb sinnvoll sein, einen Algorithmus zu verwenden der eigentlich einer höheren Komplexitätsklasse angehört, aber ein wesentlich kleineres c besitzt.

Beispiel: Aufwandabschätzung für Fakultät

```
let rec fak n = match n with
   | 0 -> 1
   | n -> n*fak(n-1);;
```

Relevante Elementaroperationen: Vergleich, Multiplikation

Unterscheidung zwischen zwei Fällen:

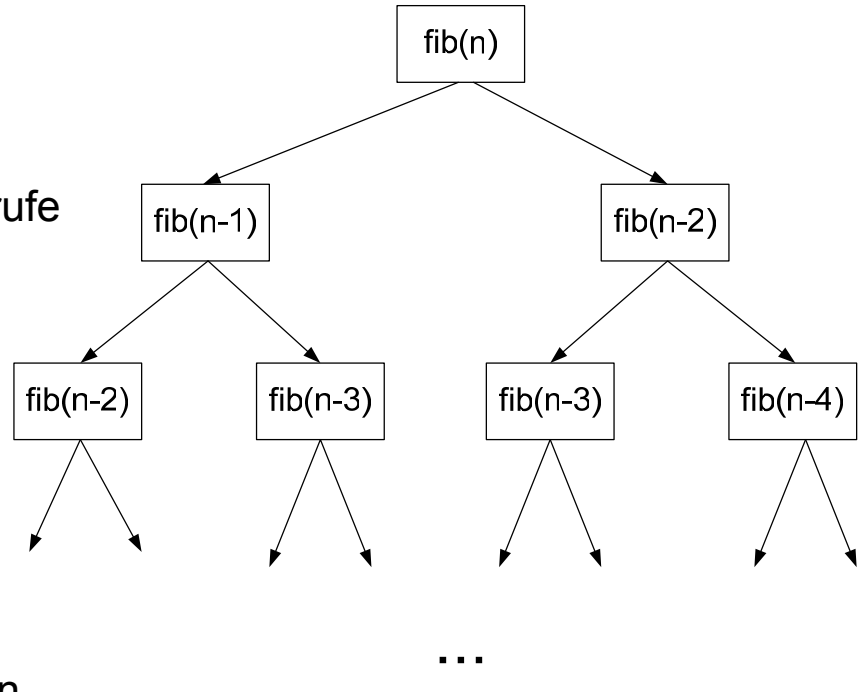
- 1.Fall: n=0: Eine Elementaroperation (Vergleich) nötig
- 2.Fall: n>0: Ein Vergleich und eine Multiplikation nötig.

⇒ Insgesamt werden $2n+1$ Elementaroperationen benötigt, der Aufwand des Algorithmus beträgt $O(n)$.

Beispiel: Aufwandabschätzung für Fibonaccizahlen

```
let rec fib n = match n with
   | 0 -> 0
   | 1 -> 1
   | n -> fib(n-1)+fib(n-2);;
```

Relevante Elementaroperationen: rekursive Aufrufe



Betrachtung des Aufruf-Baums:

⇒ Der Aufwand des Algorithmus beträgt höchstens $O(2^n)$.

⇒ Wer kennt eine bessere Abschätzung?

Das Problem kann auch effizienter gelöst werden.

Algorithmen zur Sortierung von Listen

Sortieren durch direktes Auswählen

- Grundidee: erstes Element der sortierten Liste muß Minimum aller vorhandenen Elemente sein
- Rekursiver Algorithmus: finde in jedem Schritt das Minimum der Elemente und entferne es aus der Liste, hänge das Minimum als Kopf an das Ergebnis des rekursiven Unteraufrufes
- Abbruchbedingung: leere Liste

Sortieren durch direktes Auswählen

Code (Definition von Hilfsfunktionen):

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

let rec findMin_embedded min list = match list with
  | Empty -> min
  | Prepend (head,tail) when (min<head) -> findMin_embedded min tail
  | Prepend (head,tail) when (min>head) -> findMin_embedded head tail;;

let findMin list = match list with
  | Empty -> failwith "not defined"
  | Prepend (head,tail) -> findMin_embedded head tail;;

let rec delete elem list = match list with
  | Empty -> failwith "element not in list"
  | Prepend (head,tail) when (head=elem) -> tail
  | Prepend (head,tail) -> Prepend(head,delete elem tail);;
```

Sortieren durch direktes Auswählen

```
let rec sort list = match list with
  | Empty -> Empty
  | _ -> let min=findMin list in
         Prepend(min,sort (delete min list));;

let x=Prepend(3,Prepend(-20,Prepend(6,Prepend(79,Empty))));;

sort(x);;
```

Sortieren durch direktes Auswählen: Aufwand

- Zur Aufwandsabschätzung von Sortierfunktionen wird typischerweise die Anzahl der benötigten Vergleiche von zwei Elementen der Liste als Maßstab verwendet.
- Beim Aufruf der rekursiven Funktion werden (n-1)-Vergleiche zum Finden des Minimums benötigt. Zudem werden noch mindestens 1 und maximal (n-1) weitere Vergleiche zum Löschen des Minimums benötigt.

$\Rightarrow$ Worst case: Complexity (length n) =2*(n-1)+Complexity(length n-1)= $\sum$ 2*(n-1)=(n-1)*n= n^2-n=$O(n^2)$

Sortieren durch Aufteilen: Quick-Sort

- Sortieren durch Auswählen ist sehr langsam, da wir immer wieder die ganze Liste durchlaufen und diese in jedem Aufruf nur um eins abnimmt.

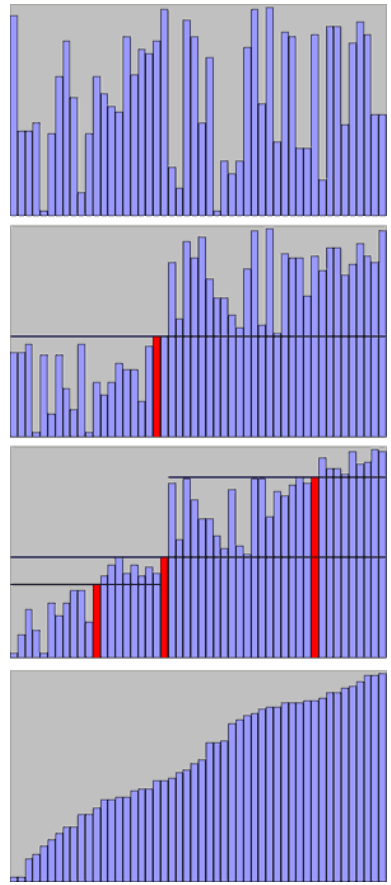
⇒ Rekursionstiefe entspricht der Anzahl der Elemente

⇒ Anzahl der Vergleiche orientiert sich ebenfalls an der Anzahl der Elemente

- **Idee**: Aufteilen der Liste in zwei Listen, wobei die Elemente der ersten Liste alle kleiner sind als die der zweiten Liste

⇒ Bei Halbierung der Liste entspräche die Rekursionstiefe nur noch $O(\log n)$

- **Problem**: Der Aufwand zum Finden des Medians (mittleres Element, Pivot-Element („Drehpunkt“)) entspricht dem Aufwand fürs Sortieren.
- **Ansatz**: Wir nehmen einfach ein Element und hoffen, daß es die Liste möglichst gut teilt



http://encyclopedia.thefreedictionary.com

Quick Sort: Implementierung

Code:

```
let rec filter_lower pivot list= match list with
    | [] -> []
    | hd::tail when (hd <= pivot ) -> hd::(filter_lower pivot tail)
    | hd::tail -> filter_lower pivot tail;;

let rec filter_higher pivot  list = match list with
    | [] -> []
    | hd::tail when (hd > pivot ) -> hd::(filter_higher pivot  tail)
    | hd::tail -> filter_higher pivot  tail;;

let rec quick_sort list = match list with
    | [] -> []
    | hd::tail->quick_sort(filter_lower hd tail)@(hd::quick_sort (filter_higher hd tail));;

let x=12::25::-12::27::80::-73::[];;
quick_sort x;;
```

Quick Sort: Implementierung (mit lokalen Funktionen)

Code:

```
let rec quick_sort list =
    let rec filter_higher pivot list = match list with
        | [] -> []
        | hd::tail when (hd > pivot ) -> hd::(filter_higher pivot tail)
        | hd::tail -> filter_higher pivot tail in
   let rec filter_lower pivot list = match list with
        | [] -> []
        | hd::tail when (hd <=pivot ) -> hd::(filter_lower pivot  tail)
        | hd::tail -> filter_lower pivot  tail in
    match list with
        | [] -> []
        | hd::tail -> quick_sort(filter_lower hd tail)@
                (hd::quick_sort (filter_higher hd tail));;

let x=12::25::-12::27::80::-73::[];;
quick_sort x;;
```

Quicksort

- Sortieren ist eine der am häufigsten benötigten Operationen
- Quicksort einer der meiststudierten Algorithmen überhaupt
- “Benchmark” für die Ausdrucksmächtigkeit/Kompaktheit/Lesbarkeit von Programmiersprachen
- Quicksort in Haskell:

```
sort :: (Ord a) => [a] -> [a]
sort [] = []
sort (pivot:rest) =
    sort [y | y <- rest, y < pivot] ++ [pivot] ++
    sort [y | y <- rest, y >=pivot]
```

Quicksort in “J”:

```
quicksort=: (($:@(<#[) , (=#[) , $:@(>#[)) ({~ ?@#)) ^: (1:<#)
```

Quick Sort in Java (rekursiv) I

```
class QuickSort {
  static void sort(int a[], int lo0, int hi0) {
      int lo = lo0;
      int hi = hi0;
      if (lo >= hi) {
          return;
      }
      int mid = a[(lo + hi) / 2];
      while (lo < hi) {
          while (lo<hi && a[lo] < mid) {
              lo++;
          }
          while (lo<hi && a[hi] >= mid) {
              hi--;
          }
          if (lo < hi) {
              int T = a[lo];
              a[lo] = a[hi];
              a[hi] = T;
          }
      }
      if (hi < lo) {
          int T = hi;
          hi = lo;
          lo = T;
      }
      sort(a, lo0, lo);
      sort(a, lo == lo0 ? lo+1 : lo, hi0);
  }
```

Quick Sort in Java (rekursiv) II

```
    static void sort(int a[]) {
        sort(a, 0, a.length-1);
    }

    static void printList (int a[]) {
        for (int i = 0; i < a.length; i++) {
            System.out.print (a[i] + "-");
        }
        System.out.println ();
    }

        public static void main (String args[]) {
        int test[] = {45, 30, 21, 2, 45, 3, 4, 5, 6};
        printList (test);
        sort (test);
        printList (test);
    }
}
```

Merge Sort

- Quick Sort sortiert sehr gut im Average Case, aber schlecht im Worst Case (wieso?)
- Neuer Ansatz: Zusammenfügen von sortierten Listen geht sehr effizient ($O(n)$) → Aufteilen der Listen (Halbierung) und rekursiver Aufruf, Abbruch, sobald die Liste höchstens ein Element enthält → → maximale Rekursionstiefe $O(\log n)$

  → Aufwand insgesamt $O(n \log n)$
- Insgesamt werden also zwei Hilfsfunktionen benötigt *divide* zum Aufteilen der Liste und *merge* zum Zusammenfügen sortierter Listen
- Generell wird die Klasse von Verfahren, bei denen das Problem in kleinere Probleme aufgeteilt wird, „**divide & conquer**“ genannt.
- Anmerkung: der Aufwand für vergleichsbasierte Sortieralgorithmen beträgt immer mindestens $O(n \log n)$ (siehe später)

Merge Sort

Code:

```
let rec divide_embedded list  list1 list2 = match list with
    | []-> (list1,list2)
    | hd::[] -> (hd::list1,list2)
    | hd1::hd2::tail -> divide_embedded tail (hd1::list1) (hd2::list2);;

let divide list = divide_embedded list [] [];;

let rec merge list1 list2 = match (list1,list2) with
    |([],_) -> list2
    |(_,[]) -> list1
    |(hd1::tail1,hd2::tail2) when (hd1<=hd2) -> hd1::(merge tail1 list2)
    |(hd1::tail1,hd2::tail2) when (hd1>hd2) -> hd2::(merge list1 tail2);;
```

Merge Sort

Code:

```
let rec merge_sort list = match list with
   | [] -> []
   | (hd::[]) -> (hd::[])
   |  _ -> match (divide list) with
       | (l1,l2) -> merge (merge_sort l1) (merge_sort l2);;


let x=12::25::-12::27::80::-73::[];;

merge_sort x;;
```

Merge Sort (mit lokal definierten Funktionen)

Code:

```
let rec merge_sort list =
    let rec merge list1 list2 = match (list1,list2) with
        |([],_) -> list2
        |(_,[]) -> list1
        |(hd1::tail1,hd2::tail2) when (hd1<=hd2) -> hd1::(merge tail1 list2)
        |(hd1::tail1,hd2::tail2) when (hd1>hd2) -> hd2::(merge list1 tail2) in
    let divide list list1 list2 = match list with
        | []-> (list1,list2)
        | hd::[] -> (hd::list1,list2)
        | hd1::hd2::tail -> divide_embedded tail (hd1::list1) (hd2::list2) in
    match list with
    | [] -> []
    | (hd::[]) -> (hd::[])
    |  _ -> match divide list [] [] with
          | (l1,l2) -> merge (merge_sort l1) (merge_sort l2);;

let x=12::25::-12::27::80::-73::[];;
merge_sort x;;
```

Bemerkung: Untere Schranke für Komplexität von Sortieren

- Fragestellung: Gibt es eine untere Schranke für vergleichsbasiertes Sortieren?
- Ja:
  - Annahme: Sie B ein Entscheidungsbaum für unseren Algorithmus, der als Blätter alle möglichen Permutationen unserer Liste enthält. Für eine Liste mit *n* Elementen gibt es *n!* Permutationen und somit *n!* Blätter.
  - Jeder innere Knoten im Baum steht für einen Vergleich (unnötige Vergleiche werden ausgeschlossen).
  - Um die Permutation zu bestimmen sind mindestens log (n!) Vergleiche nötig (Tiefe des Baumes).
  - n! kann mit $n! \geq (n/2)^{(n/2)}$ nach unten hin abgeschätzt werden.

$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\left(\frac{n}{2}\right)}\right) = \left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) = \Omega(n \cdot \log(n))$$

Kurzer Exkurs:
Strukturelle Induktion
(Beweisverfahren auf rekursiven Datenstrukturen)

Strukturelle Induktion

- Wir kennen bereits vollständige Induktion als Beweisverfahren. Dieses Verfahren ist jedoch für rekursive Datenstrukturen nur eingeschränkt benutzbar.
- Strukturelle Induktion ist eine weitere Form der Induktion, die auf der Teilstruktur-Relation beruht
- Die Teilstruktur-Relation erlaubt eine Aussage der Art $x<y$, falls x eine Teilstruktur von y ist (im Fall von Listen, gilt $x<y$ falls x eine Teilliste von y ist)
- Beispiel: Strukturelle Induktion über Listen
  - **Induktionsanfang**: Zum Induktionsanfang betrachten wir immer die leere oder die 1-elementige Liste
  - **Induktionsvoraussetzung**: Die Korrektheit sei nun für alle n-elementigen Listen bewiesen.
  - **Induktionsschritt**: Wir betrachten nun (n+1)-elementige Listen und versuchen das Problem auf n-elementige Listen zu reduzieren.

Beispiel I für Strukturelle Induktion: Assoziativität der Konkatenation

(Komplexere Beispiele später)

- **Gegeben**: Konkatenation, Operator @

$$@: List \times List \rightarrow List$$

$$[\,]@y = y$$

$$(hd::tail)@y = hd::(tail@y)$$

- **Zu zeigen**: Konkatenation ist assoziativ, es gilt also

$$(x@y)@z = x@(y@z)$$

Beispiel I für Strukturelle Induktion: Assoziativität der Konkatenation

Induktion über *x*:

- **Induktionsanfang**: *x=[ ]*

  Definition von @

  $$(x@y)@z=([\,]@y)@z=y@z=[\,]@(y@z)=x@(y@z)$$

- **Induktionsvoraussetzung**: Annahme gilt für alle Listen *lx* mit *length(lx)≤n*

- **Induktionsschritt**: *lx* $\rightarrow$ *hd::lx*

  $$((hd::lx)@y)@z=(hd::(lx@y))@z=hd::((lx@y)@z)=hd::(lx@(y@z))=$$
  $$(hd::lx)@(y@z)$$

  Definition von @ — Induktionsvoraussetzung

Beispiel II für Strukturelle Induktion: Korrektheit von Merge-Sort

- Zu beweisen: mergeSort *x* sortiert eine Liste *x* (unter der Annahme, daß merge zwei sortierte Listen zu einer sortierten Liste zusammenfügt und divide eine Liste der Länge größer 2 in zwei nicht-leere Listen aufteilt)

- **Induktionsanfang**:
  - *x*=[ ]: mergeSort [ ] = [ ], korrekt, da leere Listen per Definition sortiert sind
  - *x*=*hd*::[ ]: mergeSort *x* = *hd*::[ ], korrekt, da 1-elementige Listen per Definition sortiert sind

- **Induktionsvoraussetzung**: mergeSort sortiert alle Listen *x*, falls gilt length $x \leq n$

- **Induktionsschritt**: $x \rightarrow hd::x$

  let *(l1,l2)*=divide(*hd::x*)

  mergeSort(*hd::x*) = merge (mergeSort *l1*) (mergeSort *l2*)

  mergeSort *l1* und mergeSort *l2* liefern nach Induktionsvoraussetzung sortierte Listen, da gilt length $l1 \leq n$ und length $l2 \leq n$. Da merge nach Induktionsvoraussetzung zwei sortierte Listen zu einer sortierten Liste zusammenfasst, ist mergeSort korrekt.

Beispiel III für Strukturelle Induktion: Korrektheit von merge

- Zu beweisen: merge *x y* fügt zwei sortierte Listen *x,y* zu einer sortierten Liste zusammen
- **Induktionsanfang**:
  - *x*=[ ], *y* beliebig: merge *x y* = *y*, korrekt
  - *y*=[ ], *x* beliebig: merge *x y* = *x*, korrekt
- **Induktionsvoraussetzung**: merge *x y* ist korrekt für Listen mit length *x* = n_1 und length *y* = n_2
- **Induktionsschritt:** *x*→*hd::x*, wobei angenommen, daß *hd* < head *y* (*y*→*hd::y* analog)

  merge *hd::x y* = *hd*::merge *x y*

  Korrekt, da merge *x y* nach Induktionsvoraussetzung eine sortierte Liste zurückliefert und *hd* als kleinstes Element vorne angefügt wird.

Zentrale Datenstrukturen: Bäume

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

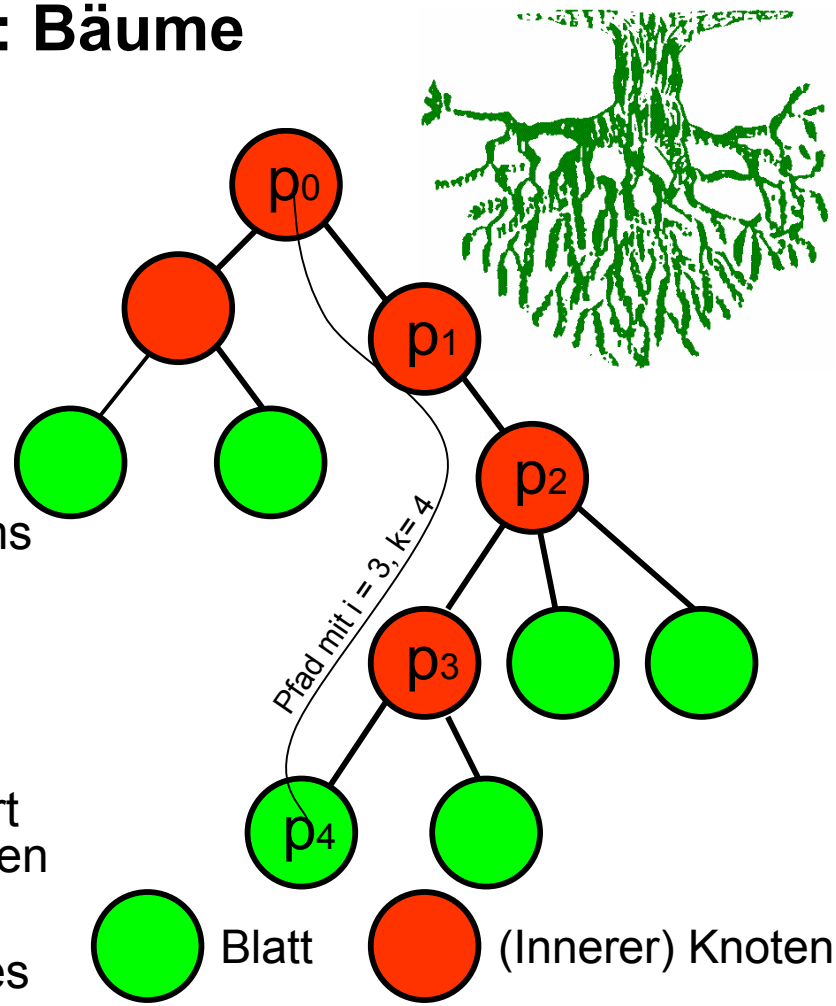
- Wie Listen, so sind auch Bäume zentrale Datenstrukturen in der Informatik: Codebäume, Suchbäume, Syntaxbäume, Entscheidungsbäume, Dateibäume ...
- Bäume bestehen aus Knoten unterschiedlichen Typs; die Verbindung zwischen zwei Knoten wird als *Kante* bezeichnet.

In der Informatik wachsen die Bäume nach unten!

Datenstruktur: Bäume

Definitionen:

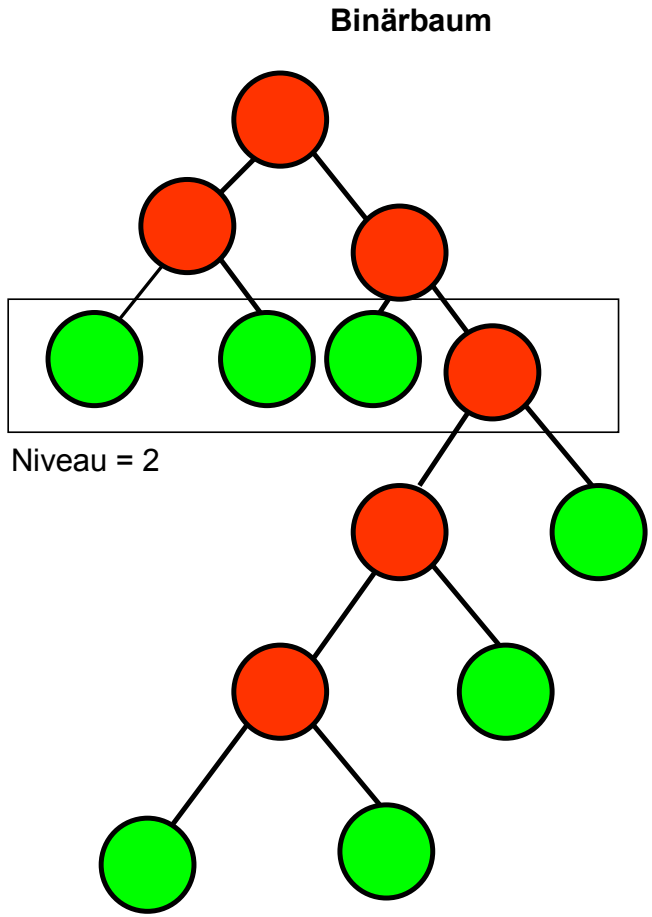
- Baum ist verallgemeinerte Listenstruktur.
- Ein Element (Knoten) hat nicht nur einen Nachfolger (wie im Fall der Liste), sondern eine begrenzte Anzahl von Nachfolgern (***Söhne***).
- Als **Rang** # eines Knotens wird die Anzahl der Söhne bezeichnet.
- „Oberster" Knoten ist die **Wurzel** des Baums (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*: Maximaler Rang eines Knotens.



Binärbäume

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: Die Anzahl der Knoten auf dem Pfad zum tiefsten Blatt
- 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*



Binärbäume in OCaml

Code:

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

let max a b =
   if(a>b) then
        a
   else
        b;;

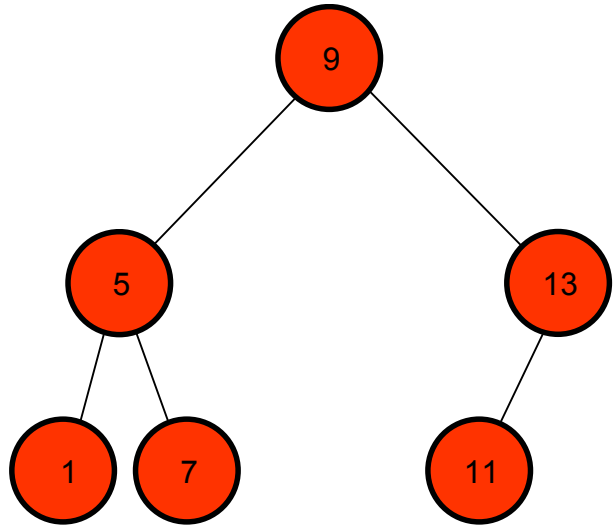
let rec height tree = match tree with
   | Empty -> 0;
   | Node(left,elem,right)->1+max (height left) (height right);;
```

Binärbäume in OCaml

Code:

```
let testTree = Node
    (
        Node
        (
                Node(Empty,1,Empty)
                ,5
                ,Node(Empty,7,Empty)
        )
        ,9
        ,Node
        (
                Node(Empty,11,Empty)
                ,13
                ,Empty
        )
    );;

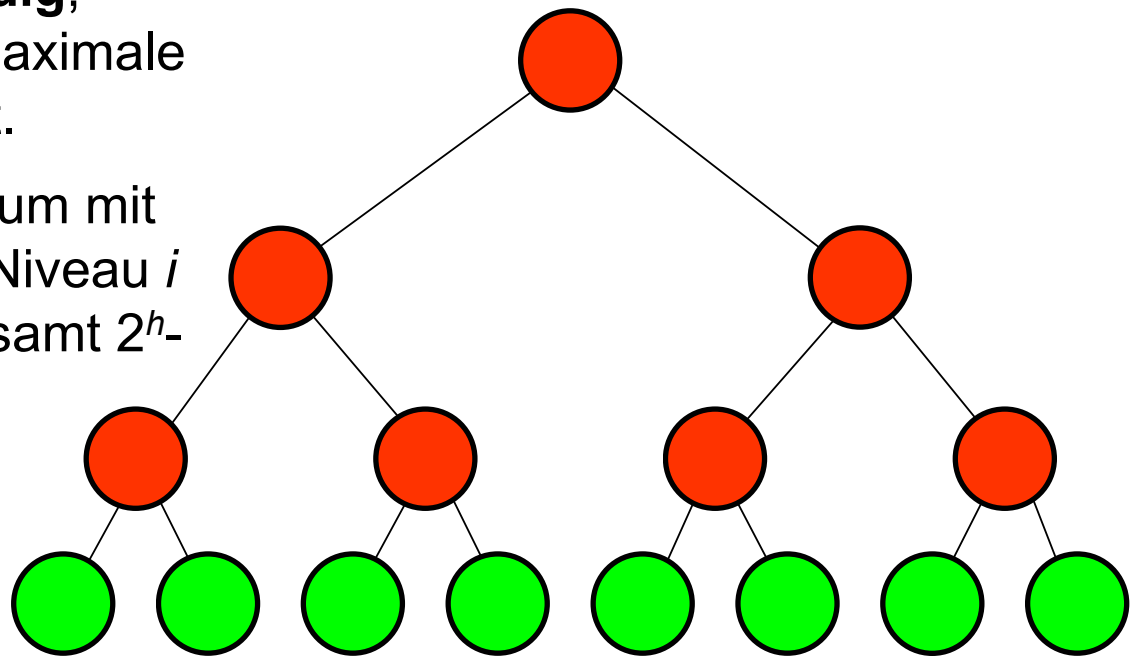
height testTree;;
```



Vollständige Bäume

Definitionen:

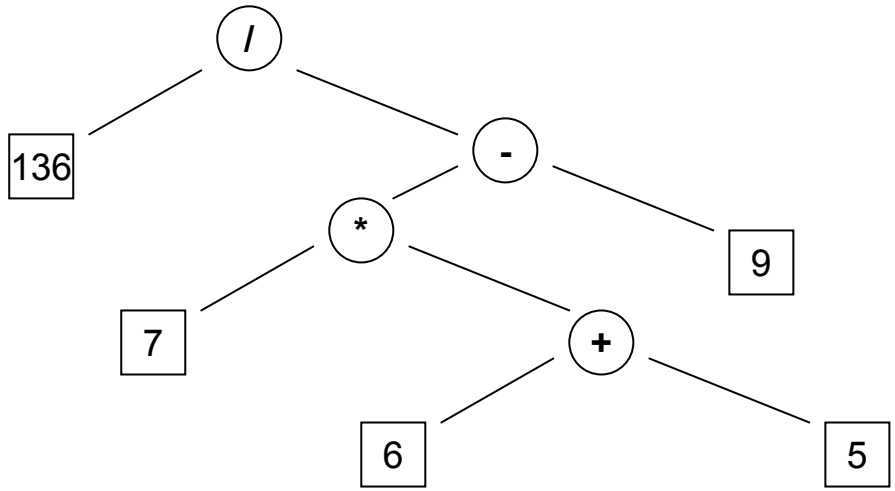
- Ein Baum heißt **vollständig**, wenn jedes Niveau die maximale Anzahl an Knoten enthält.
- Ein vollständiger Binärbaum mit Höhe *h* enthält in jedem Niveau *i* 2^i Knoten und also insgesamt 2^h-1Knoten



OCaml Typdeklaration für Binärbäume

- 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;;
```



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

OCaml-Typdeklaration für Binärbäume

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

```
# 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));;
```

OCaml-Typdeklaration für Binärbäume

- 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);;
```

  Für unseren Ausdruck ergibt sich damit:

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

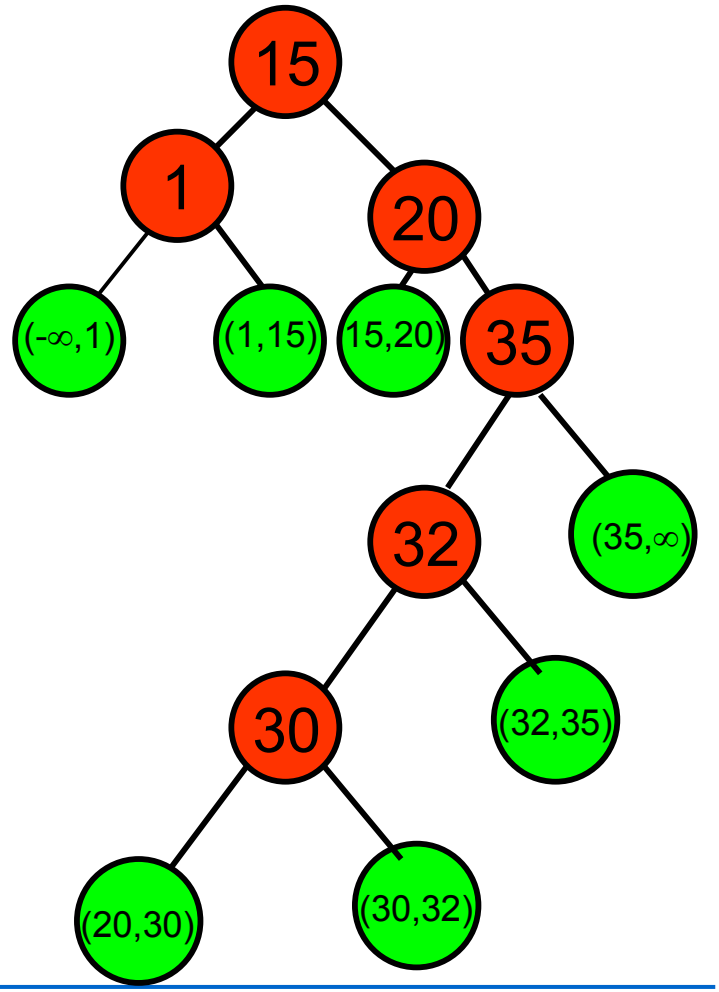
- Übung: Man baue analog der Funktion `set_of_list` (Info-1-homepage) aus einem in einer Liste repräsentierten korrekten arithmetischen Ausdruck wie `[136;/;(;7;*;(;6;+;5;);-;9;)]` einen Baum und werte ihn aus!

Suchbäume

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 wieder finden 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*

Binärer Suchbaum

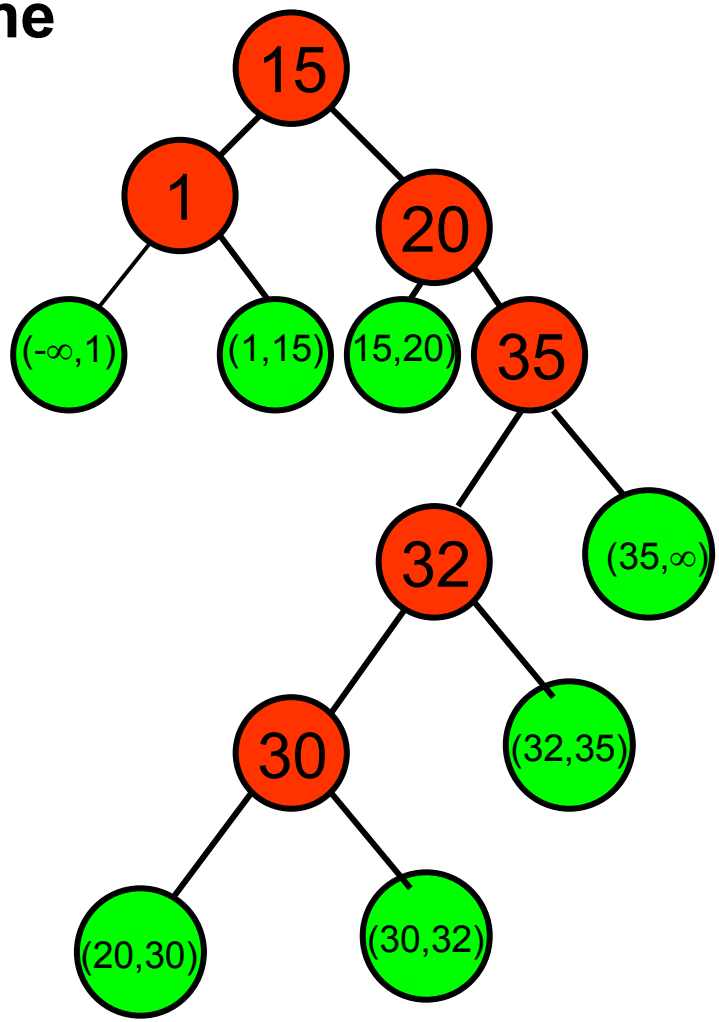


Suchbäume

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

Binärer Suchbaum



Suchbäume – isElement Funktion

Code:

```
let rec isElement elem tree = match tree with
   | Empty -> false
   | Node(left,value,right) when (elem=value) -> true
   | Node(left,value,right) when (elem<value) -> isElement elem left
   | Node(left,value,right) -> isElement elem right;;

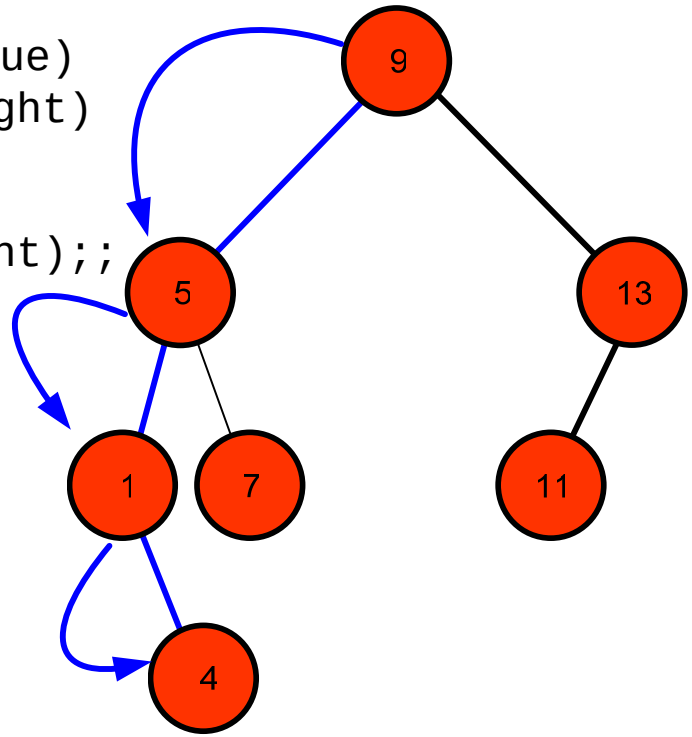

isElement 1 testTree;;
```

Suchbäume – insert Funktion

Code:

```
let rec insert elem tree = match tree with
   | Empty -> Node(Empty,elem,Empty)
   | Node(left,value,right) when (elem<value)
        -> Node(insert elem left,value, right)
   | Node(left,value,right)
        -> Node(left,value,insert elem right);;

insert 4 testTree;;
```



Suchbäume: Entfernen von Elementen

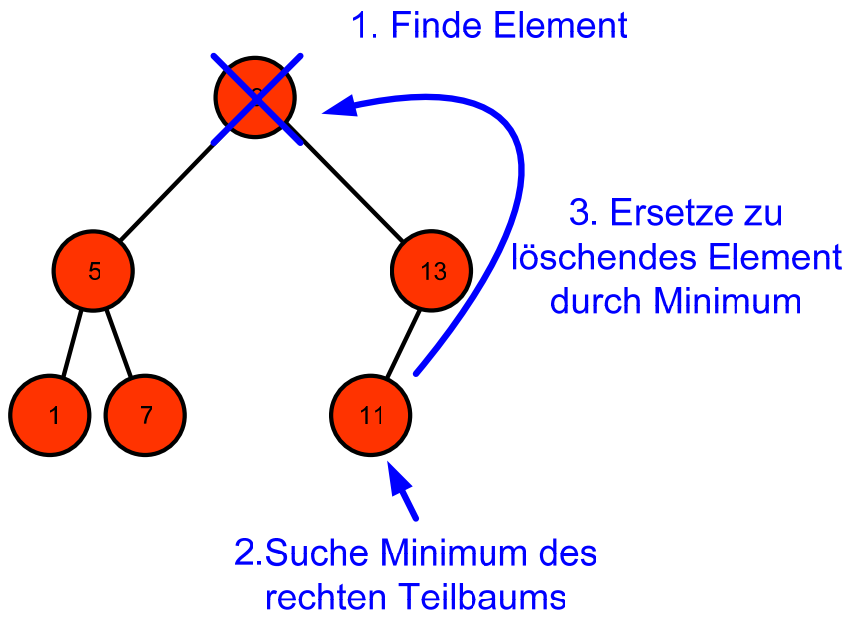
- Ziel: Entfernen eines Elementes unter Beibehaltung der Suchbaumeigenschaft
- Algorithmus:
  1. Suchen nach dem zu entfernenden Element *elem*
  2. *elem* hat 0,1 oder zwei Söhne:
     - kein Sohn: fertig, der Vater von *elem* bekommt einen Empty-Eintrag
     - ein Sohn: fertig, der Vater von *elem* bekommt den Sohn von *elem* als neuen Sohn
     - zwei Söhne: Suche und Extraktion nach dem kleinsten Element im rechten Teilbaum, dieses Element wird anstelle von *elem* eingehängt.

Suchbäume: delete Funktion

Code:

```
let rec extract_min tree= match tree with
    | Node(Empty,value,right) -> (value,right)
    | Node(left,value,right) ->
          match extract_min left with
             | (min,rest) ->
    (min,Node(rest,value,right));;

let rec delete elem tree = match tree with
    | Empty -> Empty
    | Node(left,value,right) when (value> elem)
          -> Node(delete elem left,value,right)
    | Node(left,value,right) when (value< elem)
          -> Node(left,value,delete elem right)
    | Node(Empty,_,Empty) -> Empty
    | Node(Empty,_,right) -> right
    | Node(left,_,Empty) -> left
    | Node(left,_,right) -> match extract_min right with
          |(min,rest) -> Node(left,min,rest);;
delete 9 testTree;;
```



Komplexität der Algorithmen

- Die Funktionen insert, delete und isElement durchlaufen beim Aufruf immer genau einen Pfad des übergebenen Baums.
- Die Rekursionstiefe und damit die Komplexität ist also durch die Höhe gegeben:

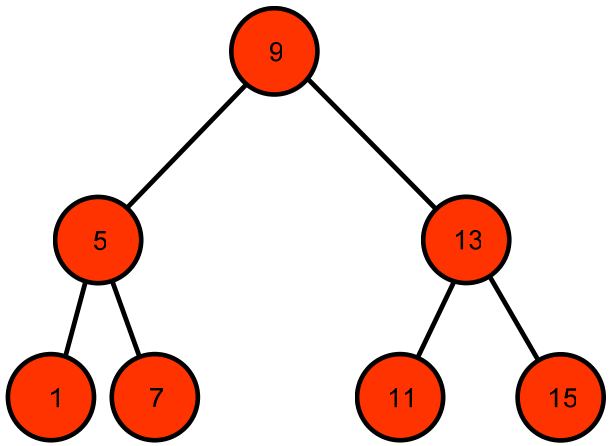
  Complexity(insert elem tree) = Complexity (delete elem tree) = Complexity (isElement elem tree)= O(height tree)

- **Problem:** Abhängig davon, wie ein Baum aufgebaut wird, entstehen Bäume unterschiedlicher Höhe
- **"Gute Nachricht":** Im Mittel beträgt die Höhe eines Binärbaums mit *n* Elementen *O(ld n)*
- **"Schlechte Nachricht":** Im "worst case" ist ein Baum zur Liste degeneriert → alle Operationen haben eine Komplexität von *O(n)*

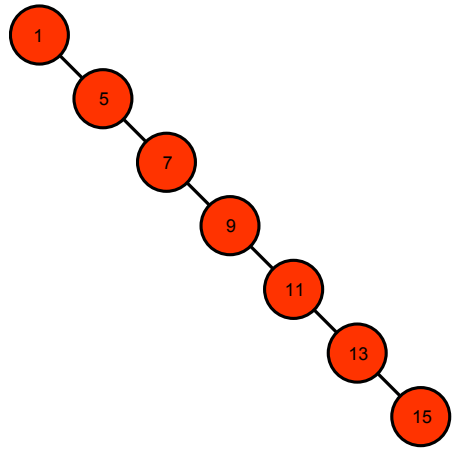
Beispiel für unterschiedliche Höhen

```
let rec convertToTree_embedded list tree = match list with
   | [] -> tree
   | hd::tail -> convertToTree_embedded tail (insert hd tree);;
let convertToTree list = convertToTree_embedded list Empty;;
let list = 9::5::13::1::7::11::15::[];;
```

```
convertToTree list;;
```




```
convertToTree (merge_sort list);;
```



Höhenbalancierte Bäume

- AVL-Bäume
- B-Bäume
- Rot-Schwarz-Bäume

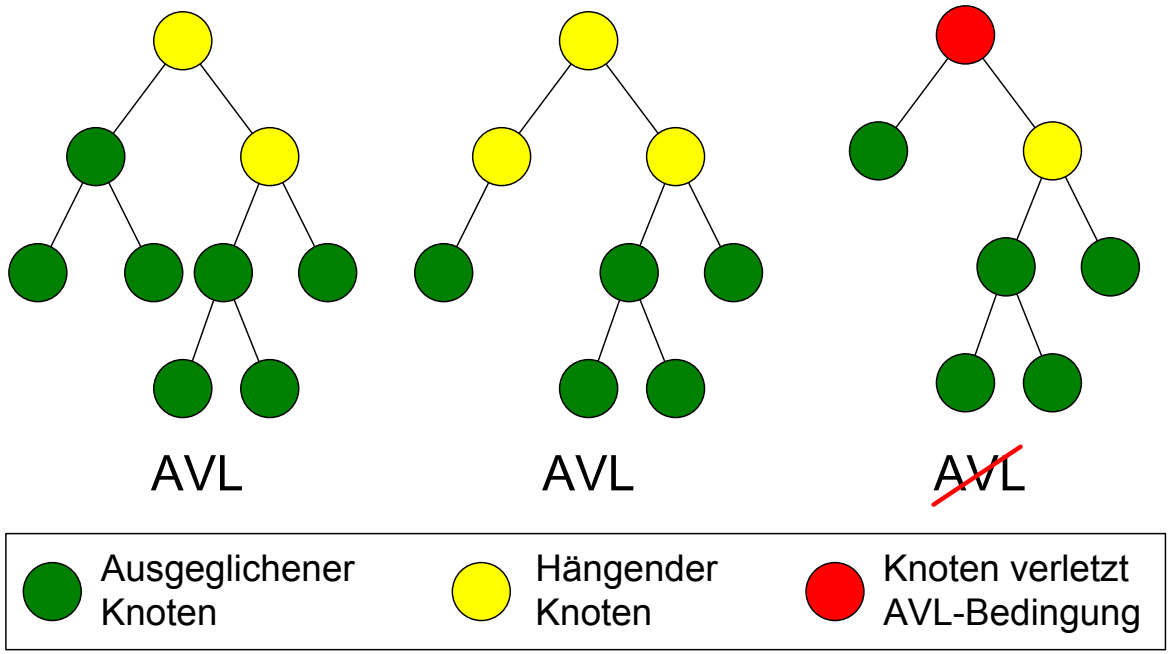
Balancierte Bäume

- **Grundidee**: Bei den Operationen **einfügen** und **delete** muß darauf geachtet werden, daß die Knoten relativ gleichmäßig verteilt sind
- **Deshalb:** Aufstellen von Bedingungen (*Invarianten*), deren Einhaltung bei den Schlüssel-Operation geprüft wird und die ein Degenerieren verhindern.
- **Problem**: Das Ausbalancieren muss natürlich auch effizient erfolgen.
- 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

Balancierte Bäume: AVL



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

AVL-Bäume

- Beweis per Induktion über die Höhe: Ein AVL-Baum mit Höhe *h* hat mindestens fib(*h*) Blätter!
- **Induktionsanfang:**
  - *h*=0 → minNode(*h*=0) =0 = fib(0) Blätter
  - *h*=1 → minNode(*h*=1)=1 = fib (1) Blätter
- **Induktionsvoraussetzung**: minNode(*h*) ≥ fib(*h*) für 0…*h*
- **Induktionsschritt:** *h* → *h*+1
  - Mindestens ein Unterbaum hat die Höhe *h*
  - Aufgrund der Invariante hat der zweite Unterbaum mindestens die Höhe *h*-1
  - minNode(*h*+1) ≥ minNode(*h*)+minNode(*h*-1)+1 > minNode(*h*)+minNode(*h*-1) = fib(*h*)+fib(*h*-1)=fib(*h*+1)

AVL-Bäume: Datenstruktur

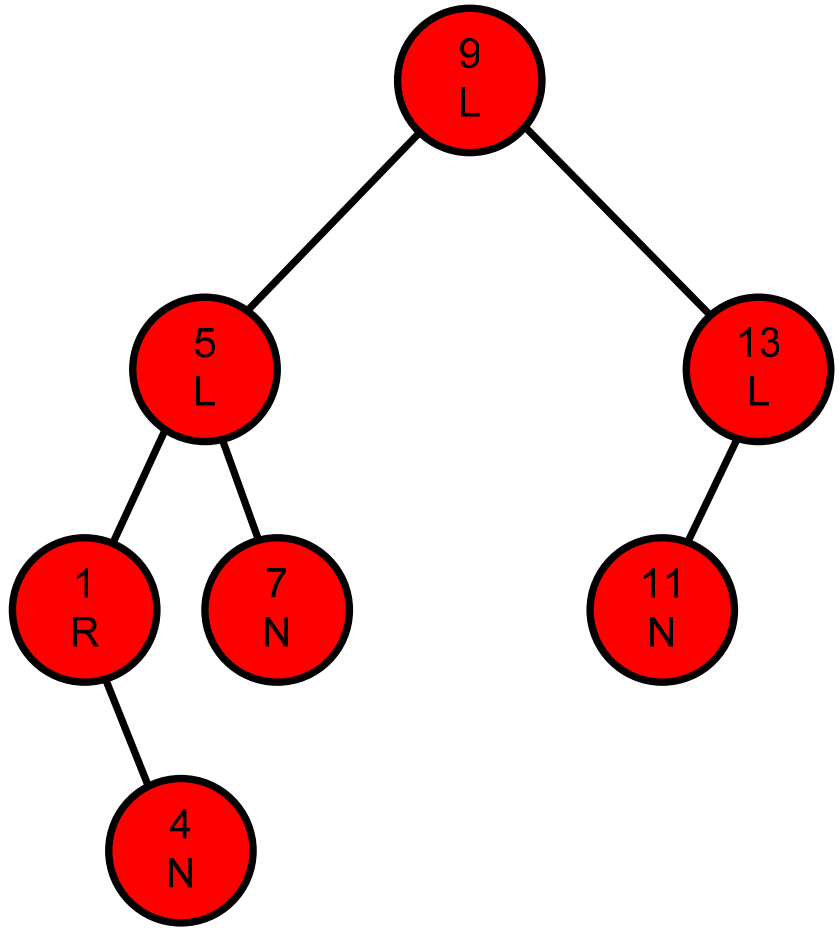
- Neben dem Inhalt des Knotens und den Söhnen interessiert nun auch die Höhe der Teilbäume.
- Es gibt hier drei Möglichkeiten:
  - Gleiche Datenstruktur wie normale Binärbaume. Nachteil: Bei der insert und der delete Funktion muss die Höhe der Teilbäume getestet werden. Dieser Test hat eine *worst case* Laufzeit von O(*n*).
  - Speichern der Höhen der Teilbäume. Nachteil: Wir benötigen zusätzlich zwei weitere Elemente → erhöhter Speicherplatzbedarf.
  - Speichern des Höhenunterschieds: Da sich die Höhe der Söhne nur um eins unterscheiden darf, reicht es aus sich den Höhenunterschied zu merken (L linker Sohn ist tiefer, N neutral, R rechter Sohn ist tiefer). Es wird nur ein zusätzliches Element benötigt, wobei wir diese Information sogar in den Typkonstruktor codieren können.

AVL-Baum: OCaml-Typdefinition

Code:

```
type 'a avl = Empty
     | L of 'a avl * 'a * 'a avl
     | N of 'a avl * 'a * 'a avl
     | R of 'a avl * 'a * 'a avl;;

let testAVL=
L(
   L (
      R (
         Empty,
         1,
         N(Empty,4,Empty)
      ),
      5,
      N(Empty,7,Empty)
   ),
       9,
   L(
      N(Empty,11,Empty),
      13,
      Empty
   )
);;
```



AVL-Baum: Rotationen

- Bei den Operationen insert und delete kann die AVL-Eigenschaft der Bäume verletzt werden.
- Eine komplette Neuordnung des Baums ist vom Aufwand zu teuer.
- Die AVL-Eigenschaft kann aber durch *Rotationen* um die die AVL-Eigenschaft verletzenden Knoten einfach wieder hergestellt werden.
- Rotationsarten:
  - Linksrotation
  - Rechtsrotation
  - Doppelrotation: links-rechts
  - Doppelrotation: rechts-links
- Die Suche nach nötigen Rotationen beginnt dabei immer beim neu hinzugefügten oder beim gelöschten Knoten.
- Auf dem Weg zur Wurzel werden eventuell mehrere Rotationen benötigt (nur bei Löschoperation nötig, beim Einfügen erfolgt höchstens eine Rotation)
- Animation der Rotationen:

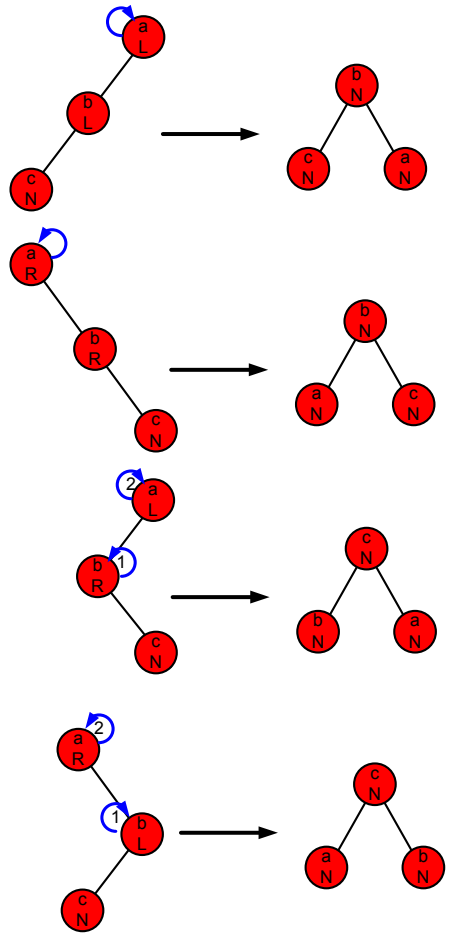
  http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm

  http://www.compapp.dcu.ie/~aileen/balance/

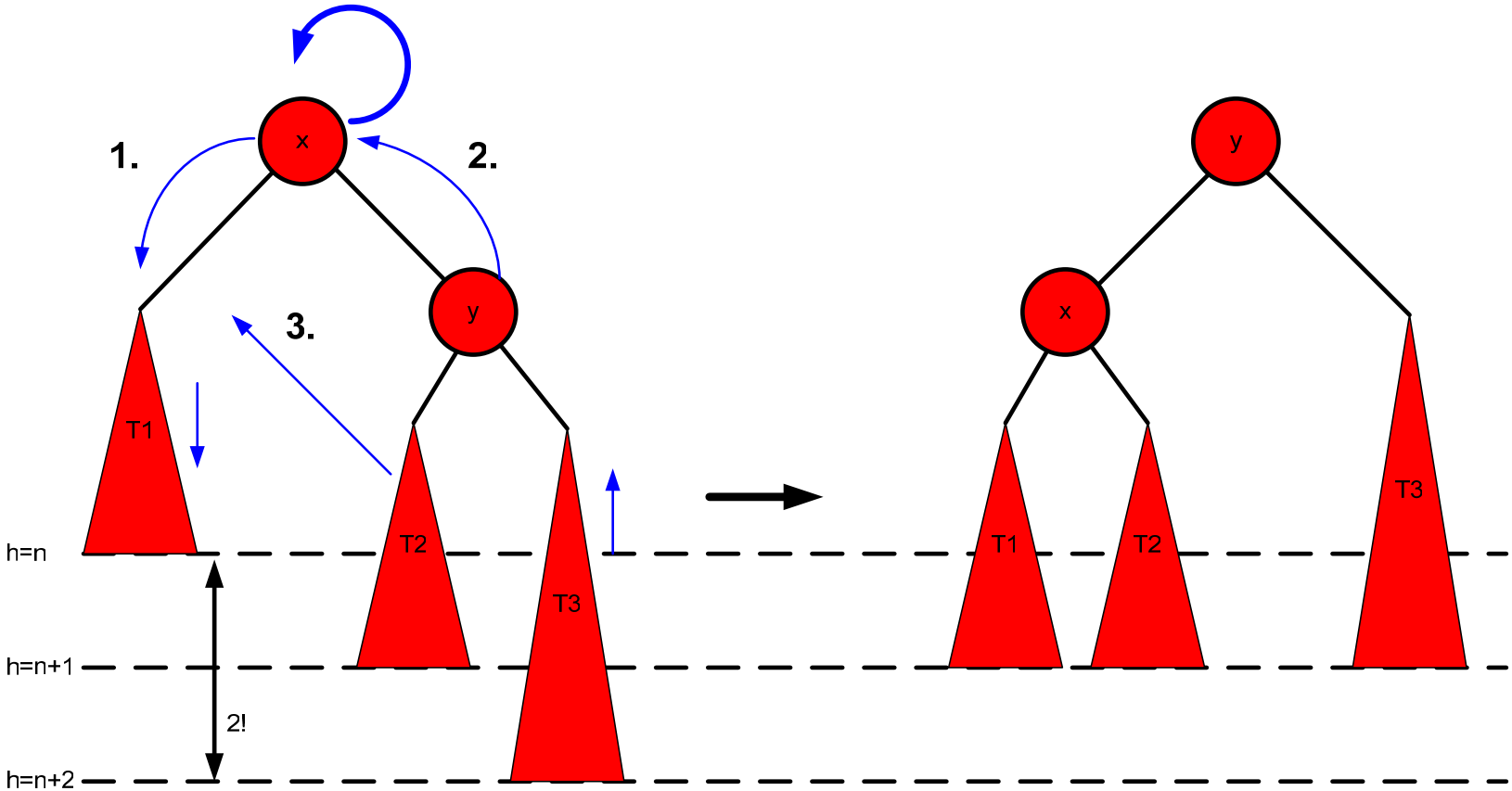
AVL-Bäume: Rotationen

Situationen, die zur Rotation führen:

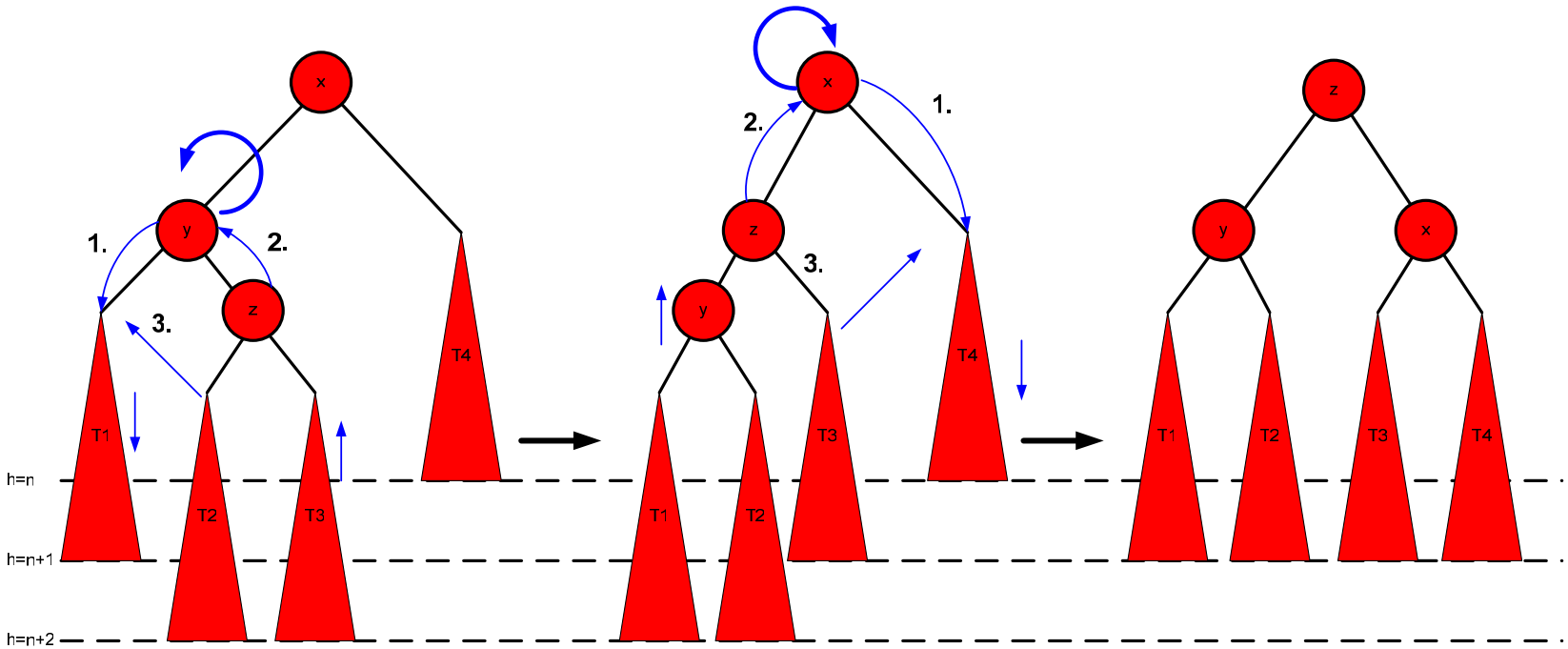
- **Rechtsrotation**: Die Höhe des linken Sohns ist um 2 größer als die des rechten Sohns und der linke Sohn ist linkslastig.
- **Linksrotation**: Die Höhe des rechten Sohns ist um 2 größer als die des linken Sohns und der rechte Sohn ist rechtslastig.
- **Doppelrechtsrotation**: Die Höhe des linken Sohns ist um 2 größer als die des rechten Sohns und der linke Sohn ist rechtslastig.
- **Doppellinksrotation**: Die Höhe des rechten Sohns ist um 2 größer als die des linken Sohns und der rechte Sohn ist linkslastig.



AVL-Baum: Linksrotation



AVL-Baum: Doppelrotation (Links-Rechts)



1. Rückführung auf einfache Rechtsrotation

2. Durchführung der Rechtsrotation

AVL-Baum: Einfügeoperation

- Folgende Schritte müssen beim Einfügen eines Elementes durchgeführt werden:
  1. Suche der korrekten Position des neuen Elementes und Einfügen des Elementes.
  2. Auf der Rückkehr zur Wurzel Balancierung der Knoten aktualisieren → Es ist nötig, dass der rekursive Aufruf neben dem aktualisierten Unterbaum auch angibt, ob die Höhe des Unterbaums gewachsen ist.
  3. Eventuell: Durchführung einer Rotation.

AVL-Bäume: Implementierung in Ocaml

Rotate-Funktion (komplette Version auf der Homepage):

```
let rotate tree = match tree with
  |L(left,x,right) ->
    (match left with
      | L (subleft,subx,subright) -> N(subleft,subx,N(subright,x,right)) (*Rechtsrotation*)
      | R (subleft,subx,subright) ->  (*Doppelrotation links-rechts*)
        (match subright with
          | L(subsubleft,subsubx,subsubright) ->
            N(N(subleft,subx,subsubleft),subsubx,R(subsubright,x,right))
          | R(subsubleft,subsubx,subsubright) ->
            N(L(subleft,subx,subsubleft),subsubx,N(subsubright,x,right))
          | _ -> failwith("invalid parameter")
          )
      | _ -> failwith("invalid parameter")
     )
```

...

AVL-Bäume: Implementierung in Ocaml II

Code der Insertfunktion (kompletter Code auf Homepage):

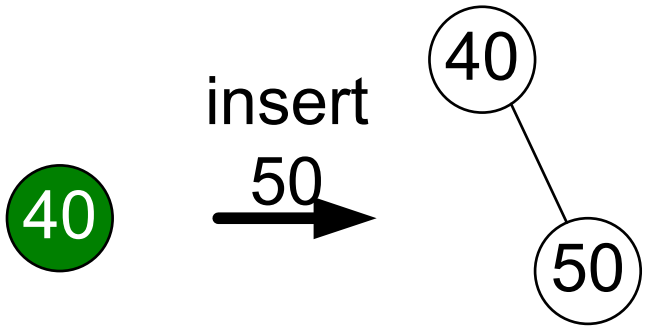
```
let rec insertAVL_embedded elem tree = match tree with
   | Empty -> (N(Empty,elem,Empty),true) (*neuer Knoten, Tiefe vergrössert*)
   | L(left,x,right) when (x>elem) ->
       (match insertAVL_embedded elem left with
         | (subtree,true) -> (rotate(L(subtree,x,right)),false)
         | (subtree,false) -> (L(subtree,x,right),false)
       )
   | L(left,x,right) ->
       (match insertAVL_embedded elem right with
                 | (subtree,true) -> (N(left,x,subtree),false)
                 | (subtree,false) -> (L(left,x,subtree),false)
       )
```

...

AVL-Bäum: größeres Beispiel

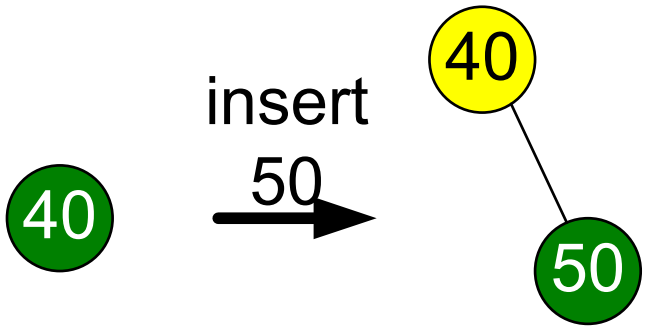
insert 40

AVL-Bäume: Größeres Beispiel



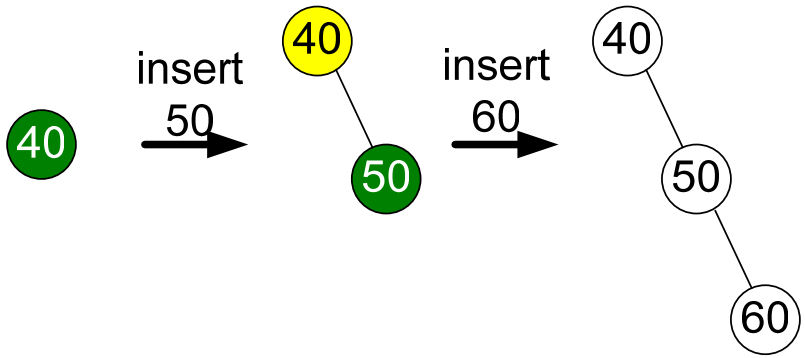
insert 50

AVL-Bäume: Größeres Beispiel



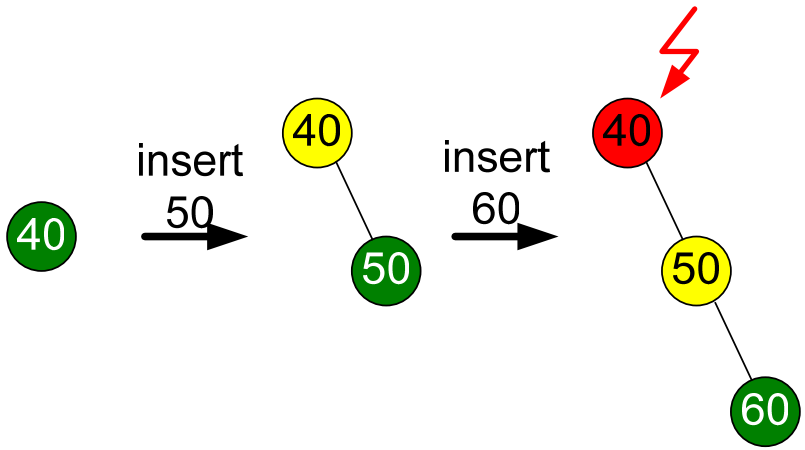
insert 50

AVL-Bäume: Größeres Beispiel



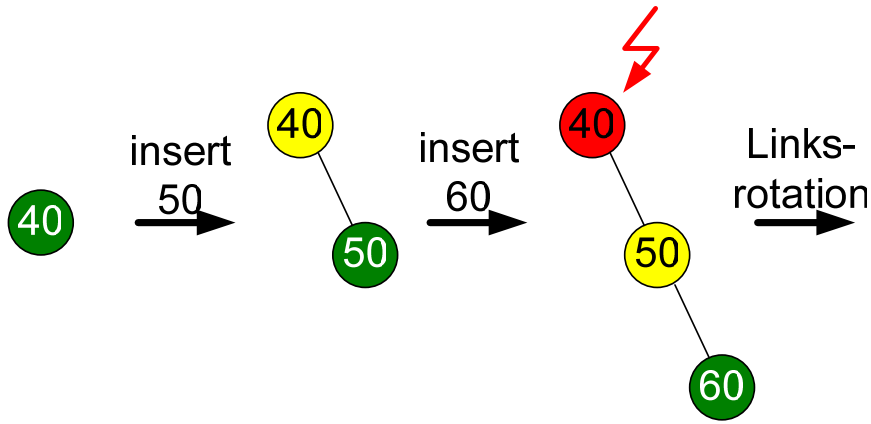
insert 60

AVL-Bäume: Größeres Beispiel



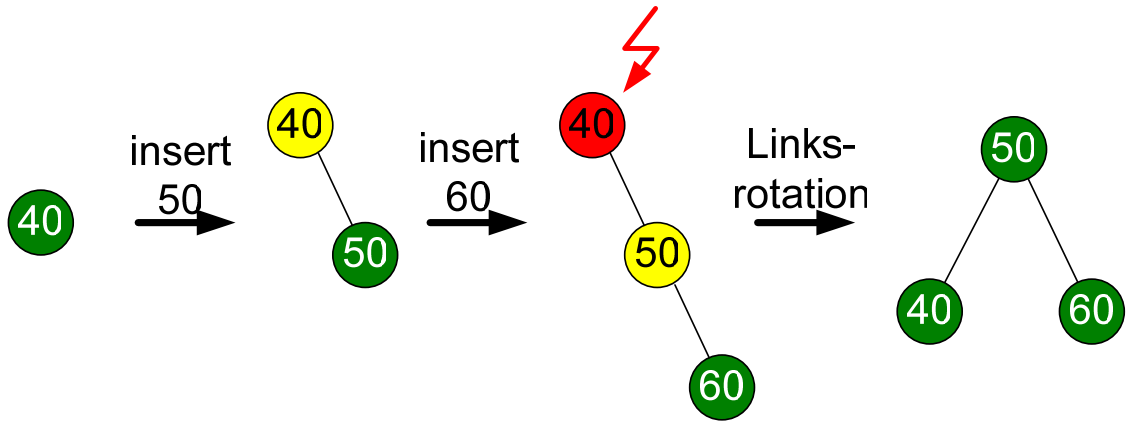
insert 60

AVL-Bäume: Größeres Beispiel



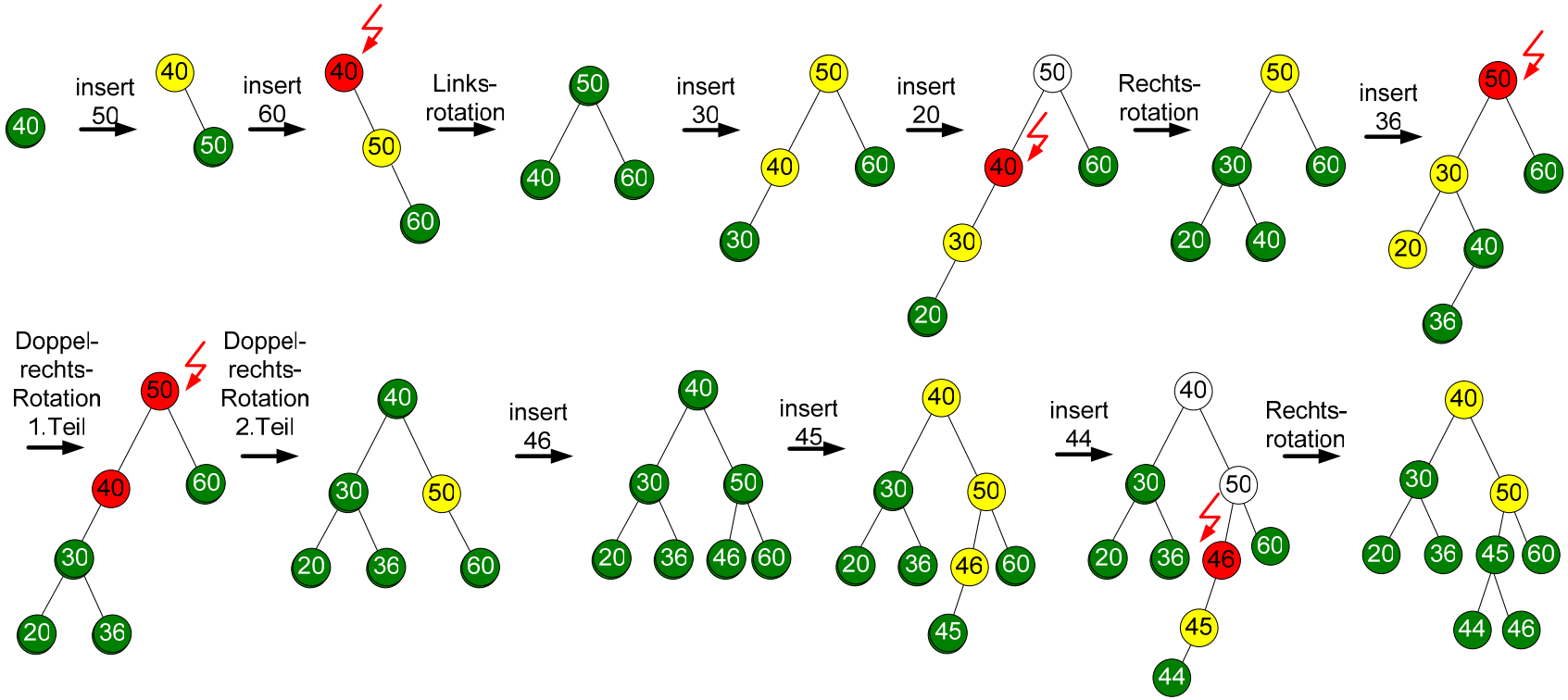
insert 60

AVL-Bäume: Größeres Beispiel



insert 60

AVL-Bäume: Größeres Beispiel



B-Bäume

- Von Rudolf Bayer und Edward M. McCreight 1972 entwickelt
- Aufgabenstellung: Datenstruktur für Datenbanken, Einfügen, Suchen und Löschen soll in logarithmischer Zeit stattfinden.
- Nebenbedingung: Hardwareeigenschaften sollen berücksichtigt werden (nur kleiner Teil der Daten liegen im schnellen Hauptspeicher, gesamte Daten sind persistent auf dem Hintergrundspeicher (z.B. Festplatte) gesichert): die Positionierung des Lese/Schreibkopfs ist sehr aufwendig, Daten an einem Stück sind sehr schnell zu lesen
- Binärbäume sind ungeeignet, da ein Übergang von einem Knoten auf einen anderen einer Neupositionierung des Lesekopfs entspricht.

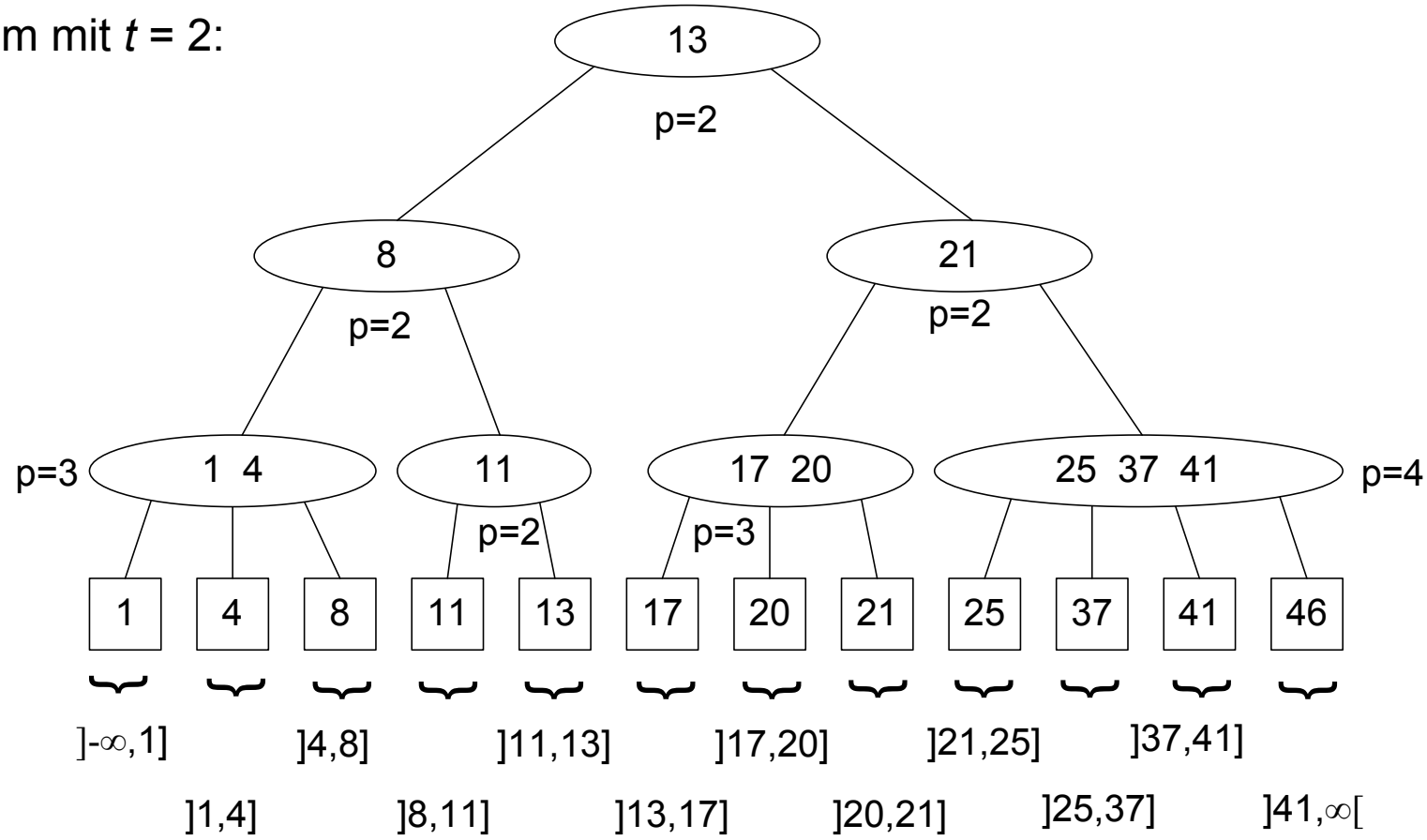
B-Bäume

- Ein Knoten ***v*** des B-Baumes besitzt nicht maximal zwei, sondern zwischen *t* und 2*t* Söhnen. Die Anzahl der Nachfolger von *v* bezeichnen wir mit *p(v).*
  - ⇒ Durch die Variable *t* kann die Datenstruktur einfach der Blockgröße (Speicherplatz der mittels einer Leseoperation gelesen werden kann) des Speichermediums angepaßt werden.
  - ⇒ Je größer *t* gewählt wird, desto flacher werden die Bäume und umso effizienter werden die Einfüge-, Such- und Löschoperationen
  - ⇒ Durch die variable Anzahl der Söhne wird die Anzahl der benötigten Balancierungsoperationen verringert.
- Die Wurzel besitzt zwischen zwei und 2*t* Söhne
- B-Bäume sind externe Bäume, d.h. die Informationen sind ausschließlich in den Blättern codiert.
- An den inneren Knoten werden noch Verwaltungsinformationen gespeichert: $p(v)$-1 Schlüssel $K_1 \ldots K_{p(v)-1}$ mit der Eigenschaft:

  $K_i <$ alle Elemente des *i-ten* Sohns $\leq K_{i+1}$, mit $K_0 = -\infty$ und $K_{p(v)} = +\infty$

B-Bäume: Beispiel

Baum mit $t = 2$:



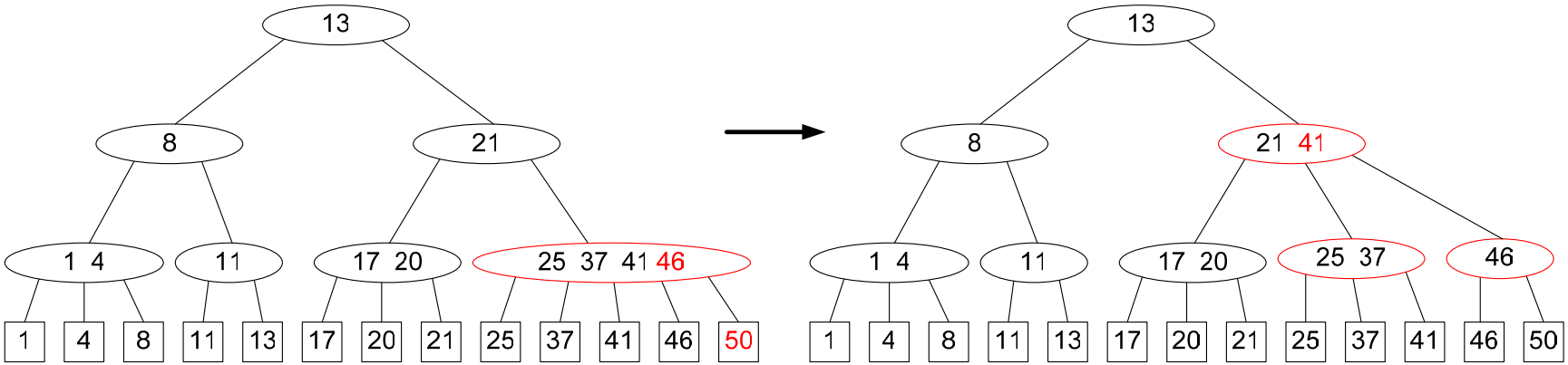
B-Bäume: Einfügen von Elementen

Schritte:

1. Suche des richtigen Interfalls (letzter Knoten *v)* mit Hilfe der in den Knoten gespeicherten Intervalinformationen.
2. Einfügen eines Blattes mit dem Element als Sohn im Knoten *v*.
3. Überprüfung $p(v) \leq 2*t$
   - ja: fertig
   - nein: Aufspalten des Knotens *v* in zwei innere Knoten und Eintragung beim Vater von *v*. Wiederholen des Schritt 3 beim Vater von *v*. Ist *v* die Wurzel des Baums, so reicht das Aufspalten.

B-Bäume: Einfügen von Elementen

Einfügen des Elementes 50:



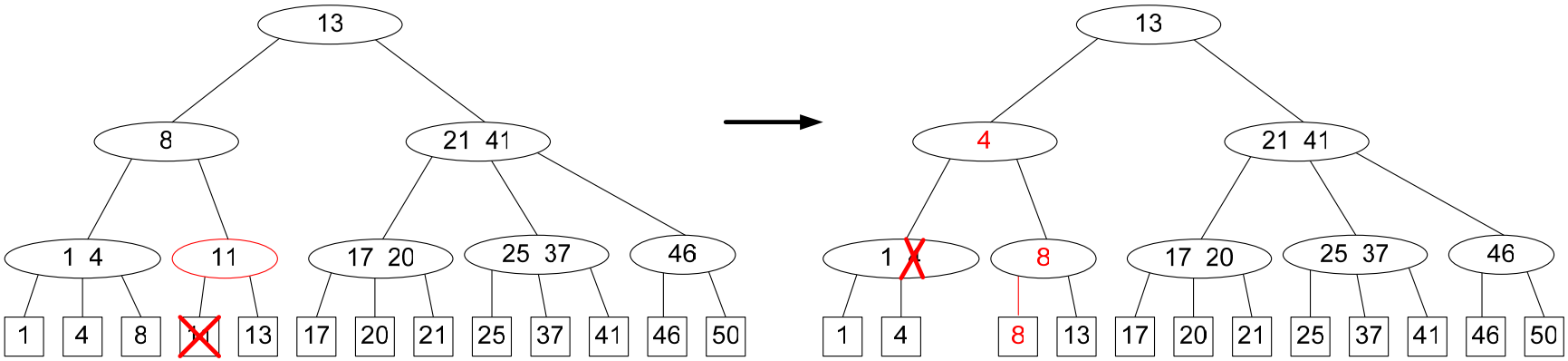
B-Bäume: Löschen von Elementen

Schritte:

1. Suchen des entsprechenden Blattes mit Vater *v*.
2. Löschen des Blattes und des entsprechenden Eintrages im Knoten *v*.
3. Überprüfung $p(v)<t$
   - nein: fertig
   - ja: Betrachung eines Nachbarns (Sohn vom Vater) *v'* von *v*:
     - 1.Fall $p(v')>t$: Adoption des nächsten Kindes von *v'* durch *v*. Fertig.
     - 2. Fall $p(v')=t$: Verschmelzen der beiden Knoten. Fortfahren mit Schritt 3 und Vater von *v'* und *v*.

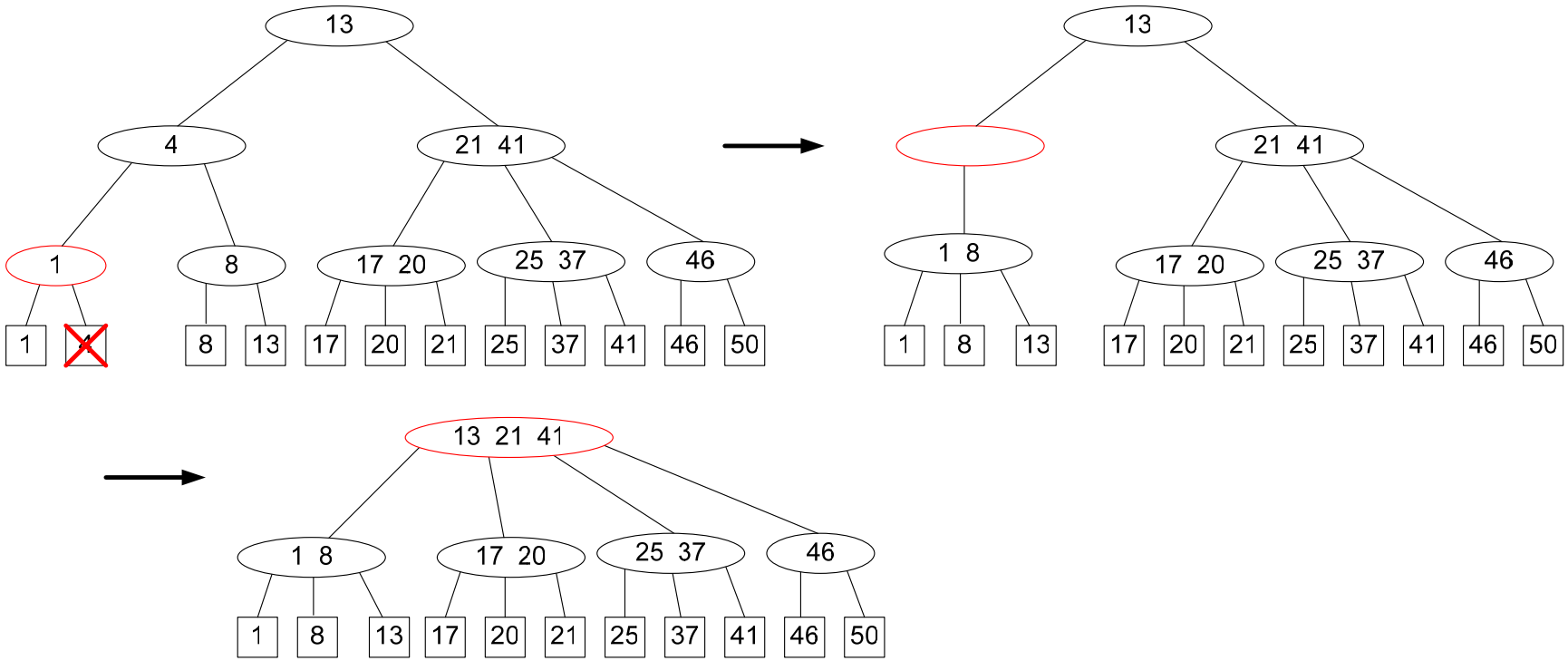
B-Bäume: Löschen von Elementen

Löschen vom Element 11:



B-Bäume: Löschen von Elementen

Löschen vom Element 4:



Rot-Schwarz-Bäume (RB-Baum)

- Wir haben nun verschiedene Bäume kennengelernt
- 2-4-Bäume haben ideale Laufzeiten, da sie stets ausgeglichen sind
- **Problem:** Durch unterschiedliche Mengen an Informationen, die in einem Knoten gespeichert werden können, ist jedoch die Implementierung schwierig.
- **Ziel**: Abbildung aller Klassen der balancierte Bäume (z.B. AVL-Bäume, B-Bäume…) auf ein Baumkonzept $\Rightarrow$ einheitliche Repräsentation und Implementierung.
- Das Baumkonzept soll dabei auf einem Binärbaum wegen der einfachen Implementierung basieren
- Übertragung der Eigenschaften von 2-4-Bäumen auf einen Binärbaum
- **Problem**: Vollständige Ausgeglichenheit kann nicht erhalten bleiben, da dies bei einer Höhe h immer 2^{h-1} Knoten voraussetzt.

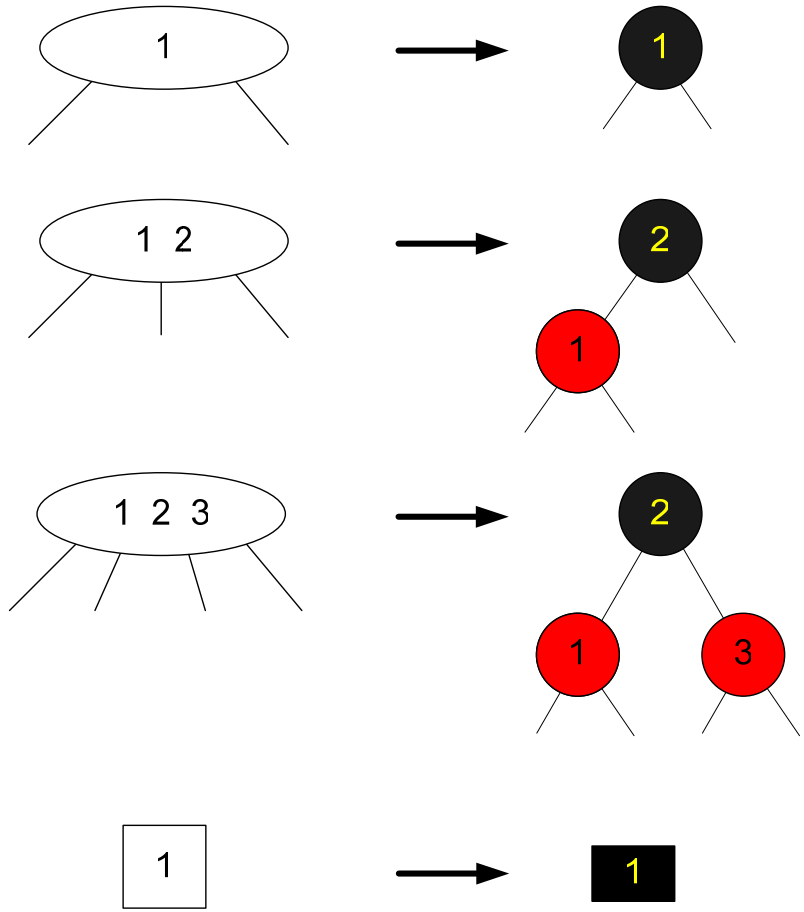
Rot-Schwarz-Bäume: Eigenschaften für 2-4-Bäume

1. Knoten sind entweder rot oder schwarz.
2. Die Wurzel des Baumes ist immer schwarz.
3. **Rot-Bedingung**: Die Kinder eines roten Knotens sind immer schwarz.
4. Blätter sind immer schwarz.
5. **Schwarz-Bedingung**: Jeder Pfad von einem Knoten x zu einem Blatt besitzt gleich viele schwarze Knoten.
6. Die schwarze Höhe ist die Zahl der schwarzen Knoten auf dem Weg zu einem Blatt (beliebiger Pfad wegen 5. → Rot-Schwarz-Bäume sind schwarzbalanciert).
7. Jeder 2-4-Baum lässt sich in einen Rot-Schwarz-Baum umwandeln und umgekehrt.

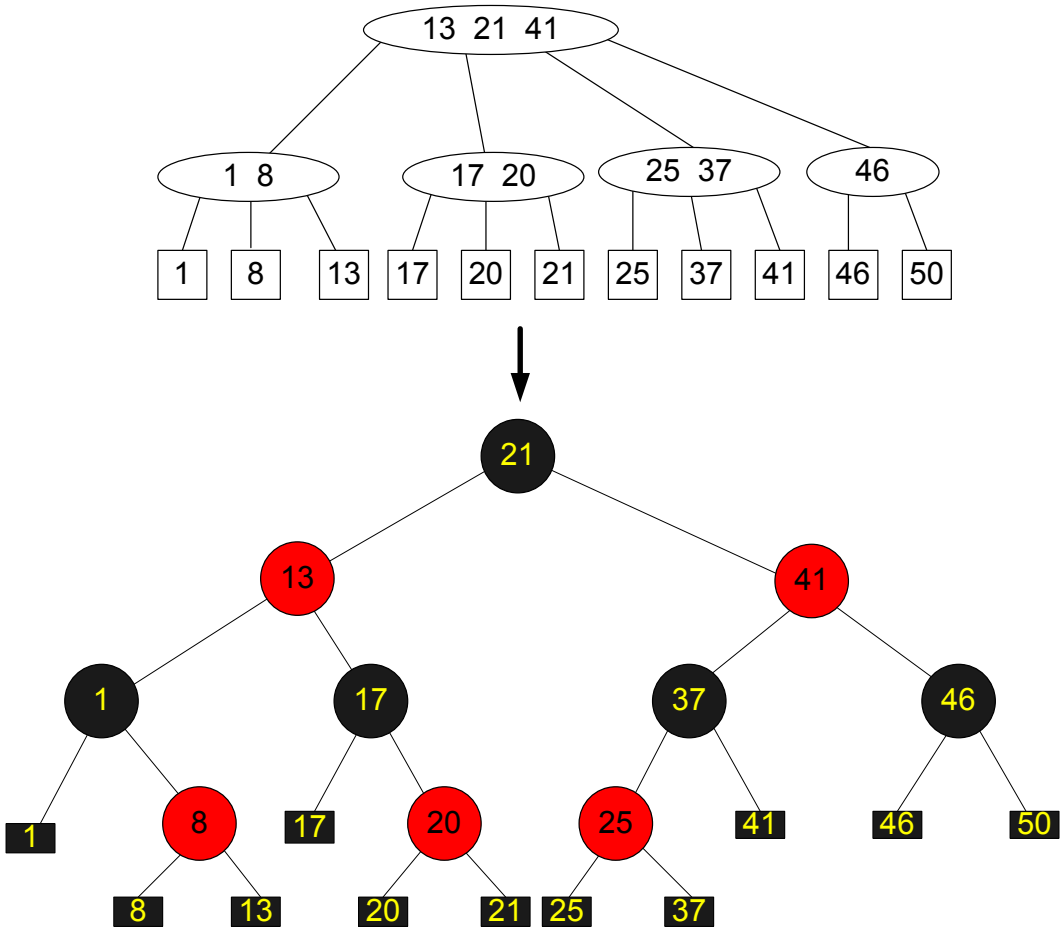
Anmerkung: Die Kanten zu einem schwarzen Knoten entsprechen immer dem Abstieg in die Tiefe im ursprünglichen Baum. Kanten zu roten Knoten (oft auch horizontale Kanten genannt) dienen zur binären Repräsentation von Knoten mit mehr als zwei Nachfolgern.

Umwandlung von 2-4-Baum in Rot-Schwarz-Baum

- Die Umwandlung erfolgt beginnend mit der Wurzel und dann absteigend nach folgendem Muster:
  - Wahl des mittleren Schlüssels (Knoten schwarz)
  - Besitzt mittlerer Schlüssel einen linken bzw. rechten Nachbar-schlüssel, so werden dem Knoten ein linker oder rechter roter Nachfolger zugewiesen



Rot-Schwarz-Bäume: Umwandlung am Beispiel



Rot-Schwarz-Bäume: Implementierung

```
type 'a rbTree =
    | EmptyRBTree
    | Leaf of 'a
    | BlackNode of 'a rbTree * 'a * 'a rbTree
    | RedNode of 'a rbTree * 'a * 'a rbTree;;

let myRBTree =
BlackNode(
  RedNode(
    BlackNode(Leaf(1),1,RedNode(Leaf(8),8,Leaf(13))),
    13,
    BlackNode(Leaf(17),17,RedNode(Leaf(20),20,Leaf(21)))
  ),
  21,
  RedNode(
    BlackNode(RedNode(Leaf(25),25,Leaf(37)),37,Leaf(41)),
    41,
    BlackNode(Leaf(46),46,Leaf(50))
  )
);;
```

Rot-Schwarz-Bäume: Einfügeoperation I

- Das Einfügen erfolgt zunächst wie beim Binärbaum, am Blatt angekommen wird dann ein neues Blatt (schwarz) und ein neuer Knoten (rot) angelegt.

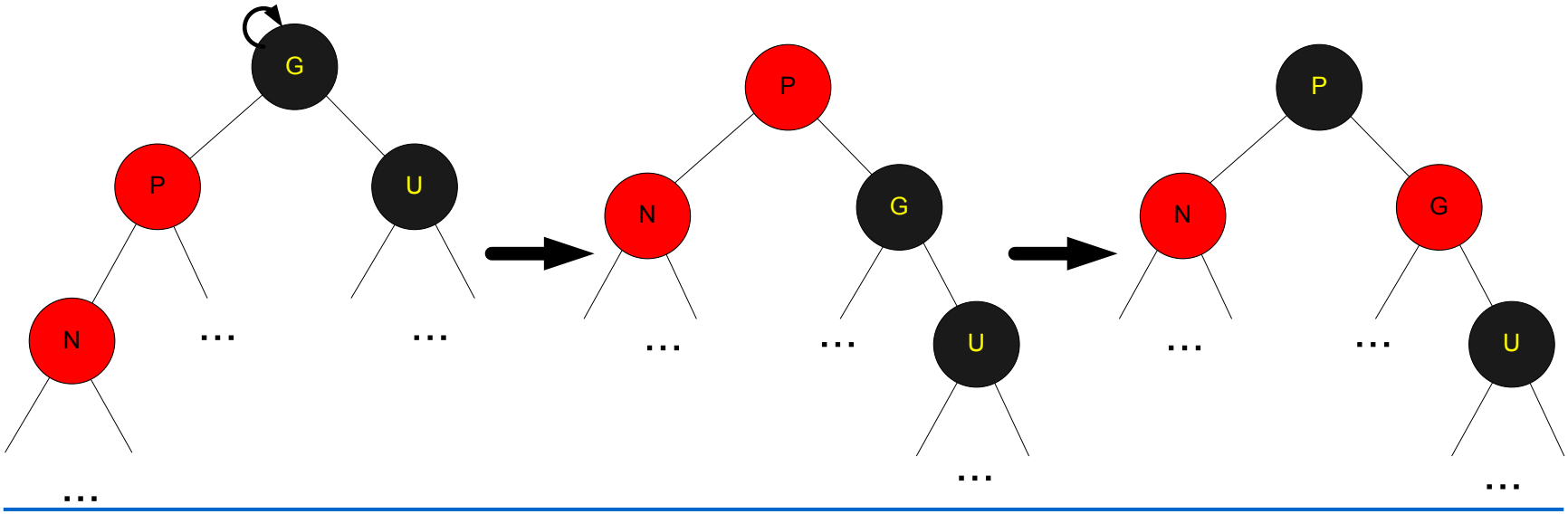
Um die Rot- und Schwarzbedingung einzuhalten müssen in folgenden Fällen Änderungen erfolgen:

1. Fall: Neu eingefügter Knoten ist Wurzel im Baum (falls der Baum vorher leer war) → Knoten muss schwarz gefärbt werden.
2. Fall: Vater ist rot (Verletzung der Rot-Bedingung) und Onkel ist ebenfalls rot: Umfärben des Vaters und des Onkels in Schwarz und des Großvaters in Rot. Problem wurde nun auf den Großvater verschoben → rekursive Behebung des Problems zur Wurzel hin.
3. Fall: siehe nächste Folie

Rot-Schwarz-Bäume: Einfügeoperation II

3. Fall: Der Vater *P* ist rot, aber der Onkel *U* ist schwarz oder nicht vorhanden. Wir nehmen weiterhin an, dass der Vater der linke Sohn des Großvaters *G* ist (ansonsten muss links und rechts in der Beschreibung vertauscht werden).

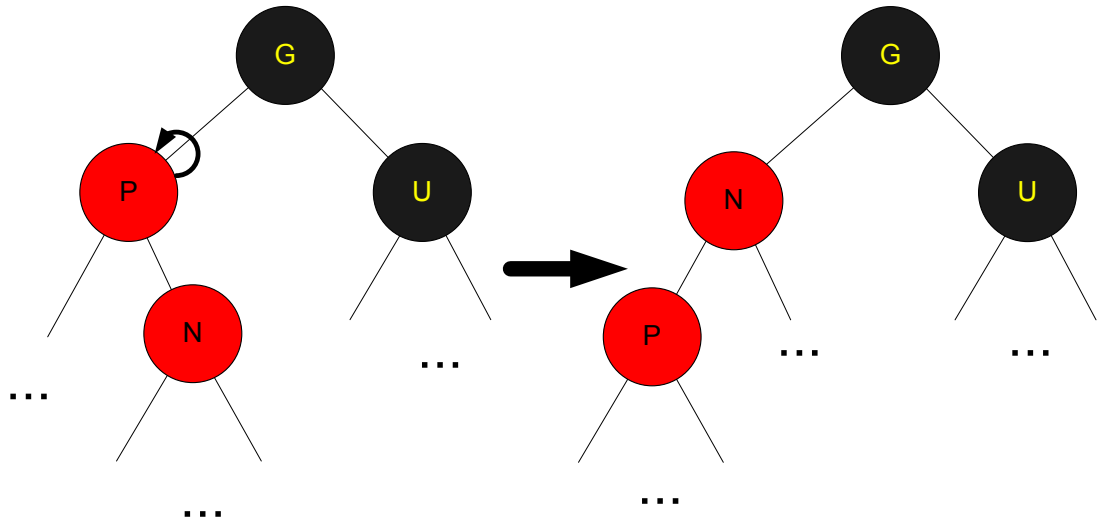
   i. Fall: Der neue Knoten *N* ist linker Sohn des Vaters. Durch Durchführung einer Rechtsrotation wird der Vater nach oben rotiert und mittels Umfärbung des Vaters und des Großvaters werden die Rot- und Schwarz-Bedingungen erfüllt. Alle Operationen haben keinen Einfluss auf darüber liegenden Knoten.



Rot-Schwarz-Bäume: Einfügeoperation III

3. Fall (Fortsetzung)

   ii. Fall: Eingefügter Knoten ist rechter Sohn des Vaters. Durch eine Linksrotation können die Rollen des Vaters und des Sohns getauscht werden und mit dem 3.i.Fall fortgefahren werden.



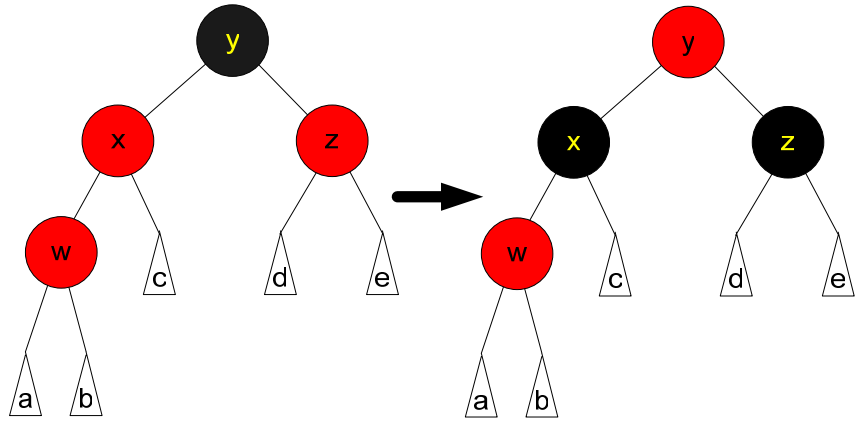
Äquivalenzen der Operationen (2-4-Baum, RB-Baum)

- Grundsätzliche Anmerkungen
  - I. Jeder Knoten des 2-4-Baums führt zu einem schwarzen Knoten im RB-Baum. Als schwarzer Knoten wird die mittlere Verwaltungsinformation gewählt.
  - II. Schwarze Knoten mit roten Nachfolgern entsprechen einem Knoten im 2-4-Baum.
  - III. Jeder rote Knoten muss einen schwarzen Vater haben (der den Knoten im 2-4-Baum repräsentiert.
- Betrachtung der einzelnen Fälle:
  1. Fall: Rote Wurzel ist nicht erlaubt wegen Anmerkung III.
  2. Fall: Knoten, Vater und Onkel sind rot, d.h. der schwarze Großvater repräsentiert einen Knoten mit 5 Kindern, die Reaktion im 2. Fall entspricht also der Aufspaltung des Knotens in 2 Knoten (nun repräsentiert von Vater und Onkel) und nach oben schieben der Verwaltungsinformation (Großvater wird rot)
  3. Fall: Sonderfall, der bei 2-4-Bäumen nicht vorkommt: da der schwarze Knoten immer die mittlere Verwaltungsinformation darstellt, kommt es zu einer Verschiebung.

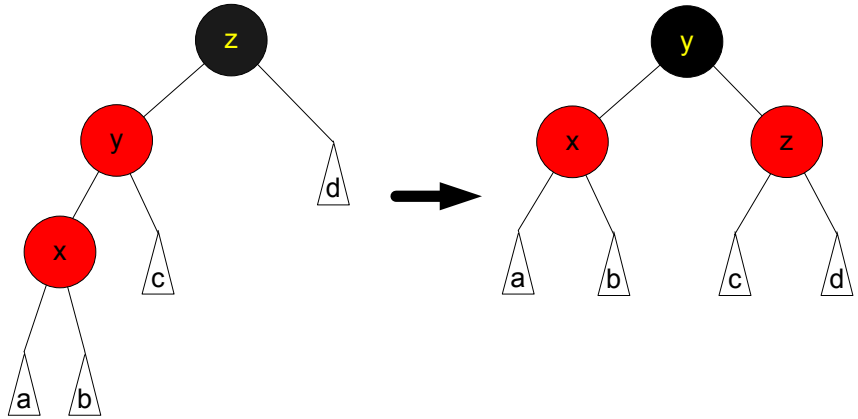
Rot-Schwarz-Bäume: Implementierung (Einfügen) I

Code: Hilfsfunktion zum Balanzieren
(Abdeckung der Fälle 2 und 3)

```
let balanceRB tree = match tree with
    |BlackNode(RedNode(RedNode(a,w,b),x,c),y,
          RedNode(d,z,e)) ->
      RedNode(BlackNode(RedNode(a,w,b),x,c),y,
          BlackNode(d,z,e))  (*coloring*)
```




```
    | BlackNode(RedNode(RedNode(a,x,b),y,c),z,
          d) ->
      BlackNode(RedNode(a,x,b),y,
      RedNode(c,z,d))(*rotation and coloring*)
```



Rot-Schwarz-Bäume: Implementierung (Einfügen) II

Fortsetzung Balancierungsfunktion:

```
| BlackNode(RedNode(a,w,RedNode(b,x,c)),y,RedNode(d,z,e)) ->
      RedNode(BlackNode(a,w,RedNode(b,x,c)),y,BlackNode(d,z,e))  (*coloring*)
| BlackNode(RedNode(a,x,RedNode(b,y,c)),z,d) ->
      BlackNode(RedNode(a,x,b),y,RedNode(c,z,d)) (*double roation and coloring*)
| BlackNode(RedNode(a,w,b),x,RedNode(RedNode(c,y,d),z,e)) ->
      RedNode(BlackNode(a,w,b),x,BlackNode(RedNode(c,y,d),z,e)) (*coloring*)
| BlackNode(a,x,RedNode(RedNode(b,y,c),z,d)) ->
      BlackNode(RedNode(a,x,b),y,RedNode(c,z,d)) (*double roation and coloring*)
| BlackNode(RedNode(a,w,b),x,RedNode(c,y,RedNode(d,z,e))) ->
      RedNode(BlackNode(a,w,b),x,BlackNode(c,y,RedNode(d,z,e))) (*coloring*)
| BlackNode(a,x,RedNode(b,y,RedNode(c,z,d))) ->
      BlackNode(RedNode(a,x,b),y,RedNode(c,z,d)) (*roation and coloring*)
| _ -> tree;;
```

Rot-Schwarz-Bäume: Implementierung (Einfügen) III

Code: insert Funktion:

```
let rec insertRB_embedded elem tree = match tree with
    | EmptyRBTree -> Leaf(elem)
    | Leaf(x) when(x>=elem)-> RedNode(Leaf(elem),elem,Leaf(x))
    | Leaf(x) -> RedNode(Leaf(x),x,Leaf(elem))
    | BlackNode(left,x,right) when (elem<=x) ->
          balanceRB(BlackNode((insertRB_embedded elem left),x,right))
    | BlackNode(left,x,right) when(elem>x)  ->
          balanceRB(BlackNode(left,x,(insertRB_embedded elem right)))
    | RedNode(left,x,right) when (elem<=x) ->
          RedNode((insertRB_embedded elem left),x,right)
    | RedNode(left,x,right) when (elem>x) ->
          RedNode(left,x,(insertRB_embedded elem right))
    | _ -> failwith "invalid argument";;

let insertRB elem tree = match (insertRB_embedded elem tree) with
    | RedNode(left,x,right) -> BlackNode(left,x,right)
    | res -> res;;
```

Funktionale Programmierung

Relationen/Ordnungen

Relationen / Ordnungen

- Wir haben Bäume als geeignete Datenstruktur zum Effizienten Speichern von Elementen kennen gelernt.
- Für das Sortieren benötigen wir aber Mittel zum Vergleich von Elementen einer Menge, z.B. „<“ Operator bei den natürlichen Zahlen.
- Entsprechende Funktionen werden **Relationen** bzw. **Ordnungen** genannt.

Ordnungsrelation über Datumswerte

- Code zum Vergleich von Datumswerten (Tupel von drei Zahlen)

type date = Date of (int*int*int);;

```
let earlier a b = match (a,b) with
    | (Date(day_a,month_a,year_a),Date(day_b,month_b,year_b))
          when (year_a<year_b) ->true
    | (Date(day_a,month_a,year_a),Date(day_b,month_b,year_b))
          when (year_a>year_b) ->false
    | (Date(day_a,month_a,year_a),Date(day_b,month_b,year_b))
          when (month_a<month_b) ->true
    | (Date(day_a,month_a,year_a),Date(day_b,month_b,year_b))
          when (month_a>month_b) ->false
    | (Date(day_a,month_a,year_a),Date(day_b,month_b,year_b))
          when (day_a<day_b) ->true
    | _  ->false;;

earlier (Date(23,4,1986)) (Date(28,4,1990));;
earlier (Date(23,4,1991)) (Date(28,4,1990));;
```

Mathematische Definitionen: Kartesisches Produkt

- **Kartesisches Produkt:**

  Seien *M*,*N* Mengen. Dann heißt

$$M \times N = \{(x, y) : x \in M \wedge y \in N\}$$

  das kartesische (Mengen-) Produkt von *M* und *N*.

- Anmerkung: Die Verallgemeinerung von M_1 x M_2 x … M_n verläuft analog.

Mathematische Definition: Relation

- **Relation**: Eine zweistellige Relation ρ zwischen *M* und *N* ist eine Teilmenge von *M* x *N*.

- **Beispiel**: Die Relation „<“ über die natürlichen Zahlen:

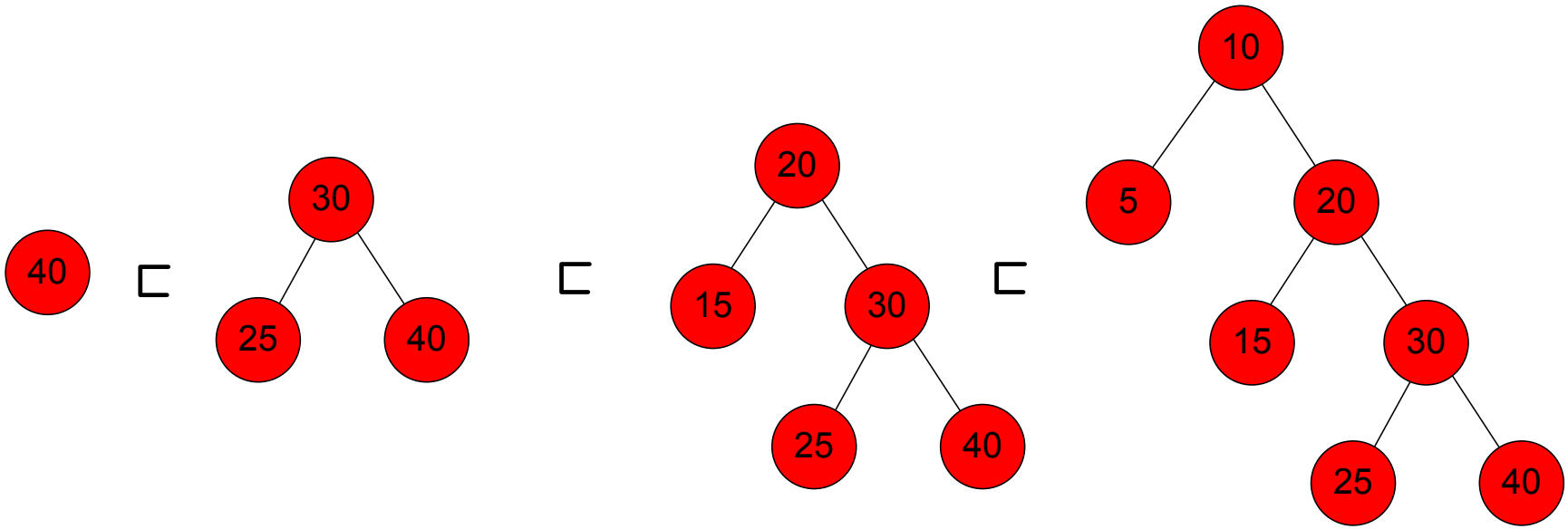
  „<“ = { (1,2),(1,3),(1,4),…
  (2,3),(2,4),(2,5),…
  ….
  }

  oder übersichtlicher:

  1 < 2 < 3 < 4 …

Relationen: weitere Beispiele

- Ebenfalls bereits kennengelernt:
  Teilstrukturrelation $\sqsubset$: (Teilbaum ist jeweils enthalten)



Relationen: weitere Beispiele

- Präfixrelation über Zeichenketten:

  Sei *C* ein Alphabet und *C** die Menge der möglichen Zeichenketten. Dann heißt $z \in C^*$ ein Präfix von x, falls

  $$\exists\, y \in C^*: z \circ y = x$$

  Anschaulich: „ab“ ist ein Präfix von „abc“

Relationen: Eigenschaften

- Eine zweistellige Relation $\rho \in M \times M$ sei gegeben, dann heißt ρ:
  - **reflexiv**, wenn $\forall x \in M: x \rho x$
  - **transitiv**, wenn $\forall x,y,z \in M: (x \rho y) \wedge (y \rho z) \Rightarrow (x \rho z)$
  - **symmetrisch**, wenn $\forall x,y \in M: (x \rho y) \Leftrightarrow (y \rho x)$
  - **antisymmetrisch**, wenn $\forall x,y \in M: (x \rho y) \wedge (y \rho x) \Rightarrow (x=y)$

- $<$ auf $\mathbb{N}$ ist nicht reflexiv, aber transitiv und antisymmetrisch
- $\leq$ auf $\mathbb{N}$ ist reflexiv, transitiv und antisymmetrisch
- <> (ungleich) auf $\mathbb{N}$ ist nicht reflexiv, nicht transitiv und symmetrisch
  - nicht-transitiv: $(1<>2) \wedge (2<>1)$, aber nicht $(1<>1)$
- Die Relation $\rho = \mathbb{N} \times \mathbb{N}$ auf $\mathbb{N}$ ist reflexiv, transitiv und symmetrisch.

Definition: Ordnung

- Eine Relation ρ heißt partielle Ordnung / Halbordnung, falls ρ reflexiv, antisymmetrisch und transitiv ist.
- Eine partielle Ordnung heißt totale Ordnung, falls alle Elemente miteinander vergleichbar sind.

Beispiele:

- Die Relation „$\leq$“ über die natürlichen Zahlen $\mathbb{N}$ ist eine totale Ordnung.
- Die Präfixrelation und die Teilstrukturrelation sind partielle Ordnungen: die Elemente „ab“ und „cd“ sind mit der Präfixrelation nicht vergleichbar weil sie keine Präfixe zueinander sind, aber „ab“ kleiner „abcd“.

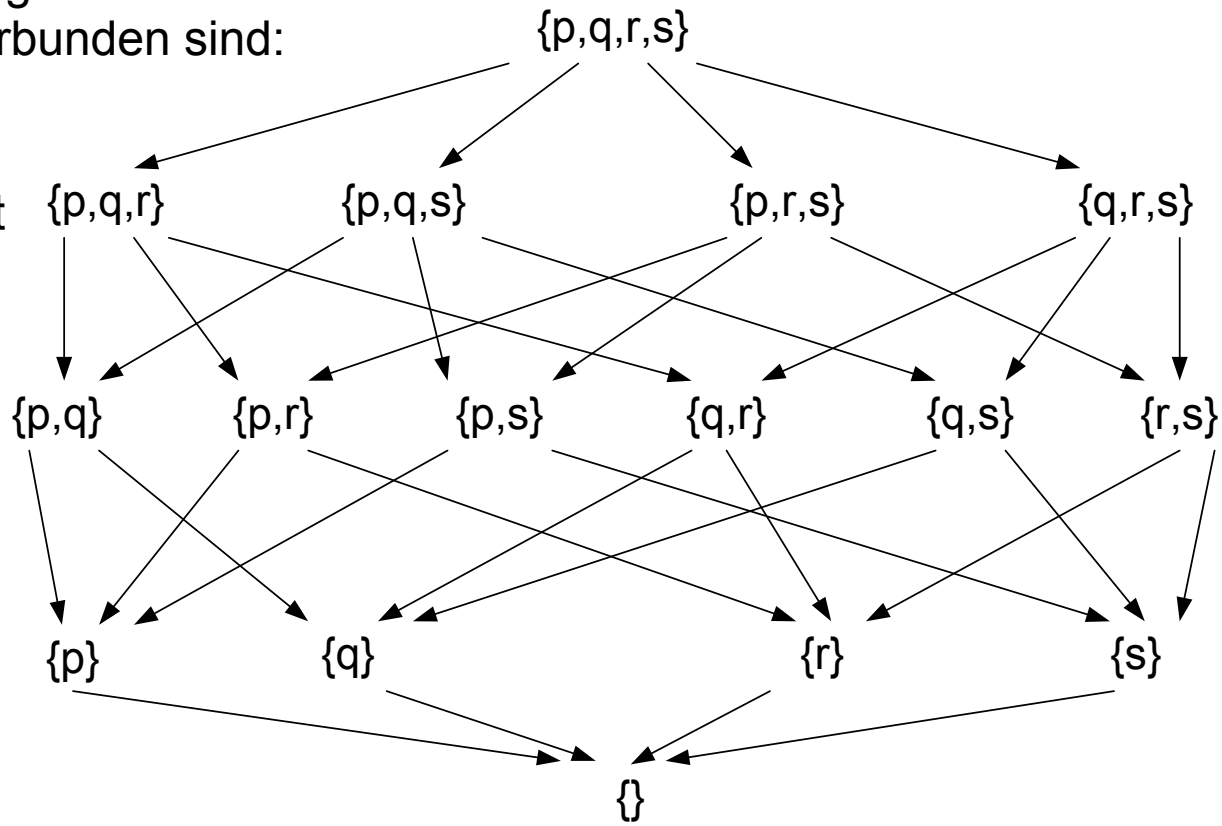
Partielle Ordnung: Teilmengenrelation

Es sind nur Elemente vergleichbar, die durch einen Pfad entlang der Pfeile (Richtung beachten) verbunden sind:

{p,q,r,s} > {p} aber

{p} und {q,r,s} sind nicht

vergleichbar



Definitionen: Kette, Supremum

- Sei M eine Menge und $\sqsubseteq$ eine Halbordnung.
  - Dann ist die Folge (m_1,m_2,..) mit $m_1 \sqsubseteq m_2 \sqsubseteq \ldots$ und $m_i \in M$ eine **Kette**.
- Sei $N \subset M$:
  - x$\in$ M ist **obere Schranke** von *N*, falls gilt: $\forall$ y$\in$ *N*: y$\sqsubseteq$ x.
  - x$\in$ M ist **Supremum** von *N* (*sup* (*N*)), falls x die kleinste obere Schranke von x ist.
- **Vollständige Halbordnung** (englisch: complete partial order cpo) : *M* heißt vollständig geordnet, falls ein Minimum von *M* existiert und für alle Ketten K ein Supremum *Sup*(*K*) existiert.
- Die Teilmengenrelation ist eine vollständige Halbordnung: das Minimum ist die leere Menge. Allerdings ist sie keine totale Ordnung, da nicht alle Elemente miteinander vergleichbar sind.

Ordnung: Weiteres Beispiel

Wir haben bereits weitere Beispiele für Ordnungen kennengelernt:

- Terminierungsbeweis für Ackermannfunktion:
  - Erinnerung:

    Ackermannfunktion:

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

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

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

Beweis der Terminierung über totale Ordnung der Aufrufparameter

$$(0,0) \leq (0,1) \leq (0,2) \leq (0,3) \leq \ldots \leq (1,0) \leq (1,1) \leq \ldots$$

Definition: fundierte Relation / Menge

- Sei ($M, \sqsubseteq$) eine Relation und $N \subset M$. Dann heißt $min \in N$ **minimal** wenn gilt: $\forall\, x \in N$: $(x \sqsubseteq min) \Rightarrow (x = min)$
- Gilt $\forall\, x \in M$: $min \sqsubseteq x$, so ist *min* das **kleinste Element**.
- Eine Relation $\sqsubseteq$ heißt bezüglich einer Menge *M* **fundiert**, falls jede nicht-leere Teilmenge von *M* mindestens ein minimales Element bezüglich $\sqsubseteq$ enthält.
- Eine Menge M mit fundierter Relation $\sqsubseteq$ heißt **fundiert** bezüglich $\sqsubseteq$.
- Beispiel für nicht-fundierte Mengen: ganze Zahlen $\mathbb{Z}$ bezüglich der Relation ≤.
  - Beispiele für Teilmengen, die kein Minimum enthalten
    - Die Menge $\mathbb{Z} \subseteq \mathbb{Z}$.
    - Die Menge der ungeraden Zahlen $Odd \subseteq \mathbb{Z}$.

Definition: Noethersche (fundierte) Induktion

- Mit dem Begriff der fundierten Relation lässt sich auch das allgemeinste Induktionsverfahren definieren: die **Noethersche Induktion**.
- **Satz** (Noethersche Induktion): Sei $(M,\sqsubset)$ eine fundierte Relation. Um eine Eigenschaft P für alle $x \in M$ nachzuweisen, genügt es zu zeigen:

$$\forall\, x \in M\colon (\forall\, y \in M\colon y \sqsubset x \Rightarrow (\mathrm{P}(y) \Rightarrow \mathrm{P}(x)))$$

- ***Beweis***: Angenommen die Menge A der Elemente aus M, für die P nicht gilt, ist nicht-leer. Dann hat A ein kleinstes Element *min*, für das laut Annahme P(*min*) nicht gilt. Nach Definition gilt aber $P(y)$ für alle $y \sqsubset min$ und damit muss nach Voraussetzung auch $P(min)$ gelten, im Widerspruch zur Annahme. Somit muss A leer sein, d.h. P gilt für alle $x \in M$.

Induktion: Übersicht

Noethersche Induktion

fundierte Relation

z.B. Terminierungsbeweis Ackermann

vollständige Induktion

Relation „<“ über $\mathbb{N}$

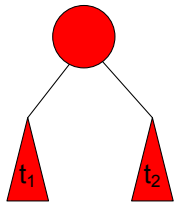
strukturelle Induktion

Teilstrukturrelation

$\sqsubseteq$

Beispiel: Strukturelle Induktion über Bäumen

- Zu zeigen: Ein Binärbaum (mindestens ein Knoten, jeder Knoten null oder zwei Nachfolger) besitzt mehr Blätter als innere Knoten.
- Induktionsanfang:  #Blatt = 1 > 0 = #InnereKnoten
- Induktionsschritt: Baum t:



- Induktionsvoraussetzung: Für alle Teilbäume von t gilt die Behauptung.

  $\Rightarrow$ #Blatt(t_1)> #InnereKnoten(t_1) $\wedge$ #Blatt(t_2)> #InnereKnoten(t_2)

  Umformulierung der Induktionsvoraussetzung:

  $\Rightarrow$ #Blatt(t_1)$\geq$ #InnereKnoten(t_1)+1 $\wedge$ #Blatt(t_2) $\geq$ +1 #InnereKnoten(t_2)

  Rückschluss auf Baum t von Teilstrukturen:

  $\Rightarrow$ #Blatt(t)= #Blatt(t_1) + #Blatt(t_2) $\geq$
  #InnereKnoten(t_1)+#InnereKnoten(t_2)+2>
  #InnereKnoten(t_1)+#InnereKnoten(t_2)+1 > #InnereKnoten(t)

Funktionale Programmierung

Funktionen höherer Ordnung

Motivation

Wir kennen bereits:

- Effiziente Algorithmen: z.B. Sortieren
- Geeignete Datenstrukturen: z.B.: Bäume, Listen
- Polymorphe Datenstrukturen und Funktionen
- Ordnungen: Möglichkeiten zum Vergleich von Elementen einer Liste

Problem:

- Bisher verwenden wir in allen Algorithmen über unsere Datenstrukturen den Vergleichsoperator „<“
- Dieser Operator ist für verschiedene Mengenelemente nicht in Ocamml vordefiniert.

Ziel:

- Einmalige Programmierung der Datenstrukturen und Algorithmen für beliebige Mengen

Funktionen höherer Ordnung I

- Funktionen, die Funktionen als Parameter tragen oder Funktionen zurückgeben, heißen **Funktionale** oder **Funktionen höherer Ordnung**
- Beispiel: Im Rahmen von Quicksort (Folie 172) haben wir bereits die filter-Funktion kennengelernt. Diese Funktion filtert aus einer Menge von Elementen genau die Elemente, die ein bestimmtes Prädikat erfüllen.

```
let rec filter_lower pivot list= match list with
   | [] -> []
   | hd::tail when (hd <= pivot ) -> hd::(filter_lower pivot tail)
   | hd::tail -> filter_lower pivot tail;;
let rec filter_higher pivot  list = match list with
   | [] -> []
   | hd::tail when (hd > pivot ) -> hd::(filter_higher pivot
   tail)
   | hd::tail -> filter_higher pivot  tail;;
```

Funktionen höherer Ordnung: filter Funktion I

- Allgemeine Implementierung der Filterfunktion:

```
let rec filter pred list= match list with
   | [] -> []
   | hd::tail when(pred hd) -> hd::(filter pred tail)
   | hd::tail -> filter pred tail;;

let positiveorzero number = (number>=0);;
let negative number = (number<0);;

let list = 13::-2::2165::25::-58::-436::[];;

filter positiveorzero list;;
filter negative list;;
```

Funktionen höherer Ordnung: filter Funktion II

- Es ist auch möglich neben der Funktion weitere Argumente zu übergeben

```
let rec filter pred list= match list with
   | [] -> []
   | hd::tail when(pred hd) -> hd::(filter pred tail)
   | hd::tail -> filter pred tail;;

let higher pivot elem= (pivot<elem);;

let list = 13::-2::2165::25::-58::-436::[];;
filter (higher 25)list;;
```

Funktionen höherer Ordnung: map Funktion

- Weiteres sinnvolles Anwendungsbeispiel für Funktionen höherer Ordnung: die map Funktion
- Die map Funktion wendet eine Funktion auf alle Elemente einer Liste an:

```
let rec map func list= match list with
   | [] -> []
   | hd::tail -> (func hd)::(map func tail);;

let negate number = -number;;
let increment number = number+1;;

let list = 13::-2::2165::25::-58::-436::[];;
map negate list;;
map increment list;;
```

Funktionen höherer Ordnung: fold Funktion

- Häufig soll ein binärer Operator auf eine Liste angewandt werden: z.B. Summe, Produkt, Minimum, Maximum aller Element
- Falls der Operator nicht assoziativ ist, ist das Ergebnis von der Anwendungsrichtung abhängig: Anwendung von links nach rechts (foldl) oder von rechts nach links (foldr)

```
let rec foldr func list = match list with
   | x::[] -> x
   | hd::tail -> func hd (foldr func tail);;

let sum a b = a+b;;
let mult a b = a*b;;
let max a b = if(a>b) then a else b;;
let min a b = if(a<b) then a else b;;

let list = 1::2::3::4::5::6::[];;
foldr sum list;;
foldr mult list;;
foldr max list;;
foldr min list;;
```


LASST
EUCH
NICHT
TYPISIEREN
VOR ALLEM NICHT
BEIM KNOLL
MatchMarkt
Ich bin doch nicht rekursiv.


http://www.jast-net.gr/project.htm
Erste Schritte
Aktuelle Nachrichte...
JAST
Joint-Action
Science and Technology
Conference
Project Description
Objectives
Potential Impact
Envisioned Results
Work Packages
Partners
Contact Info
PROJECT DESCRIPTION
The success of the human species critically depends on our extraordinary ability to engage in joint action. Our perceptions, decisions and behaviour are tuned to those of others with whom we share beliefs, intentions and goals and thus form a group.
These insights underlie the motivation of the JAST project to develop jointly-acting autonomous systems that communicate and work intelligently on mutual tasks in dynamic unstructured environments. A goal that is far-reaching beyond studying individual cognitive systems and that will expand the concept of 'group' to 'human plus artificial agent(s)'.
The interdisciplinary effort of JAST will broaden the scientific basis of cognitive science. As a result, JAST will influence almost any field building upon concepts and results of cognitive science, like "ambient intelligence" and "human-computer interaction". The project will initiate a new way of thinking in cognitive science and develop beyond state-of the-art autonomous systems.
JAST will build cognitive systems that will be "socially aware", which will build trust and confidence in technology and finally, the ultimate tools that will result from the project will be applicable in industry and society.
Designed & Developed: OTEnet A.E. Copyright © 2004 JAST All rights reserved

Imperativer Programmierstil

Vorbemerkung I: Programmierparadigmen

- Paradigma: "Denkmuster"
- Funktional/applikativ: Anwendung von Funktionen auf Werte, kein Begriff des "Zustands"
- Imperativ/Prozedural: Instruktionen verändern den globalen Zustand des Programms ($\rightarrow$ änderbare Variable, Mehrfachzuweisung)
- Objektorientiert: Datenobjekte bilden Objekte in der realen Welt ab. Objekte senden sich gegenseitig Nachrichten, die ihren inneren Zustand ändern
- Deklarativ: Beschrieben wird das Problem, nicht der Lösungsweg

Vorbemerkung II: Historie der Programmiersprachen

Vorläufer, Vorgänger	Jahr	Name	Entwickler, Hersteller
*	1840~	erstes Programm	Ada Lovelace
*	1946	Plankalkül	Konrad Zuse
*	1952	A-0	Grace Hopper
*	1954	Mark I Autocode	Tony Brooker
A-00	1954	FORTRAN	John Backus
A-0	1955	FLOW-MATIC	Grace Hopper
*	1957	General Problem Solver	Allen Newell
*	1958	Algol 58	
FORTRAN	1958	FORTRAN II	
*	1959	LISP	John McCarthy
FLOW-MATIC, FACT, COMTRAN	1960	COBOL	Grace Hopper, CODASYL
Algol 58	1960	Algol 60	John Backus, Peter Naur
FORTRAN II	1962	FORTRAN IV	
*	1962	APL	Kenneth E. Iverson
Algol 58	1964	JOSS	
FORTRAN, Algol 60	1964	PL/1	IBM
Algol	1965	Simula	Ole-Johan Dahl und Kristen Nygaard bei Norsk Regnesentral
Fortran	1965	BASIC	John George Kemeny, Thomas Eugene Kurtz
FORTRAN IV	1966	FORTRAN 66	
LISP	1966	Logo	Seymour Papert
	1967	MUMPS	Massachusetts General Hospital
COWSEL	1968	POP-1	Rod Burstall, Robin Popplestone
*	1968	REFAL	Valentin Turchin
Simula	1967	Simula 67	
Algol 60	1968	Algol 68	Adriaan van Wijngaarden, Koster, Mailloux, Peck
CPL	1969	BCPL	Martin Richards
	1969	PILOT	
POP-1	1970	POP-2	
BCPL	1970	B	Ken Thompson
*	1970	Forth	Charles H. Moore
Algol, SNOBOL 4	1970	Icon	University of Arizona
Algol 60	1971	Pascal	Niklaus Wirth, Jensen
SIMULA 67	1972	Smalltalk 72	Xerox PARC
B, BCPL, Algol	1972	C	Dennis Ritchie

Quelle: Wikipedia

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 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)

Imperativer Programmierstil: Änderbare Datentypen II

– 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 III

– **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"};;
```

Imperativer Programmierstil: Änderbare Datentypen IV

```
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!

Imperativer Programmierstil: Kontrollstrukturen III

- **Und**: **Iteration** oder WHILE-Schleife (abweisende Schleife),
  `while expr1 do expr2; update_expr1 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 = ()
```

Hinweis: ref 1 ist ein sogenannter Zugriffstyp (Zeiger, pointer), siehe OCAML-Manual

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`**

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 = ()
```

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`**

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 Ausnahmebehandler „**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: 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.

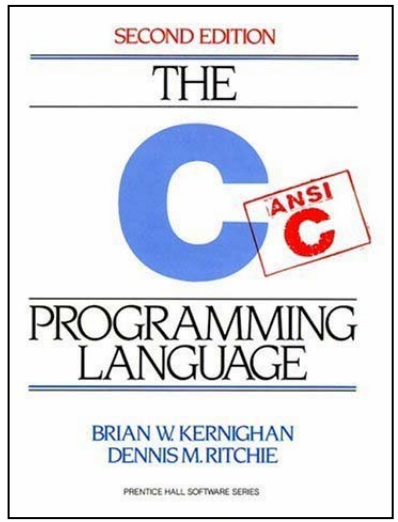
– Damit wird der vollständig funktionale Bereich verlassen, denn die Auswertungsreihenfolge spielt jetzt u.U. doch eine Rolle!

```
# 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;;
```

Imperative Strukturen in "C"

- Einfache Programmiersprache, entworfen u.a. zur Programmierung des Betriebssystems "UNIX" auf Maschinen von Digital Equipment
- http://www.open-std.org/jtc1/sc22/wg14/
- Buch (von den Sprachentwicklern):
  Brian W. Kernighan, Dennis Ritchie
  Zweite Auflage
  Prentice Hall, 1988



Imperative Strukturen in "C"

- Ende der sechziger Jahre wurden Minirechner wie PDP-11 mit Assembler oder in FORTRAN programmiert
- Wünschenswert war eine problemorientierte Sprache der Ausdrucksmächtigkeit von Algol, aber mit direktem Maschinenzugriff, sowie übersetz- und ausführbar auf Minirechner.
- Folge von Versuchen:
  - CPL (Combined Programming Language), 1963
  - BCPL (Basic Combined Programming Language; 1967), im wesentlichen sog. Makroassembler (der in Teilen in C bis heute überlebt hat)
  - B (1969), weiter vereinfachtes BCPL, nur Maschinenwort als Datentyp, erste Version von UNIX auf DEC PDP-7
- C (1972), Programmierung von UNIX in den Bell-Laboratories (AT&T), wurde beeinflußt vom PDP-11-Befehlssatz bzw. Makroassembler

Ritchie und Thompson an einer PDP-11

2.3.1 General Registers

The general registers can be used for a variety of purposes; the uses varying with requirements. The general registers can be used as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register.

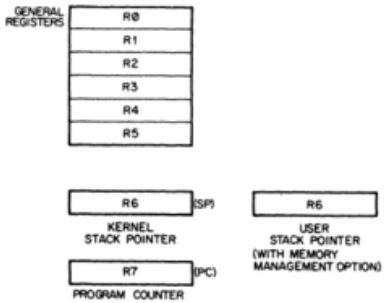
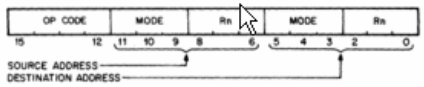


Figure 2-1 The General Registers

R7 is used as the machine's program counter (PC) and contains the address of the next instruction to be executed. It is a general register



The source address field is used to select the source operand, the first operand. The destination is used similarly, and locates the second operand and the result. For example, the instruction ADD A, B adds the contents (source operand) of location A to the contents (destination operand) of location B. After execution B will contain the result of the addition and the contents of A will be unchanged.

Examples in this section and further in this chapter use the following sample PDP-11 instructions:

Mnemonic	Description	Octal Code
CLR	clear (zero the specified destination)	0050DD
CLRB	clear byte (zero the byte in the specified destination)	1050DD
INC	increment (add 1 to contents of destination)	0052DD
INCB	increment byte (add 1 to the contents of destination byte)	1052DD
COM	complement (replace the contents of the destination by their logical complement; each 0 bit is set and each 1 bit is cleared)	0051DD
COMB	complement byte (replace the contents of the destination byte by their logical complement; each 0 bit is set and each 1 bit is cleared).	1051DD
ADD	add (add source operand to destination operand and store the result at destination address)	06SSDD

DD = destination field (6 bits)

SS = source field (6 bits)

( ) = contents of

3.3 DIRECT ADDRESSING

The following table summarizes the four basic modes used with direct addressing.

DIRECT MODES

Mode	Name	Assembler Syntax	Function
0	Register	Rn	Register contains operand
2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data then incremented
4	Autodecrement	-(Rn)	Register is decremented and then used as a pointer.
6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) are modified.

3.3.1 Register Mode

OPR Rn

With register mode any of the general registers may be used as simple accumulators and the operand is contained in the selected register. Since they are hardware registers, within the processor, the general registers operate at high speeds and provide speed advantages when used for operating on frequently-accessed variables. The PDP-11 assembler interprets and assembles instructions of the form OPR Rn as register mode operations. Rn represents a general register name or number and OPR is used to represent a general instruction mnemonic. Assembler syntax requires that a general register be defined as follows:

R0 = %0 (% sign indicates register definition)

R1 = %1

R2 = %2, etc.

Registers are typically referred to by name as R0, R1, R2, R3, R4, R5, R6 and R7. However R6 and R7 are also referred to as SP and PC, respectively.

Register Mode Examples
(all numbers in octal)

	Symbolic	Octal Code	Instruction Name
1.	INC R3	005203	Increment
Operation:		Add one to the contents of general register 3	

Imperative Strukturen in "C"

- Einfachstes Programm:

```
#include <stdio.h>
   main() {
        printf(“Hello, world.\n”);
   }
```

- # leitet Kommando an *C-Präprozessor* ein, d.h. hier: die Zeile mit #include-Anweisung wird durch den Inhalt der Datei "stdio.h" ersetzt
- *Header files* (.h) sind Schnittstelle der eigentlichen Programmdatei (.c) zur Außenwelt
- "main" () ist (Beginn des) Hauptprogramms
- Printf ("Print formatted") wird benutzt wie vorher bei OCAML beschrieben

Imperative Strukturen in "C"

- Etwas komplizierteres Beispiel: Erzeuge Tabelle von Fahrenheit-Celsius-Werten (C = 5/9 * (F-32)) mit while-Schleife

```
#include <stdio.h>
main() {
   int fahr, cels, lwr, upr, step;
   lwr = 0; upr = 300;

   fahr = lwr;
   while (fahr <= upr) {
        cels = 5*(fahr-32)/9;          /* Wegen Rundung bei int */
        printf("%d\t%d\n", fahr, cels);
        fahr = fahr + step;
   }
}
```

Imperative Strukturen in "C"

- Prinzipiell sollten immer alle Variablen *initialisiert* werden, da C im Gegensatz zu anderen Programmiersprachen (z.B. Java) Variablen nicht bei der Deklaration automatisch mit einem Standardwert initialisiert.
- Bei der Deklaration wird nur der Speicherplatz für die Variable reserviert, der (zufällige) Inhalt des Speicherplatzes bleibt unverändert.
- Die Ausführung eines Programmes kann damit vom Zustand des Rechnersspeichers abhängig sein.

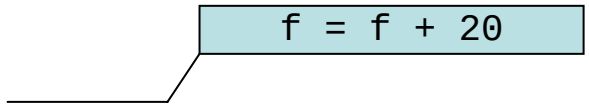
```
#include <stdio.h>
 main() {
    int fahr, cels, lwr, upr, step;
    lwr = 0; upr = 300; step=5;

    fahr = lwr;
    while (fahr <= upr) {
      cels = 5*(fahr-32)/9;                    /* Wegen Rundung bei int */
      printf("%d\t%d\n", fahr, cels);
      fahr = fahr + step;
    }
 }
```

Imperative Strukturen in "C"

- Beispiel Umwandlung Fahrenheit-Celsius mit FOR-Schleife:

```
main() {
      int f;
      for (f = 0; f <= 300; f += 20)
                printf("%3d %6.1f\n", f, (5.0/9)*(f-32));
}
```



- `for (initial_value; test; increment) {}` entspricht:

```
init;
while (test) {
      /* Schleifenrumpf */
      inc;
}
```

Imperative Strukturen in "C"

- Selektion I: If...then...else

- ```
  if (year == 2005)…;
  else if (year == 2006)…;
  ```
  ```

Imperative Strukturen in "C"

- Selektion II: switch-statement
-
```
switch (expr) {
    case val1 : …;
                break;
    case val2 : …;
                break;
    …
    default: …;
    /* optional */
}
```

-
```
switch (year) {
    case 2005: …;
               break;
    case 2006: …;
               break;
    …
    default: …;
}
```

Imperative Strukturen in "C": Felder (Arrays)

```
#include <stdio.h>
/* count digits, white space, and others */
int main(int argc, char *argv[])
{
   int c, i, nwhite, nother;
   int ndigit[10];
   FILE *input;
   /* initialize */
   nwhite = nother = 0;
   for (i=0; i<10; i++) ndigit[i] = 0;
   /* scan input */
   if(argc==1){
          printf("Please specify input
    file");
          return -1;
   }
   input=fopen(argv[1],"rt");
 if(input==NULL){
          printf("invalid file name");
          return -1;
   }
c = fgetc(input);
   while (c != EOF){
      if (c >= '0' && c <= '9')
         ++ndigit[ c-'0' ];
      else if (c == ' ' || c == '\n' || c == '\t')
         ++nwhite;
      else
         ++nother;
      c=fgetc(input);
   }
   printf("digit = ");
   for (i=0; i<10; i++) printf(" %d", ndigit[i]);
   printf(", white space = %d, other = %d\n",
          nwhite, nother);
   return 0;
}
```

Imperative Strukturen in "C" : Records

- "Records" oder "Verbünde" heißen in C `struct`
- 1. Schritt: Typdefinition (legt noch keine Variable an):

```
struct Student
{
    char family_name[MAXNAME];
    char first_name[MAXNAME];
    int social_security_number;
};
```

2. Schritt: Anlage der Variablen

```
struct Student sarah;
```

Imperative Strukturen in "C" : Records

3. Schritt: Zugriff auf die einzelnen Felder über Punktnotation:

```
struct Student sarah;
    sarah.family_name
    sarah.first_name
    sarah.social_security_number
```

Also z.B.:

```
    sarah.social_security_number = 2315877;
```

Zeiger in C

- Generell haben wir folgende Datentypen kennengelernt:
  - einfache Datentypen: int, float, char, …
  - Felder (Arrays) von einfachen Datentypen
  - Strukturen (Records, Verbünde): Datentypen, die aus einfachen Datentypen zusammengesetzt werden.
- Neben diesen Datentypen bietet C noch den Typ *Zeiger* an:
  - Zeiger (Pointer, access types) sind Variablen, die anstatt Werten (Speicher-) Adressen beinhalten.
  - An der Speicheradresse ist ein anderes Objekt bzw. der Wert einer anderen Variablen abgespeichert.
- Vorteil: Zeiger erlauben die flexible Nutzung von Speicherplätzen, es können Datenstrukturen (Listen, Bäume) angelegt werden, die zur Laufzeit wachsen (dynamische Datenstrukturen).
- Nachteil: Speicherzugriffsfehler (unbeabsichtigte Änderungen von anderen Daten) können auftreten, da Zeiger nicht mit einer bestimmten Größe des beschriebenen Datenbereichs verbunden sind.
- In C ist der Wert einer Variablen nach der Deklaration unbestimmt. Um zu vermeiden, daß ein Zeiger auf beliebige Datenbereiche zeigt, kann ihn mit NULL, dem Nullzeiger, initialisieren.
- Zugriffe auf NULL führen immer automatisch zu einem Fehler. Wird NULL nicht verwendet, kann es passieren, daß unbeabsichtigt andere Datenbereiche überschrieben werden.

Zeiger in C: Syntax

- Eine Variable vom Datentyp Zeiger kann per '*'-Syntax deklariert werden.

```
int *pointer_to_int;

char *pointer_to_char;
```

- Beachte: Mit der Deklaration eines Zeigers wird nur der Speicherplatz für den Zeiger, nicht aber der Platz für das referenzierte Objekt angelegt (d.h. der Speicherplatz, auf den gezeigt wird).
- Um mit den Zeiger auf bereits existierende Objekte zu zeigen, kann der Adreßoperator & benutzt werden. `&var` ist eine Funktion, die die Speicheradresse zurückgibt, an der `var` steht.

```
pointer_to_int = &b;
```

- Zum Zugriff auf das referenzierte Objekt steht der Dereferenzierungsoperator * zur Verfügung.

```
b = *pointer_to_int;

*pointer_to_int = 154;
```

Zeiger in C: Syntax

Beispielcode:

```
#include <stdio.h>
int main()
{
   int *pointer_to_int = (int*) NULL;
   int content = 20;
   pointer_to_int = &content;
   printf("An der Adresse %d beginnend steht der Wert %d im
         Speicher\n", pointer_to_int, content);
   *pointer_to_int = 10;
   printf("Nach der Änderung steht nun beginnend an der Adresse
        %d der Wert %d im Speicher\n", pointer_to_int, content);
   return 0;
}
```

Zeiger: Syntax in anderen Programmiersprachen

- Zeigerdeklaration:
  - Pascal: VAR Pointer_To_Int : **^**Integer;
  - Modula:

    TYPE IntegerPointer = POINTER TO INTEGER;

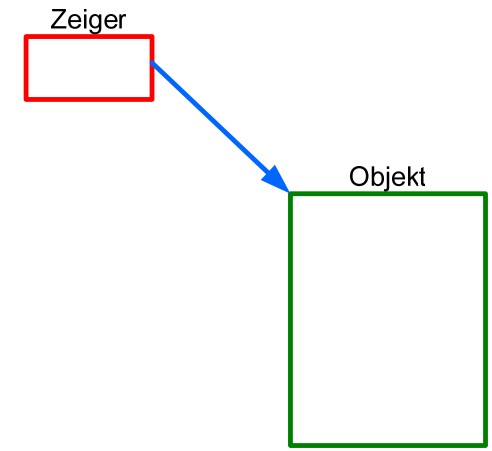
    VAR my_int_pointer: IntegerPointer;
- Dereferenzierung:
  - Pascal: Pointer_To_Int^
  - Modula: my_int_pointer^
- Adreßoperator:
  - Pascal: Addr(Object) oder @Object (nicht im Standard-Sprachumfang)
  - Modula: ADR(Object) (im Modul SYSTEM)

Zeiger in C: Anlegen von neuem Speicher

- Zum Anlegen eines neuen Speicherbereichs kann die Funktion `malloc()` benutzt werden.

```
int *pointer_to_int =
        (int*) malloc (100*sizeof(int));
```

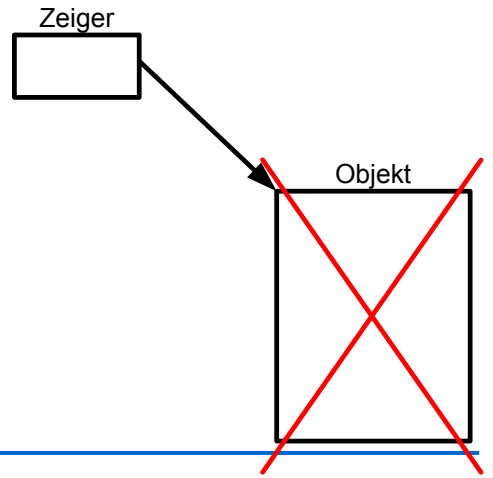
  - Die Funktion `malloc()` gibt einen Zeiger auf den unbestimmten Datentyp (`void*`) zurück. Um Warnungen des Compilers zu vermeiden, muß das Ergebnis angepaßt werden (*type casting*). Dies wird durch `(int*)` erreicht.
  - Als Argument bekommt `malloc()` die Größe des anzulegenden Speicherbereichs in Byte übergeben. Die Funktion `sizeof()` hilft bei der Bestimmung des benötigten Speicherplatzes (im Beispiel 100-mal der Speicherbedarf von `int`) .
  - Ist nicht genügend Speicherplatz vorhanden, so liefert die Funktion einen NULL-Zeiger zurück.



- Mittels der Funktion free() kann angelegter Speicher wieder freigegeben werden.

```
free(pointer_to_int);
```

  Achtung: es wird nur der mit dem Zeiger assoziierte Speicherbereich freigegeben, die Variable/der Zeiger selber kann weiter benutzt werden.



Anlegen/Freigeben von Speicher

- Beispielcode

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pointer_to_int = NULL;
   int content;

   pointer_to_int=(int*) malloc(sizeof(content));
   *pointer_to_int=20;
   content=*pointer_to_int;
   printf("An der Adresse %d beginnend steht der Wert %d im
             Speicher\n",pointer_to_int,content);
   free(pointer_to_int);
   return 0;
}
```

Lebensdauer von Variablen

- Mit der Deklaration einer Variablen wird ein Objekt im Speicher erzeugt, auf das mit dem Namen der Variablen zugegriffen werden kann.
- Wie lange diese Objekte verfügbar und existent sind, hängt von verschiedenen Faktoren ab:
  - Ort der Deklaration: Variablen, die in einem Blockbereich `{…}` definiert werden, existieren nur bis zum Ende dieses Bereichs und werden bei Verlassen des Blocks aufgelöst. Variablen, die global deklariert werden, existieren bis zum Programmende (siehe dazu die "Verschattungsregeln" in OCAML).
  - Art der Speicherreservierung: Wird ein Speicherplatz durch die Funktion `malloc()` angelegt, so existiert dieser Speicherplatz so lange, bis `free()` durch den Programmierer aufgerufen wird oder das Programm beendet wird.
- **Gefahrenquelle 1:** Wird eine Zeigervariable innerhalb eines Blockbereichs deklariert und ebenso ein Speicherbereich mit `malloc()` angelegt und der Zeigervariablen zugewiesen, so existiert nach Beendigung des Blockbereichs zwar der neu angelegte Speicherbereich, allerdings nicht mehr der darauf verweisende Zeiger.

  ⇒ Alle Datenbereiche, die mit `malloc()` innerhalb eines Blockbereichs reserviert werden, müssen auch wieder freigegeben werden
- **Gefahrenquelle 2:** Zeigt eine außerhalb eines Blockbereichs deklarierte Zeigervariable auf ein innerhalb dieses Blockbereichs statisch deklariertes Objekt, so existiert zwar nach Ende des Blockbereichs noch der Zeiger, jedoch zeigt dieser auf einen ungültigen, weil mittlerweile freigegebenen Bereich.

  ⇒ Einer Zeigervariable außerhalb eines Blockbereichs sollte nie die Adresse einer lokalen Variable zugewiesen werden.

Syntax von Zeigern in Bezug auf Records

- Zeigt der Zeiger auf einen Record-Datentyp, so kann anstelle der Dereferenzierung mittels * und dem Zugriffsoperator . der Zugriffsoperator -> verwendet werden.

```
struct Student
{
     char family_name[MAXNAME];
     char first_name[MAXNAME];
     int social_security_number;
};
```

```
struct Student *pointer_to_student=(struct Student*) malloc(sizeof(struct Student));
```

```
(*pointer_to_student).social_security_number=2315877;
```
entspricht
```
pointer_to_student->social_security_number=2315877;
```

Verwendung von Zeigern (1): Arrays

- Felder/Arrays können wie Zeiger behandelt werden. Vereinbarung: In der Zeiger-Variablen ist die Adresse des Beginns des Speicherplatzes-Bereichs gesichert, in dem die Feldelemente gespeichert sind.
- Ein Nachteil von Feldern ist die notwendige Festlegung der Feldgröße vor der Übersetzung. Durch die Verwendung von Zeigern und `malloc()` ist auch eine Festlegung der Array-Größe zur Laufzeit möglich.
- Der Zugriff auf die einzelnen Elemente kann wie bei Feldern mit dem Zugriffsoperator `[]` erfolgen. Alternativ kann zur Zeigeradresse ein so genanntes *Offset* addiert werden. Der Effekt des Offsets richtet sich auch nach dem verwendeten Datentyp (siehe Beispiel).

Verwendung von Zeigern (1): Arrays

Beispiel für Zugriffsoperatoren bei Feldern:

```
#include <stdio.h>
int main()
{
    int content0,content1;
    int *pointer_to_int_array=(int*) NULL;
    pointer_to_int_array=(int*) malloc(5*sizeof(int));
    pointer_to_int_array[0]=0;
    *(pointer_to_int_array+1)=1;
    content0=*(pointer_to_int_array+0);
    content1=pointer_to_int_array[1];
    printf("Der erste Wert des Arrays ist %d, der zweite
          %d\n",content0,content1);
    free(pointer_to_int_array);
    return 0;
}
```

Wieso funktioniert die Adressierung des nächsten Elements mit +1 und nicht mit +sizeof(int)?

Offset bei Zeigern I

- Beispielcode:

```
#include <stdio.h>
int main()
{
   int *pointer_to_int_array = (int*) malloc(5*sizeof(int));
   int i;
   for(i=0;i<5;i++)
        pointer_to_int_array[i]=0;
   *(pointer_to_int_array+1)=1;
   *(pointer_to_int_array+sizeof(int))=1;
   for(i=0;i<5;i++)
        printf("%d\n",pointer_to_int_array[i]);
   return 0;
}
```

Ausgabe:
0
1
0
0
1

Offset bei Zeigern II

Beispiel für Offsets bei Zeigern:

```
#include <stdio.h>
int main()
{
    char string[3]="ab";
    char *pointer_to_char=string;
    int int_array[2]={1,2};
    int *pointer_to_int=int_array;

    printf("An Adresse %d steht der Wert %c\n",pointer_to_char,*pointer_to_char);
    printf("An Adresse %d steht der Wert %c\n",pointer_to_char+1,*(pointer_to_char+1));
    printf("An Adresse %d steht der Wert %d\n",pointer_to_int,*pointer_to_int);
    printf("An Adresse %d steht der Wert %d\n",pointer_to_int+1,*(pointer_to_int+1));
    printf("Benötiger Speicherplatz für char: %d\n",sizeof(char));
    printf("Benötiger Speicherplatz für int: %d\n",sizeof(int));
    return 0;
}
```

Offset bei Zeigern III

- Bildschirmausgabe bei Programmausführung:

```
An Adresse 2289408 steht der Wert a
An Adresse 2289409 steht der Wert b
An Adresse 2289392 steht der Wert 1
An Adresse 2289396 steht der Wert 2
Benötiger Speicherplatz für char: 1
Benötiger Speicherplatz für int: 4
```

Felder von Zeigern

- Es ist auch möglich Felder von Zeigern zu verwenden.
- Anwendungsbeispiel: Funktion `main`

```
int main(int argc, char *argv[])
```

- Mittels dieser Syntax ist es möglich, `main` verschiedene Argumente (als `char`-Felder) mittels der Kommandozeile zu übergeben.
- `argc` enthält die Anzahl der Argumente (inklusive Dateiaufruf), `argv` die Argumente.
- Die Mitteilung der Anzahl der Argumente ist nötig, weil bei einem Zeiger nicht die Anzahl der Elemente im referenzierten Objekt bestimmbar ist.

Arrays von Zeigern

- Beispielcode:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
   printf("Folgende Parameter wurden dem Programm
            übergeben:\n");
  for(i = 0; i < argc; i++)
  {
    printf("Argument %d : %s \n",i,argv[i]);
  }
  return 0;
}
```

Verwendung von Zeigern(2): Funktionen mit Zeigern als Parametern

- Die Parameter beim Funktionsaufruf können auf verschiedene Weisen übergeben werden:
  - **Call-by-value:** Beim Funktionsaufruf werden die Argumente kopiert. Innerhalb der Funktion bestehen diese Argumente also als eigenständige Objekte und eine Veränderung der Argumente kann von außen nicht gesehen werden. Call-by-value in C kann durch einfache Übergabe der Objekte erreicht werden z.B. `ggt(a,b)`
  - **Call-by-Reference**: Beim Funktionsaufruf werden Zeiger anstelle der Objekte übergeben. Wird das Objekt innerhalb der Funktionsausführung verändert, so bleibt diese Änderung auch nach Funktionsausführung bestehen. z.B. `negate(int *arg)`,
    in Pascal/Modula-2: `negate(VAR arg);`
  - **Call-by-Text**: Exotische Programmiersprachen unterstützen auch die Übergabe von Programmcode in Form von Text. Innerhalb der Funktion wird der Text dann interpretiert und ausgeführt.

Beispiel: Call-by-value, Call-by-reference

Beispielprogramm:

```
#include <stdio.h>
void negate_value(int arg){
   arg = -arg;
}
void negate_reference(int *arg){
   *arg = -(*arg);
}
int main(){
   int a = 10;
   negate_value(a);
   printf("Wert nach Aufruf der Call-by-Value Funktion: %d\n",a);
   negate_reference(&a);
   printf("Wert nach Aufruf der Call-by-Reference Funktion: %d\n",a);
   return 0;
}
```

Ausgabe:
Wert nach Aufruf der Call-by-Value Funktion: 10
Wert nach Aufruf der Call-by-Reference Funktion: -10

Verwendung von Zeigern (2): Funktionen mit Zeigern als Parametern

- Die Verwendung von Zeigern als Parameter kann diverse Gründe haben:
  - Die Parameter sind so groß, daß das Kopieren der Parameter ineffizient ist.
  - Häufig soll eine Funktion die Parameter manipulieren, z.B. Sortieren einer Liste. Der Wert des Aufrufsparameter soll also direkt durch das Ergebnis ersetzt werden. Durch die Verwendung von Zeigern kann das Kopieren erspart werden.
  - Besitzt die Funktion mehrere Ergebnisse, so gibt es zwei Möglichkeiten: die Verwendung von Records zur Zusammenfassung der Ergebnisse in einer Variable oder die Verwendung von Zeigern.

    Beispiel: Im AVL-Baum hatten wir eine Funktion `insert()` definiert, die ein Element in einen Unterbaum einsortiert hat und als Ergebnis den neuen Unterbaum, sowie eine eventuelle Höhenänderung zurückgegeben hat. Eine mögliche Signatur der Funktion wäre:

    ```
    int insert(int elem, struct Tree *tree);
    ```

Verwendung von Zeiger (3): Zeiger auf Funktionen

- Wie bereits bei Quicksort dargestellt, kann es sinnvoll sein, auch Funktionen als Argumente zu übergeben. In C kann dies durch Zeiger erreicht werden.
- Vereinbarung: Der Zeiger auf eine Funktion zeigt auf den Beginn des Funktionscodes (= Adresse des ersten Befehls) im Speicher.
- Syntax:

```
void compare(void *a, void *b, int (*func)(void*, void*));
```

  - Die Funktion `compare` soll am Bildschirm ausgeben, ob zwei Elemente *a* und *b* eines beliebigen, aber gleichen Typs denselben Wert aufweisen.
  - Neben *a* und *b* bekommt die Funktion als drittes Argument eine Vergleichsfunktion übergeben.
  - Die Vergleichsfunktion bekommt zwei Argumente vom Typ `void*` (entspricht `'a` in OCaml) übergeben und gibt ein Ergebnis vom Typ Integer zurück (z.B. 0 falls Argumente gleich, -1 falls erstes Argument größer, 1 falls 2. Argument größer).

Beispiel: compare I

Allgemeine Vergleichsfunktion

```
void compare(void *a, void *b, int (*func)(void*, void*))
{
   int ret = func(a,b);
   switch(ret)
   {
        case 0:
                printf("Die Argumente sind gleich\n");
                break;
        case 1:
                printf("Das zweite Element ist größer\n");
                break;
        case -1:
                printf("Das erste Element ist größer\n");
   }
}
```

Beispiel: compare II

Vergleichsfunktion für Integer-Werte:

```
int compareInt(void *a, void *b)
{
   if(*((int*)a)==*((int*)b))
        return 0;
   else if(*((int*)a)>*((int*)b))
        return -1;
   else
        return 1;
}
```

Beispiel: compare III

Test der beiden Funktionen:

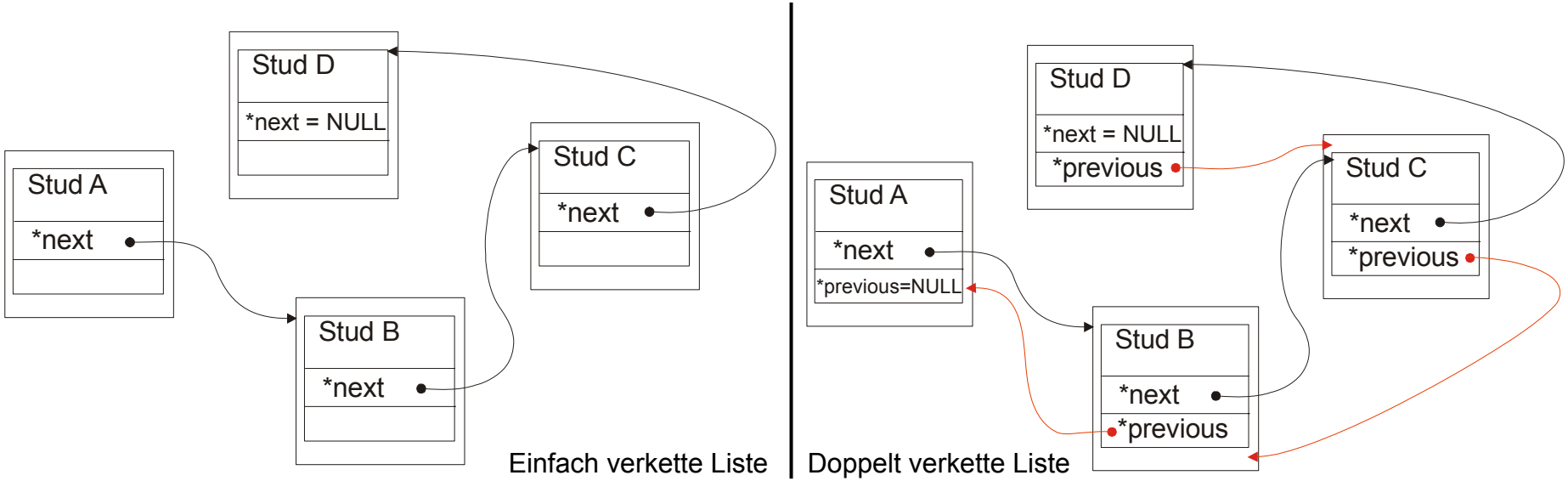
```
int main()
{
   int a=0,b=10;
   compare(&a,&b,compareInt);
   a=20;
   compare(&a,&b,compareInt);
   a=10;
   compare(&a,&b,compareInt);
   return 0;
}
```

Ausgabe des Programms:

```
Das zweite Element ist größer
Das erste Element ist größer
Die Argumente sind gleich
```

Dynamische Datenstruktur Liste bei imperativer Programmierung

- Zeiger erlauben die Verknüpfung von Objekten (Verbünde, Prozeduren, ...), die im Speicher des Rechners an beliebiger Stelle stehen, zu *geordneten* Strukturen (Listen, Bäumen, allg. Graphen)
- Die Ordnung (Aufeinanderfolge von Objekten) wird dabei so vorgenommen, daß sie für spätere Zugriffsoperation effizient ist (Ordnung nach Alphabet für Einwohnerdaten, Nachrichtenblöcke nach Eintreffen für Warteschlangen, Städtedaten nach Topologie, etc.).
- Illustration: Objekte werden als Rechtecke veranschaulicht (mit angedeutetem Objektinhalt und den Namen der Zeiger), Zeiger durch Pfeile.



Dynamische Datenstruktur Liste bei imperativer Programmierung

- Deklaration einer dynamische Datenstruktur in C:

```
struct List
{
  int item;
  struct List *next;
};
```

- Einfach-verkettete Liste.
- Listen werden also in C wie in OCaml rekursiv definiert: eine Liste besteht aus dem aktuellen Element (Kopf) und einem Zeiger auf den Rest der Liste.
- Das Ende der Liste und die leere Liste wird durch den 'NULL'-Pointer angezeigt. Dies muss durch den Programmierer sichergestellt werden.

Funktion zur Bestimmung der Länge einer Liste

```
int length(struct List *list)
{
    int len=0;                          /*parameter for length*/
    struct List *cur = list;            /*current list item */
    while(cur!=NULL)                    /*while we have not reached the end*/
    {
          len++;                        /*we increment the length */
          cur = cur->next;              /*and we go to the next element */
    }
    return len;                         /*we return the length*/
}
```

Anmerkung: Die Funktion kann natürlich nur funktionieren, wenn `list` tatsächlich auf den Anfang der Liste zeigt.

Einfach verkettete Liste: Anfügen von Elementen

```
/* append elem to list and return the new list */


struct List* append(int elem, struct List* list)

{    /* initialise new list element */

     struct List *new_list_item = (struct List*) malloc(sizeof(struct List));
     struct List *cur = list;                                  /* declare variable for parsing list */
     new_list_item->item = elem;                               /* assign elem to list item */
     new_list_item->next = NULL;                               /* there is no successor to the new element */

     if(list==NULL)                                            /* if list empty */
     {
            return new_list_item;                              /* return new element */
     }

     while(cur->next != NULL)                                  /* otherwise parse list and search for last elem */
     {
            cur=cur->next;
     }
     cur->next=new_list_item;                                  /* append new element to the last element */
     return list;                                              /* we return the new list */
}
```

Testprogramm

```
int main()
{
   struct List *my_list = NULL;
   int i=0;

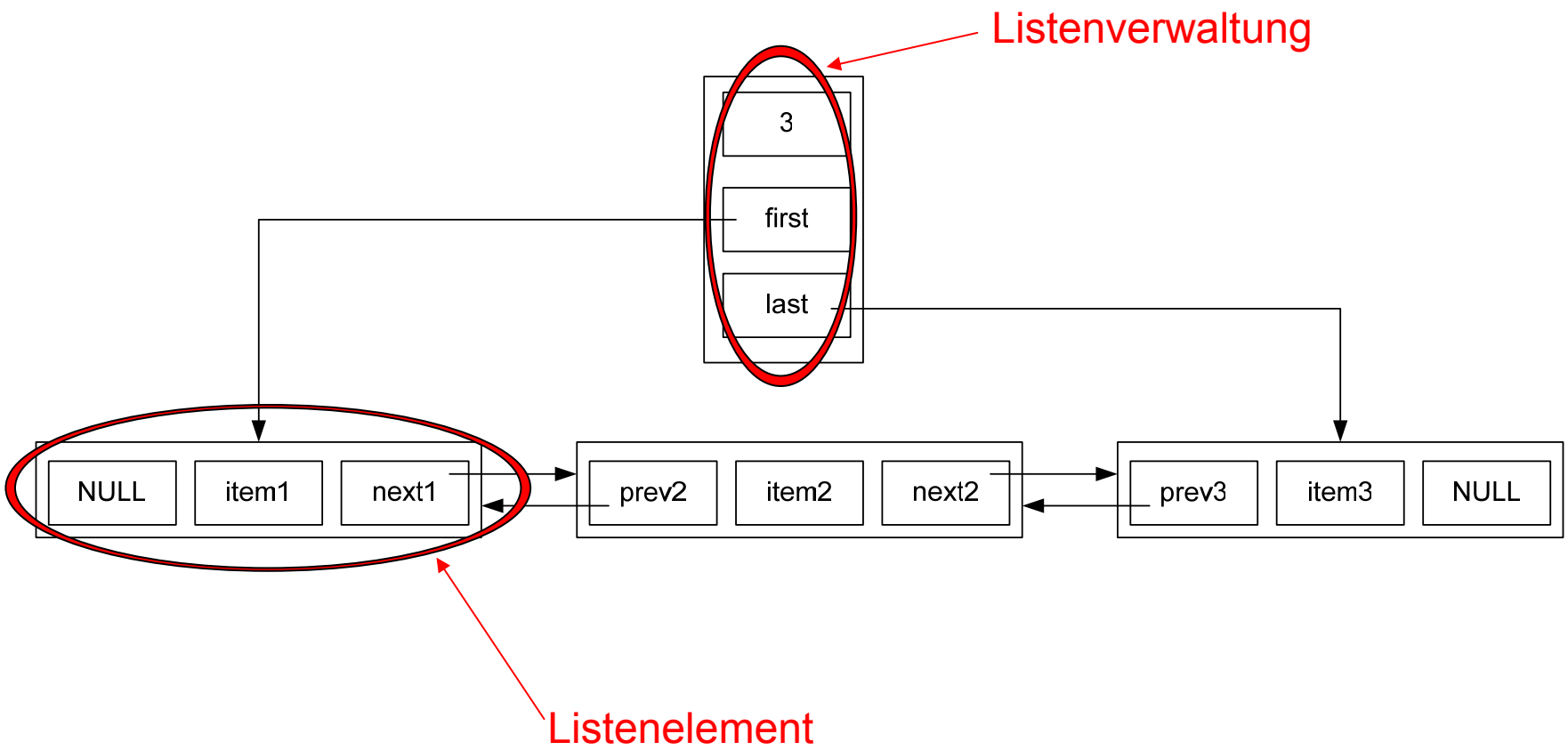
   for(i=0;i<10;i++)
   {
     my_list=append(i,my_list);
   }
   print_list(my_list);
   return 0;
}
```

Anmerkung: Eigentlich müßte man noch eine Funktion zum Löschen der Elemente (und zum Freigeben des Speichers) schreiben. Da die Liste bis zum Ende der Applikation verwendet wird, kann auch darauf verzichtet werden (der angelegte Speicher wird ohnehin zum Ende der Anwendungsausführung freigegeben). Dies spart Schreibarbeit und ist schlechter Stil ....

Listen in C

- Einfach verkettete Listen haben diverse Nachteile:
  - Man kann nicht feststellen, ob ein Element am Anfang einer Liste steht oder ob es in der Mitte steht, d.h. es ist unmöglich den Anfang der Liste zu finden.
  - Das Anhängen von Elementen ist ineffizient, da immer die komplette Liste durchlaufen werden muss.
  - Durch Merken des letzten Elementes kann dieses Problem zwar behoben werden, dies hilft aber nicht beim *Löschen* des letzten Elements, da das vorletzte Element entsprechend modifiziert werden muß.
  - Bessere Lösung: doppelt verkettete Listen, in denen jedes Element sowohl einen Zeiger auf den Vorgänger als auch auf den Nachfolger besitzt:
    - In einem Datentyp List werden die Verwaltungsinformationen gespeichert (Anzahl der Elemente in der Liste, Zeiger auf das erste und letzte Element der Liste)
    - In einem Datentyp ListElement werden die einzelnen Listenelemente gespeichert. Die Listenelemente werden mit einem Zeiger auf den Vorgänger und den Nachfolger ausgestattet.
    - Die Laufzeit beim Anhängen und Löschen des letzten Elementes wird durch doppelt verkettete Listen verringert, dafür steigt der Speicherplatzbedarf.

Doppelt verkettete Listen mit Listenverwaltung



Implementierung doppelt verketteter Listen I

- Typ-Deklaration in C

```
struct ListElement
{
   int elem;
   struct ListElement *prev;
   struct ListElement *next;
};

struct List
{
   int count;
   struct ListElement *first;
   struct ListElement *last;
};
```

- Typ-Deklaration in Pascal

```
TYPE
   PointerToListItem = ^ListItem;

   ListItem = Record
        Elem : Integer;
        Next : PointerToListItem;
        Prev : PointerToListItem
   End;

   List = Record
        Count : Integer;
        First : PointerToListItem;
        Last : PointerToListItem
   End;
```

Implementierung von doppelt verketten Listen II

- Erstellen einer leeren Liste:

```
struct List* createList()
{
   struct List* ret= malloc(sizeof(struct List));
   ret->count=0;
   ret->first=NULL;
   ret->last=NULL;
}
```

- Löschen einer Liste:

```
void deleteList(struct List* l)
{
   while(0!=l->count)
     removeLastItem(l);
   free(l);
}
```

Implementierung von doppelt verketten Listen III

- Anhängen eines Elementes ans Ende der Liste:

```
void append(struct List* l, int item) {
   struct ListElement *newElem=malloc(sizeof(struct ListElement));
   newElem->elem=item;
   newElem->next=NULL;
   if(0==l->count) {
     l->first=newElem;
     l->last=newElem;
     l->count=1;
     newElem->prev=NULL;
   }
   else {
     l->last->next=newElem;
     newElem->prev=l->last;
     l->last=newElem;
     l->count++;
   }
}
```

Implementierung von doppelt verketten Listen IV

- Löschen des letzten Elementes:

```
void removeLastItem(struct List* l){
   struct ListElement *tmp;
   if(1==l->count){
     l->count=0;
     free(l->first);
     l->first=NULL;
     l->last=NULL;
   }
   else{
     tmp=l->last;
     l->last=l->last->prev;
     l->last->next=NULL;
     free(tmp);
     l->count--;
   }
}
```

Implementierung von doppelt verketten Listen V

- Testfunktionen:

```
void printList(struct List* l){
    struct ListElement *tmp=l->first;
    printf("Liste mit %d Elementen: ",l->count);
    while(NULL!=tmp){
      printf("%d;",tmp->elem);
      tmp=tmp->next;
    }
    printf("\n");
}
int main(){
    struct List* myList=createList();
    append(myList,1);
    append(myList,2);
    append(myList,3);
    printList(myList);
    removeLastItem(myList);
    printList(myList);
    deleteList(myList);
}
```

Zusammenfassung

- Die Programmiersprache C zeichnet sich durch:
  - Konstrukte zur imperativen Programmierung (for-, while-Schleifen)
  - maschinennahe Programmierung
  - das Konzept der Zeiger

  aus.
- Weitere Konzepte gegenüber funktionalen Programmiersprachen sind:
  - Variablen, Zustandsbegriff
  - Notwendigkeit zur Verwaltung von Speicher.
- Die uneingeschränkte Freiheit des Programmierers in bezug auf Zeiger und Speicherstrukturen stellt sowohl die Stärke, als auch die größte Schwäche der Programmiersprache dar. Bei der Programmierung muss sehr stark auf die Typkorrektheit der Programme geachtet werden (keine/geringe Überprüfung durch den Compiler).

Überprüfung von C Programmen

- Es stehen Werkzeuge zur (teilweisen) semantischen Überprüfung von C-Programmen bereit, die die meisten typischen Fehler erkennen.
- Das bekannteste Programm ist Lint (z.B. http://lclint.cs.virginia.edu/)
  - Eine Einführung in Lint ist unter http://www.ratiosoft.com/artikel/pedant_org.htm verfügbar.
- Beispiel: Wo ist der Fehler in folgendem Programm:

```
#include <string.h>
#include <stdio.h>

char orgStr[] = "Hello World";

char *getStr( void)
{
   char newStr[20] = "Hello World";
   return newStr;
}
```

```
int doNothing()
{
   return 0;
}

int main()
{
   printf(getStr());
   return doNothing();
}
```

Semantische Überprüfung von C Programmen

- Entscheidende Warnung von Lint: „`Stack allocated storage newStr reachable form return value: newStr`

  `A stack reference is pointed to by an external reference when the function returns. The stack-allocated storage is destroyed after the call, leaving a dangling reference.`“
- In folgender main-Funktion hätte der Fehler eine Auswirkung:

```
int main()
{
  char *str=getStr();
  if( strcmp( orgStr, str) == 0 )
    printf("Die Strings sind
gleich\n");
  else
    printf("Die Strings
     unterscheiden sich\n");
  doNothing();
  if( strcmp( orgStr, str) == 0 )
    printf("Die Strings sind
     gleich\n");
  else
    printf("Die Strings
     unterscheiden sich\n");
  return 0;
}
```

- Anmerkung: Moderne Compiler führen neben der rein syntaktischen Prüfung auch einige Tests in Bezug auf die Semantik durch, allerdings wird eine Fehlerabdeckung wie bei lint nicht erreicht.

Weiteres Beispiel I

- Lint findet im bereits in der Vorlesung vorgestellten Programm folgende potentielle Fehler:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pointer_to_int = NULL;
   int content;

   pointer_to_int=(int*) malloc(sizeof(content));
   *pointer_to_int=20;
   content=*pointer_to_int;
   printf("An der Adresse %d beginnend steht der Wert %d im
               Speicher\n",pointer_to_int,content);
   free(pointer_to_int);
   return 0;
}
```

Die Funktion malloc kann auch NULL zurückliefern, falls nicht genügend Speicher vorhanden ist. Ein Dereferenzieren würde in diesem Fall einen Fehler verursachen.

Es wird eine Variable vom Typ int erwartet. Stattdessen wird eine Variable vom Typ int* verwendet.

Weiteres Beispiel II

- Verbesserte Version:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *pointer_to_int = NULL;
   int content;

   pointer_to_int=(int*) malloc(sizeof(content));
   if(NULL==pointer_to_int){
     printf („Es steht nicht genügend Speicher zur Verfügung\n");
     return -1;
   }
   *pointer_to_int=20;
   content=*pointer_to_int;
   printf("An der Adresse %d beginnend steht der Wert %d im
              Speicher\n",(int)pointer_to_int,content);
   free(pointer_to_int);
   return 0;
}
```

Automaten und Formale Sprachen

Motivation I

- Formale *Sprachen* sind Mengen von Zeichenreihen über einem Alphabet
  - Beispiele:
    - Deutsche Sprache
    - Menge aller Boolschen Ausdrücke
    - Menge aller korrekten OCaml-Programme
    - Menge aller korrekten Anweisungen auf Linux-Kommandozeile (shell-commands)
- *Grammatiken* beschreiben eine Sprache bzw. die korrekten Ausdrücke, die sich in ihr formulieren lassen
  - Wie wird ein Satz im Deutschen aufgebaut?
  - Wie lautet ein korrekter Ausdruck in der Programmiersprache Pascal?
- Ein *Automat* ist ein abstraktes Modell eines (mechanischen, elektrischen, …) Systems zur Verarbeitung von Informationen (Eingaben)
  - ‚Cola'-Automat in der Mensa
  - ‚T9'-Spracherkennung im Mobiltelefon: abhängig von den bisher eingegebenen Zeichen werden ihnen Wörter vorgeschlagen

Motivation II

- Anwendung von formalen Sprachen in der Informatik:
  - Programmiersprachen: Syntaxbeschreibung und automatische Übersetzung
  - Beschreibung/Erkennung gültiger Systemeingaben
- Im Zusammenhang mit Sprachen unterscheidet man *generierende* und *akzeptierende* Systeme.
- Generierende Systeme (z.B. Grammatiken) erzeugen (alle möglichen) Ausdrücke einer Sprache und werden deshalb zur Sprachbeschreibung verwendet.
- Akzeptierende Systeme (z.B. Automaten) prüfen, ob ein vorgelegtes Wort bzw. eine Folge von Wörtern zu einer Sprache gehört. Beispiel: Übersetzer prüfen, ob ein Programm syntaktisch korrekt ist.
- Wichtig: es gibt unterschiedliche Sprach*klassen* verschiedener Mächtigkeit (nach Chomsky, Chomsky-Hierarchie: reguläre, kontextfreie, kontextabhängige, allgemeine Regelsprachen), denen entsprechende Grammatik-Klassen als Erzeugende und Automatenklassen als Akzeptoren zugeordnet werden.

Motivation III

- Sind folgende Programme korrekt? Was berechnen sie?

```
MULTIPLY B BY B GIVING B-SQUARED.
MULTIPLY 4 BY A GIVING FOUR-A.
MULTIPLY FOUR-A BY C GIVING FOUR-A-C.
SUBTRACT FOUR-A-C FROM B-SQUARED
      GIVING RESULT-1.
COMPUTE RESULT-2 = RESULT-1 ** .5.
SUBTRACT B FROM RESULT-2 GIVING
      NUMERATOR.
MULTIPLY 2 BY A GIVING DENOMINATOR.
DIVIDE NUMERATOR BY DENOMINATOR
      GIVING X.
```

```
// JOB // FOR
*LIST SOURCE PROGRAM
*ONE WORD INTEGERS
C----------  C COMPUTE THE CRITIAL VALUES FOR A QUADRAITIC
EQN
C 0=A*X**2+B*X+C
C RETURNS DISCRIMINANT, ROOTS, VERTEX, FOCAL LENGTH,
FOCAL POINT
C X1 AND X2 ARE THE ROOTS
C---------- SUBROUTINE QUADR(A,B,C,DISCR,X1,X2,VX,VY,FL,FPY)
     REAL A,B,C,DISCR,X1,X2,VX,VY,FL,FPY

C DISCRIMINANT, VERTEX, FOCAL LENGTH, FOCAL POINT Y
     DISCR = B**2.0 - 4.0*A*C
     VX = -B / (2.0*A)
     VY = A*VX**2.0 + B*VX + C
     FL = 1.0 / (A * 4.0)
     FPY = VY + FL
     FL = ABS(FL)

C COMPUTE THE ROOTS BASED ON THE DISCRIMINANT
     IF(DISCR) 110,120,130

C -VE DISCRIMINANT, TWO COMPLEX ROOTS, REAL=X1, IMG=+/-X2
110   X1 = -B / (2.0*A)
     X2 = SQRT(-DISCR) / (2.0*A)
     RETURN
     C ZERO DISCRIMINANT, ONE REAL ROOT
120   X1 = -B / (2.0*A)
     X2 = X1
     RETURN
C +VE DISCRIMINANT, TWO REAL ROOTS
130   X1 = (-B + SQRT(DISCR)) / (2.0*A)
     X2 = (-B - SQRT(DISCR)) / (2.0*A)
     RETURN
```

...

Grundbegriffe: Zeichen, Alphabet, Wort

- Ein **Alphabet** ist eine endliche Menge von **Zeichen** (**Symbolen**):
  - Beispiele:
    - Das Alphabet $\sum_1 = \{a,b,c,\ldots,z,A,B,\ldots,Z\}$
    - Das um deutsche Sonderzeichen erweiterte Alphabet $\sum_2 = \sum_1 \cup \{ä,ö,ü,Ä,Ö,Ü,ß\}$.
- Eine endliche Sequenz $a_1a_2\ldots a_n$ mit $a_i \in \sum$ mit $i \in \{1,\ldots,n\}$ heißt **Wort** über dem Alphabet $\sum$.
- Die Menge aller Wörter über $\sum$ wird im folgendem mit $\sum^*$ bezeichnet.
- Die leere Sequenz (das **leere Wort)** bezeichnen wir mit ε.
- Es gilt:

  $\varnothing^* = \{\varepsilon\}$ (Ergebnis der Wiederholung der leeren Sprache ist das leere Wort Analogon $0^0=1$)

  $\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots\}$ (keine oder beliebig viele Wiederholungen)

Grundbegriffe: Konkatenation, Länge eines Wortes

- Für zwei Wörter $v = a_1a_2\ldots a_n \in \Sigma^*$ und $w = b_1b_2\ldots b_n \in \Sigma^*$ definieren wir die **Konkatenation** $v \circ w$ wie folgt:

$$v \circ w = a_1a_2\ldots a_n\, b_1b_2\ldots b_n$$

- Die **Länge** eines Wortes $w = a_1a_2\ldots a_n$ bezeichnen wir als $|w|=n$.
- Die Länge eines Wortes kann auch induktiv definiert werden:

$$|\varepsilon| = 0$$

$$|a| = 1 \text{ für } a \in \Sigma$$

$$|v \circ w| = |v| + |w|$$

Implementierung (OCaml)

- Funktionen zur Konkatenation und Längenbestimmung von Wörtern (wir verwenden Listen zur Speicherung von Wörter)
- Code:

```
let rec concatWord w1 w2 = match w1 with
   | [] -> w2
   | hd::tail -> hd::(concatWord tail w2);;
```

Testaufruf: Konkatenation der Wörter "abc" und "def"

```
concatWord 'a'::'b'::'c'::[] 'd'::'e'::'f'::[];;
```

```
let rec length word = match word with
   | [] -> 0
   | hd::tail -> 1+ length tail;;
```

Testaufruf: Bestimmung der Länge von "abc"

```
length 'a'::'b'::'c'::[];;
```

Implementierung (C) I

- **Anmerkung:** Das komplette Programm inklusive Kommentaren steht auf der Vorlesungshomepage zum Download zur Verfügung.
- Zunächst werden für die Implementierung geeignete Datentypen benötigt.
- Da wir im Folgenden Listen verschiedener Datentypen benötigen, verwenden wir eine allgemeine Implementierung von Listen

```
struct List
{
    void *content;
    struct List *next;
};
```

- `content` zeigt dabei auf einen Speicherplatz, in dem wir ein Objekt eines beliebigen Datentyps speichern können.
- Wörter können also als Listen von Buchstaben implementiert werden.
- `next` zeigt auf das nächste Listenelement.

Implementierung (C) II

- Für den allgemeinen Datentyp Liste können wir nun verschiedene Funktionen implementieren:
  - add() hängt ein Element an eine Liste hinten an. Um nicht in einen Konflikt zu geraten, wird der Wert des Elementes in einen neu angelegten Speicherplatz kopiert. Hierzu ist als Parameter noch die Größe des Speicherplatzes notwendig.

```
struct List* add(void *elem, int size, struct List *list)
{
   struct List *pos=list;
   struct List *new_elem= (struct List*) malloc(sizeof(struct List));
   void *new_content=malloc(size);
   memcpy(new_content,elem,size);
   new_elem->content=new_content;
   new_elem->next=NULL;
   if(NULL==list)
     return new_elem;
   while(NULL!=pos->next)
     pos=pos->next;
   pos->next=new_elem;
   return list;
}
```

Implementierung (C) III

- Für den allgemeinen Datentyp Liste können wir nun verschiedene Funktionen implementieren:
  - copy() kopiert eine Liste. Diese Funktion ist in diversen Fällen nötig, um Kollisionen zu vermeiden.

```
struct List* copy(struct List *list,int size)
{
   struct List *ret=NULL;

   while(NULL!=list)
   {
     ret=add(list->content,size,ret);
     list=list->next;
   }
   return ret;
}
```

Implementierung (C) IV

- Implementierung der Funktion `concat()`:
- Die Funktion liefert eine komplett neue Liste zurück, d.h. alle Elemente beider Listen werden kopiert.

```
struct List* concat(struct List *first, struct List *second, int size)
{
        struct List *ret=NULL;
        struct List *pos=NULL;
        if(NULL==first)
                return copy(second,size);
        else if(NULL==second)
                return copy(first,size);
        ret=copy(first,size);
        pos=ret;
        while(pos->next!=NULL)
                pos=pos->next;
        pos->next=copy(second,size);
        return ret;
}
```

Implementierung (C) V

- Implementierung der Funktion `length()`:

```
int length(struct List *list)
{
  int ret=0;
  while(list!=NULL)
  {
    ret++;
    list=list->next;
  }
  return ret;
}
```

Grundbegriffe: Sprache

- Eine **Sprache** über $\sum$ ist eine Teilmenge $L \subseteq \sum^*$ (d.h. eine Teilmenge der Menge aller möglichen Wörter über das Alphabet $\sum$).
- Die Konkatenation zweier Sprachen $L_1, L_2 \subseteq \sum^*$ ergibt sich aus:

$$L_1 \circ L_2 = L_1 L_2 = \{vw | v \in L_1, w \in L_2\}$$

  d.h. die Konkatenation aller Wörter der ersten Sprache mit allen Wörtern der zweiten Sprache.

- Die **Kleene'sche Hülle *L**** einer Sprache ist definiert durch:

$$L^* = \cup_{i \in \mathbb{N}} L^i, \text{ wo } L^0 = \{\varepsilon\}, L^{i+1} = LL^i \text{ und } L^+ = L^* \setminus \{\varepsilon\}$$

  d.h. die Menge der Zeichenreihen (Wörtern), die durch *n*-fache Konkatenation mit $n=[0,\infty[$ einer Sprache mit sich selbst gebildet werden kann.

Implementierung (OCaml)

- Anmerkungen: zur Speicherung von Sprachen werden Listen von Wörtern verwendet ⇒ somit also Listen von Listen.
- Wir benötigen eine Hilfsfunktion, die ein Wort an sämtliche Wörter einer Sprache anhängt:

```
let rec concat_help w l = match l with
    | [] -> []
    | hd::tail -> (concatWord w hd) :: concat_help w tail;;
```

Testaufruf (Anhängen des Wortes "*abc*" an eine Sprache bestehend aus "*xyz*" und "*uvw*"):

```
concat_help 'a'::'b'::'c'::[]
   [ 'x'::'y'::'z'::[]; 'u'::'v'::'w'::[] ];;
```

- Eigentliche Funktion:

```
let rec concatLanguage l1 l2 = match l1 with
   | [] -> []
   | hd::tail -> (concat_help hd l2)::(concatLanguage tail l2);;
```

Testaufruf (Konkatenation der Sprache ("*abc*", "*def* ") and die Sprache ("*xyz*", "*uvw* ")):

```
concatLanguage [ 'a'::'b'::'c'::[]; 'd'::'e'::'f'::[] ]
   [ 'x'::'y'::'z'::[]; 'u'::'v'::'w'::[] ];;
```

Implementierung (C)

- Als Datentyp für Sprachen verwenden wir die bereits implementierten allgemeinen Listen: Listen von Wörtern.
- Die Funktion zur Implementierung der Konkatenation von Sprachen kann wie folgt implementiert werden:

```
struct List* concat_elements(struct List* first, struct List* second,
   int size_list, int size_elements){
   struct List *ret=NULL;
   struct List *pos=NULL;
   while(NULL!=first){
     pos=second;
     while(NULL!=pos){
          ret=add(concat((struct List*)first->content,
               (struct List*)pos->content,size_elements),size_list,ret);
     pos=pos->next;
     }
     first=first->next;
   }
   return ret;
}
```

Grundbegriff: Grammatik

- Frage: Wie können Sprachen beschrieben werden?
- Antwort 1: Endliche Sprachen können durch Aufzählung aller Wörter beschrieben werden.
  - Beispiel: Duden
- Antwort 2: Bei unendlichen Sprachen müssen die Eigenschaften beschrieben werden:
  - Beispiele (siehe gleich):
    - Sprache, mit allen Wörtern die mit "a" anfangen
    - Sprache der Palindrome (Wörter, die von vorne und hinten gelesen gleich bleiben)
    - alle Wörter, die genau 7-mal den Buchstaben "a" enthalten
- Zur Beschreibung dieser Eigenschaften werden in der Informatik **Grammatiken** verwendet.

Grundbegriffe: Grammatik

- Eine Grammatik beschreibt die Regeln, nach denen die Wörter einer Sprache gebildet werden.
- Eine Grammatik *G* kann durch das Tupel G=(V,$\sum$,P,S) beschrieben werden mit,
  - einer endlichen Menge V von Variablen (Nonterminalen) mit $V \cap \sum = \emptyset$.
  - einem Alphabet $\sum$, wobei die einzelnen Elemente von $\sum$ auch Terminale genannt werden.
  - den Produktionen $P \subseteq (V \cup \sum)^+ \times (V \cup \sum)^*$. Die Produktionen beschreiben also, wie eine nicht-leere Zeichenkette des Alphabets oder der Variablen in eine Zeichenkette des Alphabets oder der Variablen umgewandelt werden kann.
  - einer Startvariablen $S \in V$.
- Die Regeln einer Grammatik / Produktionen (α,β) mit $\alpha \in (V \cup \sum)^+$ und $\beta \in (V \cup \sum)^*$ werden zumeist in der Form $\alpha \rightarrow \beta$ (α geht über in β bzw. die Produktion überführt α nach β) angegeben.
- Im Folgenden werden als Notation für Variablen Großbuchstaben und für Terminale Kleinbuchstaben verwendet.

Grammatik: Beispiel

- Grammatik $G_1=(V,\sum,P,S)$ zur Beschreibung aller Wörter des Alphabets $\sum$={a,b,c}, die mit a anfangen:

  $\sum$={a,b,c}

  V = {A,X}

  S = A

  P = {$A \rightarrow$ aX, $X \rightarrow$ aX, $X \rightarrow$ a, $X \rightarrow$ bX, $X \rightarrow$ b, $X \rightarrow$ cX, $X \rightarrow$ c, $X \rightarrow \varepsilon$}.

- P kann auch kürzer ausgedrückt werden:

  P = {$A \rightarrow$ aX,

  $X \rightarrow$ aX| a | bX | b | cX | c | ε }

Grammatik: Beispiel

- Grammatik $G_2=(V,\sum,P,S)$ zur Beschreibung aller Palindrome des Alphabets $\sum=\{a,b,c\}$:

  $\sum=\{a,b,c\}$

  $V = \{S\}$

  $S = S$

  $P = \{ S \rightarrow aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \varepsilon\}$

Grammatik: Beispiel

- Grammatik $G_3=(V,\sum,P,S)$ zur Beschreibung aller Wörter des Alphabets $\sum$={a,b,c}, die 7 ‘a‘s enthalten:

  $\sum$={a,b,c}

  $V = \{A,X\}$

  $S = A$

  $P = \{A \rightarrow XaXaXaXaXaXaXaX,$

  $X \rightarrow bX \mid b \mid cX \mid c \mid \varepsilon\}$

Definition: Ableitung

Sei $G=(V,\Sigma,P,S)$ eine Grammatik und seien $\alpha,\beta \in (V \cup \Sigma)^*$:

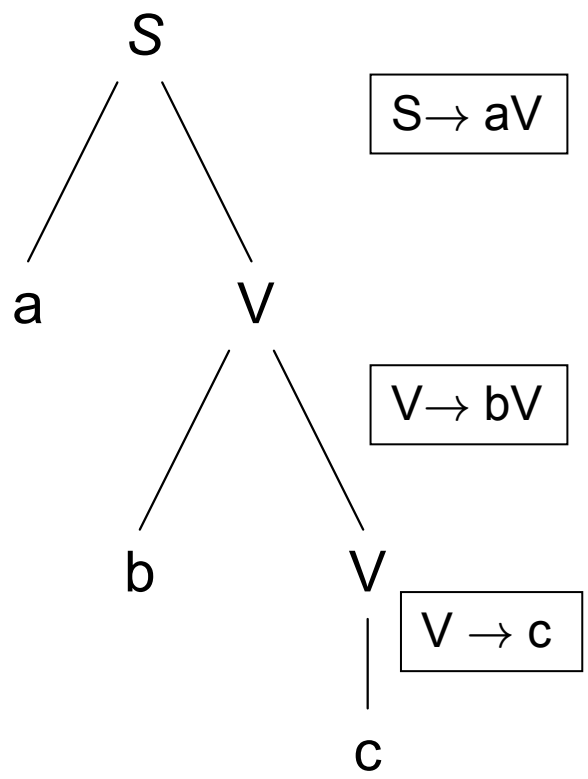
- β heißt aus α direkt ableitbar, wenn eine Produktion die Überführung von α nach β ermöglicht. Formal: β heißt aus α direkt ableitbar, wenn $\gamma,\gamma' \in (V \cup \Sigma)$ und eine Produktion $\alpha' \rightarrow \beta' \in P$ (existieren, so dass gilt: $\alpha=\gamma\alpha'\gamma'$ und $\beta=\gamma\beta'\gamma'$.

  Wir schreiben dann $\alpha \Rightarrow \beta$. (α kann nach β direkt abgeleitet werden)

  β heißt aus α ableitbar, falls $\alpha \Rightarrow^* \beta$ gilt, wobei $\Rightarrow^*$ die transitiv-reflexive Hülle von $\Rightarrow$ ist. Es gibt also eine Menge von Ableitungen mittels derer α nach β überführt werden kann.
- Beispiele:
  - $aSa \Rightarrow aaSaa$ in G_2 (Grammatik der Palindrome). Achtung: nicht "$\rightarrow$ "
  - $S \Rightarrow^* aaaa$ in G_2 ($S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aaaa$)
- Die zu einer Grammatik gehörende Sprache besteht aus allen Wörter *w* über das Alphabet Σ, für die gilt: $S \Rightarrow^* w$.

Definition: Ableitungsbaum

- Die Ableitung eines Wortes aus der Startvariablen kann mittels eines Ableitungsbaums dargestellt werden.
- Beispiel: Ableitung des Wortes "abc" der Grammatik G_1
- Achtung: Der Ableitungsbaum muss nicht immer eindeutig sein (siehe später).

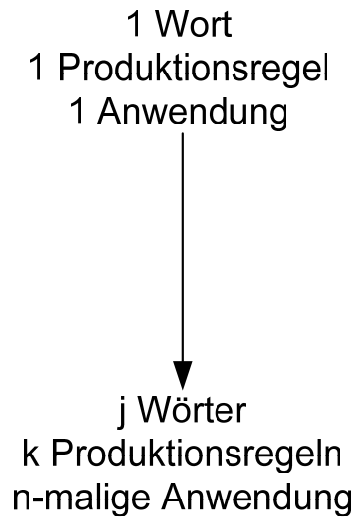


Implementierung (OCaml) I

- Wir wollen nun eine vereinfachte Grammatik in OCaml programmieren.
- **1. Vereinfachung**: Eine Produktion $\alpha \rightarrow \beta$ wird nur auf das erste Vorkommen von α in einem Wort *w* angewendet.
- **2. Vereinfachung:** Es wird nicht zwischen Terminalen und Non-Terminalen Zeichen unterschieden.
- Wir wollen nun eine Funktion implementieren, die für eine gegebene Grammatik alle Wörter herleitet, die in *n* Schritten von der Startvariablen abgeleitet werden können.
- Produktionen, der Form $\alpha \rightarrow \beta$, werden als Listen zweier Wörter α und β codiert.

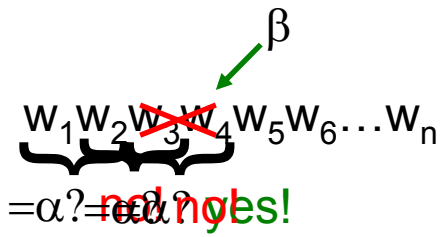
Implementierung (OCaml) II

- Wir benötigen folgende Funktionen:
  - Funktion `executeFirstProduction()` zur Anwendung einer Produktion auf ein Wort. Kann eine Produktion nicht angewendet werden, so soll das Ergebnis der Funktion das unveränderte Wort sein.
  - Funktion `executeFirstProductionOnList()` zur Anwendung einer Produktion auf eine Sprache (also eine Liste von Wörtern).
  - Funktion `expandGrammarStep()` zur Anwendung einer Liste von Produktionen auf eine Sprache.
  - Hauptfunktion `expandGrammar()`: Ermittelung aller Wörter die mittels einer Liste von Produktionen in *n* Schritten von einer Sprache/ Startvariablen abgeleitet werden können.



Implementierung (Ocaml) III

- Vorüberlegung für die Funktionsweise der Anwendung einer Produktion $\alpha \rightarrow \beta$ auf ein Wort *w*.



Schritte:

1. Durchlaufen des Wortes und Suche nach α (`isPrefix()` nötig)
2. Falls α gefunden, Löschen von α (`deletePrefix()` nötig)
3. Einfügen von β

Implementierung (OCaml) IV

- Anwendung einer Produktion auf ein Wort:
  - Hilfsfunktionen: Funktion isPrefix(*pre*,*word*)→ bool, die prüft, ob ein Wort *pre* Präfix des Wortes *word* ist.

```
let rec isPrefix pre word = match (pre,word) with
   | ([],_)-> true
   | (_,[])-> false
   | (prehd::pretail,wordhd::wordtail) when (prehd=wordhd)
      ->(isPrefix pretail wordtail)
   | _ -> false;;
```

  - Hilfsfunktionen: Funktion deletePrefix(*pre*,*word*), die ein Präfix *pre* von einem Wort *word* entfernt.

```
let rec deletePrefix pre word = match (pre,word) with
   | ([],_) -> word
   | (prehd::pretail,wordhd::wordtail) when (prehd=wordhd)
            -> deletePrefix pretail wordtail
   | _ -> failwith "invalid arguments";;
```

Implementierung (OCaml) V

- Anwendung einer Produktion auf ein Wort:
  - Eigentliche Funktion:

```
let rec executeFirstProduction word production = match (word,production) with
   | (word,[alpha;beta]) when (isPrefix alpha word) ->
              beta @ (deletePrefix alpha word)
   | ([],_) -> []
   | (hd::tail,_) -> hd::(executeFirstProduction tail production);;
```

Anwendung der Produktion "b"$\rightarrow$ "d" auf das Wort "abc":

```
executeFirstProduction 'a'::'b'::'c'::[] [ 'b'::[] ; 'd'::[] ];;
```

Implementierung (OCaml) VI

- Wir haben nun eine Funktion zur Anwendung einer Produktion auf ein Wort.
- Nächster Schritt: Implementierung einer Funktion zur Anwendung einer Produktion auf eine Sprache:
  - Funktion:

```
let rec executeFirstProductionOnList list production =
  match list with
  | [] -> []
  | hd::tail ->
   (executeFirstProduction hd production) ::
          (executeFirstProductionOnList tail production);;
```

Implementierung (OCaml) VII

- Bei Anwendung der Funktion können Duplikate entstehen. Wir müssen diese Duplikate also entfernen.
- 1.Schritt: Funktion, die für ein Wort prüft, ob es bereits in einer Liste enthalten ist:

```
let rec inList arg list = match list with
   | [] -> false
   | hd::tail when (arg=hd) -> true
   | hd::tail -> inList arg tail;;
```

- 2. Schritt: Hauptfunktion zur Eliminierung der Duplikate:

```
let rec eliminateDouble list = match list with
   | [] -> []
   | hd::tail when (inList hd tail) -> eliminateDouble tail
   | hd::tail -> hd::eliminateDouble tail;;
```

Implementierung (OCaml) VIII

- Nächster Schritt: Anwendung einer Liste von Produktionen auf eine Sprache:

```
let rec expandGrammarStep start productions = match
  productions with
  | [] -> []
  | hd::tail ->
   eliminateDouble(
     (executeFirstProductionOnList start hd)
     @
     (expandGrammarStep start tail)
   );;
```

Implementierung (OCaml) IX

- Testen der Funktion:

```
let productions =     [ 'S'::[] ; 'a'::'S'::[] ]::
                      [ 'S'::[] ; 'b'::'S'::[] ]::
                      [ 'S'::[] ; 'c'::'S'::[] ]::
                      [ 'S'::[] ; 'a'::[] ]::
                      [ 'S'::[] ; 'b'::[] ]::
                      [ 'S'::[] ; 'c'::[] ]::[];;
```

$S \rightarrow aS$

$S \rightarrow bS$

$S \rightarrow cS$

$S \rightarrow a$

$S \rightarrow b$

$S \rightarrow c$

Testaufrufe:

```
expandGrammarStep [ 'S'::[] ] productions;;
expandGrammarStep [ 'a'::'S'::[] ; 'b'::'S'::[] ] productions;;
expandGrammarStep [ 'a'::[]; 'b'::[] ] productions;;
```

Implementierung (OCaml) X

Ergebnisse der Testaufrufe:

```
expandGrammarStep [ 'S'::[] ] productions;;
```

```
[['a'; 'S']; ['b'; 'S']; ['c'; 'S']; ['a']; ['b']; ['c']]
```

```
expandGrammarStep [ 'a'::'S'::[] ; 'b'::'S'::[] ] productions;;
```

```
[['a'; 'a'; 'S']; ['b'; 'a'; 'S']; ['a'; 'b'; 'S']; ['b'; 'b'; 'S'];
 ['a'; 'c'; 'S']; ['b'; 'c'; 'S']; ['a'; 'a']; ['b'; 'a']; ['a'; 'b'];
 ['b'; 'b']; ['a'; 'c']; ['b'; 'c']]
```

```
expandGrammarStep [ 'a'::[]; 'b'::[] ] productions;;
```

```
[['a']; ['b']]
```

Implementierung (OCaml) XI

- Hauptfunktion: *n*-malige (*n*=count) Anwendung von Produktionen (productionList) auf eine Sprache (start, z.B. das Startsymbol)

```
let rec expandGrammar start productionList count = match count with
    | 0 -> start
    | _ -> expandGrammar (expandGrammarStep start productionList)
         productionList (count-1);;
```

Testaufruf 2-fache Anwendung der Produktionsregeln von Folie 371 auf das Startsymbol:

```
expandGrammar [ 'S'::[] ] productions 2;;
```

Ergebnis:

```
[['a'; 'a'; 'S']; ['b'; 'a'; 'S']; ['c'; 'a'; 'S']; ['a'; 'b'; 'S'];
 ['b'; 'b'; 'S']; ['c'; 'b'; 'S']; ['a'; 'c'; 'S']; ['b'; 'c'; 'S'];
 ['c'; 'c'; 'S']; ['a'; 'a']; ['b'; 'a']; ['c'; 'a']; ['a'; 'b']; ['b'; 'b'];
 ['c'; 'b']; ['a'; 'c']; ['b'; 'c']; ['c'; 'c']; ['a']; ['b']; ['c']]
```

Implementierung (C) I

- Zum Abspeichern einer Produktion können wir einen Record von zwei Wörtern verwenden:

```
struct Production
{
  struct List *pattern;
  struct List *replacement;
};
```

- Die Produktionen einer Grammatik können dann wieder als Listen von Produktionen gespeichert werden.

Implementierung (C) II

- Zunächst werden noch einige Hilfsfunktionen benötigt:
  - Eine Funktion is_prefix() die testet, ob ein Wort das Präfix eines anderen Wortes ist. Neben den beiden Wörtern benötigt die Funktion als letztes Argument eine Funktion zum Vergleich von Listenelementen (Zeiger auf Funktion).

```
int is_prefix(struct List* prefix, struct List *word, int (*cmp) (void*,void*)){
   while(NULL!=prefix){
     if(NULL==word)
               return 0;
     if(0!=cmp(prefix->content,word->content))
               return 0;
     prefix=prefix->next;
     word=word->next;
   }
   return 1;}
```

  - Eine mögliche Vergleichsfunktion wäre:

```
int compare_char(void* c1, void* c2) {
   if(*((char*) c1)==*((char*) c2))
     return 0;
   return 1;}
```

Implementierung (C) III

- Zunächst werden wieder einige Hilfsfunktionen benötigt:
  - Eine Funktion `delete_list` die es ermöglicht eine Liste samt den angeforderten Speicher wieder freizugeben (als Parameter wird ein Zeiger auf eine Funktion zum Löschen der einzelnen Elemente erwartet):

```
void delete_list(struct List* list, void (*delete_function) (void*)){
    struct List* next;
    void* ptr;
    while(NULL!=list){
      next=list->next;
      ptr=list->content;
      free(list);
      delete_function(ptr);
      list=next;
    }
}
```

  - Mögliche Funktionen zum Löschen der einzelnen Elemente wären:

```
void normal_free(void* ptr){
    free(ptr);
}
void list_free(void* list){
    delete_list((struct List*)list,normal_free);
}
```

Implementierung (C) IV

- Zunächst werden wieder einige Hilfsfunktionen benötigt:
  - Eine Funktion `replace` die innerhalb eines Wortes ein Muster durch ein anderes Wort ersetzt:

```
struct List* replace(struct List* list, struct List* pattern, struct List* replacement, int size, int
    (*cmp)(void*,void*)){
    struct List *ret=NULL;
    while(NULL!=list){
      if(1==is_prefix(pattern,list,cmp)){
                  struct List *tmp=concat(ret,replacement,size);
                  if(NULL!=ret)
                              delete_list(ret,normal_free);
                  ret=tmp;
                  while(NULL!=pattern)
                  {
                              pattern=pattern->next;
                              list=list->next;
                  }
                  tmp=concat(ret,list,size);
                  delete_list(ret,normal_free);
                  return tmp;
      }
      ret=add(list->content,size,ret);
      list=list->next;
    }
    return NULL;}
```

Implementierung (C) V

- Analog zu OCaml benötigen wir eine Version, die für ein Wort überprüft, ob es bereits in einer Sprache enthalten ist:

```
static int in_language(struct List* word, struct List* language)
{
    struct List* cur_word=language;
    struct List* cur_char1;
    struct List* cur_char2;
    while(NULL!=cur_word)                {
        cur_char1=word;
        cur_char2=cur_word->content;
        while((cur_char1!=NULL)&&(cur_char2!=NULL))                    {
                    if(*((char*)cur_char1->content)!=*((char*)cur_char2->content))
                                break;
                    cur_char1=cur_char1->next;
                    cur_char2=cur_char2->next;
        }
        if((cur_char1==NULL)&&(cur_char2==NULL))
                    return 1;
        cur_word=cur_word->next;
    }
    return 0;
```

Implementierung (C) VI

Funktion zur Anwendung einer Grammatik:

```
struct List* expand_grammar(struct List* start, struct List* productions, int count){
  int i=0;
  struct List* ret=NULL, *pos_start=NULL, *pos_prod=NULL, *new_word=NULL;
  for(i=0;i<count;i++){
    ret=NULL;
    pos_start=start;
    while(NULL!=pos_start){
      pos_prod=productions;
      while(NULL!=pos_prod){
        new_word=replace((struct List*)pos_start->content,((struct Production*)pos_prod->content)->pattern,
            ((struct Production*)pos_prod->content)->replacement,sizeof(char),compare_char);
       if(NULL!=new_word){
          if(0==in_language(new_word,ret))
             ret=add(new_word,sizeof(struct List),ret);
       }else{
          if(0==in_language(pos_start->content,ret))
             ret=add(copy(pos_start->content,sizeof(char)),sizeof(struct List),ret);
       }
        pos_prod=pos_prod->next;
      }
      pos_start=pos_start->next;
    }
    delete_list(start,list_free);
    start=ret;
  }
  return ret;
}
```

Anmerkungen

- Beim schrittweiser Ausführung der Produktionen entstehen Zwischenzustände von Sprachen. Die einzelnen Wörter können entweder rein aus Terminalen, rein aus Non-Terminalen oder gemischt sein.
- Zu den Wörtern einer Sprache gehören alle Wörter, die nach beliebig vielen Ausführungen der Produktionen rein aus Terminalen bestehen.
- Bei einer endlichen Sprache gibt es einen Anzahl von Produktionsausführungen nach denen sich die Menge der Wörter aus rein-terminalen Zeichen nicht mehr ändert. Dieses Phänomen nennt man **Fixpunkt**. Für jeden Fixpunkt *a* einer Funktion *f* gilt: $f(a)=a$.

Grammatiken zur Beschreibung der Syntax

- Zur Beschreibung der Syntax von Programmiersprachen eignen sich Grammatiken besonders.
- Am häufigsten verwendet: EBNF (Extended Backus Naur Form)
- Die Syntax bei EBNF unterscheidet sich etwas von der bisher kennengelernten Syntax:
  - Anstelle von $\rightarrow$ wird ::= verwendet
  - Non-Terminale werden durch spitze Klammern gekennzeichnet: z.B. <Funktionsname>
  - Terminale werden durch Anführungszeichen gekennzeichnet “a“
- Leider werden oft auch eine abweichende Syntax verwendet, z.B.:
  - Terminale fett gekennzeichnet
  - Anstelle von ::= wird = oder := verwendet
- Referenz: Niklaus Wirth, "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?," Communications of the ACM 20 (11), 1977, pp. 822-823.

Klammern in EBNF

- Klammern erlauben in EBNF eine komfortablere Notation
  - Ausdrücke der Form A $\rightarrow$ aAa | aBa | aCa können in EBNF wie folgt ausgedrückt werden: <A> ::= "a" (<A> | <B> | <C>) "a"

    $\Rightarrow$ runde Klammer in Kombination mit | bedeuteu die exakt einmalige Anwendung eines Elementes der Klammer
  - Ausdrücke der Form A $\rightarrow$ aB | a werden in EBNF zu <A> ::= a[<B>]

    $\Rightarrow$ eckige Klammern bedeuten die maximal einmalige Anwendung
  - Die transitiv-reflexive Hülle der Produktion A $\rightarrow$ aA | ε kann durch <A> ::={ a } notiert werden.

    $\Rightarrow$ geschwungene Klammer bedeuten die n-fache Anwendung auf den Inhalt mit n $\geq$ 0
  - Die transitiv-reflexive Hülle der Produktion A $\rightarrow$ aA | a wird häufig durch <A> ::={a}$^+$ notiert.

EBNF: Syntax von Ocaml (Anfang)

<toplevel-command> ::= “let “<let-binding>{“ and “<let-binding>}“;;“
| “let rec “<let-binding>{“ and “<let-binding>}“;;“
| <expression>“;;“

<expression> ::= <constant>
| “(“<expression>“)“
| <expression><infix-op><expression>
| “if “<expression>“ then “<expression>“ else “<expression>
| “let “<let-binding>{“ and “<let-binding>} “ in “<expression>
| “let rec “<let-binding>{“ and “<let-binding>}“ in “<expression>
| “function “<ident>“ -> “<expression>
| {<expression>}$^{+}$
| <expression>“,“<expression>

<infix-op> ::= “+“ | “-“ | “*“ | “/“ | “+.“ | “-.“ | “*.“ | “/.“ | “&&“ | “||“ | “mod“ | “<“ | “>“ | “<>“
| “<=“ | “>=“

<let-binding> ::= {<ident>}$^{+}$“ = “<expression>

EBNF: Syntax von Pascal (Anfang)

<Program> ::= <ProgramHeading Block> **.**

<ProgramHeading> ::= **program** <Identifier>
(<FileIdentifier> { **,** <FileIdentifier> } **)**
; .

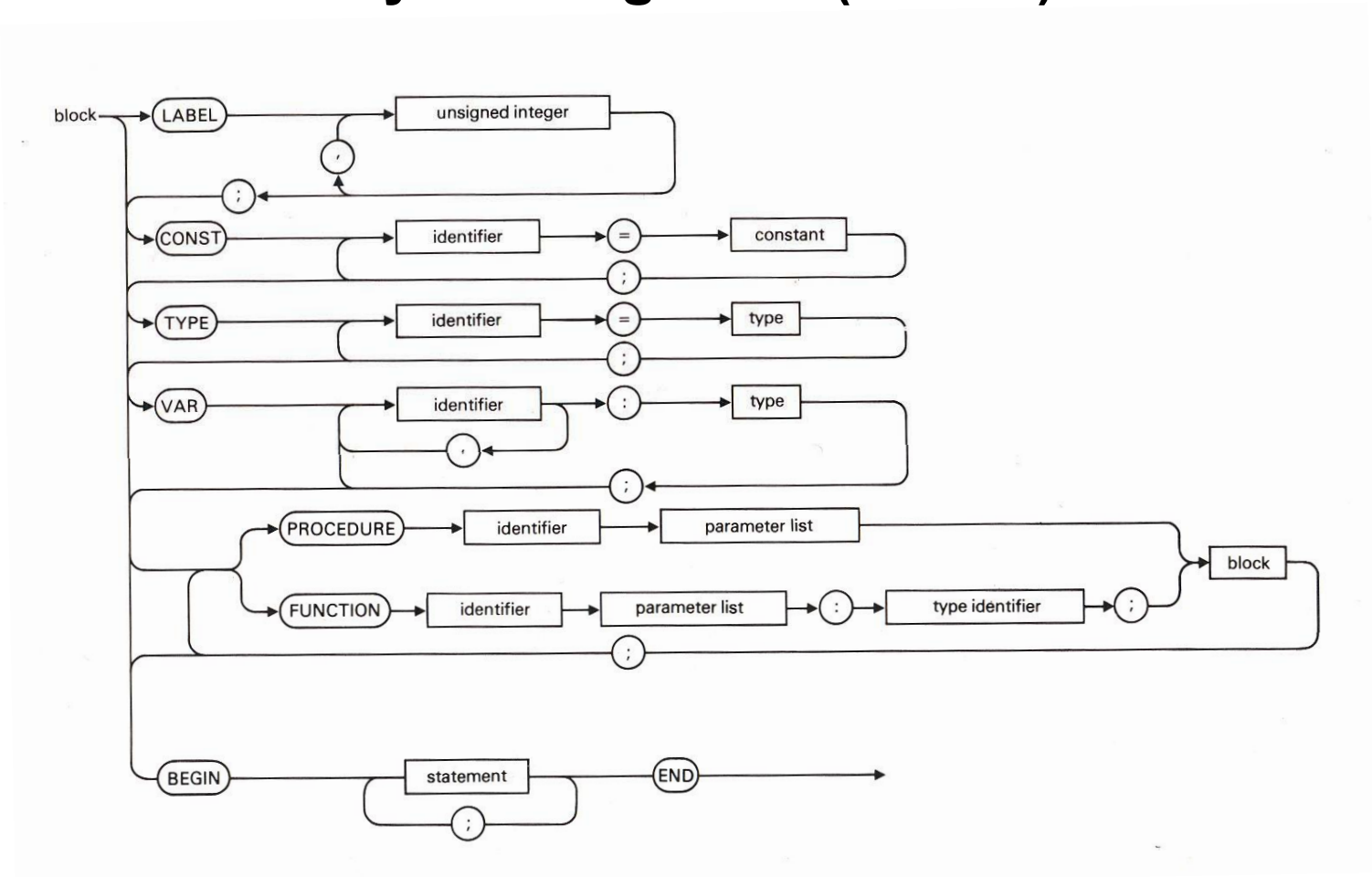
<FileIdentifier> ::= <Identifier>

<Identifier> ::= <Letter> { <LetterOrDigit> }

<Block> ::= [<LabelDeclarationPart>]
[<ConstantDefinitionPart>]
[<TypeDefinitionPart>]
[<VariableDeclarationPart>]
<ProcedureAndFunctionDeclarationPart>
<StatementPart>

<LabelDeclarationPart> ::= **label** <Label> { **,** <Label> } **;**

Syntaxdiagramm (Pascal)



Entscheidungsverfahren

- In Bezug auf Sprachen gibt es diverse Probleme, die interessant für Informatiker sind:
  - Wortproblem: Ist ein Wort in einer Sprache enthalten? Beispiel: Syntaxcheck bei Übersetzern.
  - Endlichkeit / Unendlichkeit: besitzt die Sprache eine endliche oder unendliche Anzahl von Elementen?
  - Äquivalenz: Beschreiben zwei Grammatiken die gleiche Sprache?
  - Inklusion: Ist eine Sprache eine Untermenge einer zweiten Sprache?
  - Disjunktheit: Beinhalten zwei Sprachen mindestens ein gleiches Wort?
- Im Weiteren werden verschiedene Sprachklassen betrachtet und in Bezug auf diese Entscheidungsverfahren und die Effizienz der Entscheidungsverfahren untersucht.
- Dabei werden für jede Klasse die entsprechenden akzeptierenden Automaten und generierenden Sprachen betrachtet.

Chomsky-Hierarchie

- Vorausschau: Die Chomsky-Hierarchie bietet einen Überblick über die verschiedenen Sprachklassen und ihre Eigenschaften:

Typ	Name	Grammatikregeln	Automaten-Modell
Typ 0	Rekursiv aufzählbare Sprachen	beliebig	Turingmaschine TM, es gilt deterministische TM = nicht deterministische TM
Typ 1	Kontextabhängige Sprachen	Kontextsensitiv: $\alpha A \beta \rightarrow \alpha\gamma\beta$ mit $\|\gamma\| \geq 1$, $\alpha,\beta \in (\Sigma \cup V)^*$, $A \in V$	Turingmaschine mit linear beschränktem Band
Typ 2	Kontextfreie Sprachen	Kontextfrei: $A \rightarrow \alpha$ mit $\alpha \in (\Sigma \cup V)^*$, $A \in V$	Kellerautomat
Typ 3	Reguläre Sprachen	Regulär: $A \rightarrow wB$ (rechtslinear) $A \rightarrow Bw$ (linkslinear) Mit $w \in \Sigma^*$, $B \in V \cup \varepsilon$, $A \in V$	Endlicher Automat

- Beginnend bei Typ3-Sprachen werden im Folgenden die einzelnen Automatenkonzepte und Grammatikeigenschaften erläutert.

Reguläre Sprachen

Reguläre Sprachen

- Reguläre Ausdrücke sind eine Untermenge der Menge aller möglichen Wörter über ein Alphabet, die durch sehr einfache Regeln beschrieben werden können:
- Beispiele:

ab*c a(bc)* (a|b)c*

- Der erste Ausdruck beschreibt alle Wörter, die mit „a“ anfangen und mit „c“ enden und dazwischen eine beliebige Anzahl von „b“ enthalten. Der zweite Ausdruck beschreibt alle Wörter die mit „a“ anfangen, gefolgt von einer beliebigen Anzahl von „bc“. Der letzte Ausdruck beschreibt alle Wörter, die entweder mit „a“ oder „b“anfangen, gefolgt von einer beliebigen Anzahl von „c“.
- Die Menge der Symbole (), * und | ist die minimale Menge der Symbole, die zur Beschreibung von beliebigen regulären Ausdrücken nötig sind.
- Alle Wortmengen, die durch die oben genannten Symbole ausgedrückt werden können, gehören zu den regulären Sprachen.
- Häufig werden noch weitere Symbole verwendet, die die Beschreibung einer regulären Sprache erleichtern. Beispiele sind der [ ] Operator (der Ausdruck innerhalb der eckigen Klammern ist optional, oder $^+$ (der Ausdruck kommt beliebig oft, aber mindestens einmal vor), oder ? (steht für ein alphanumerisches Zeichen).

Beispiele zur Anwendung von regulären Sprachen

- Reguläre Ausdrücke werden in zahlreichen Gebieten der Informatik angewandt, u.a. zur :
  - Steuerung des Betriebssystems UNIX durch reguläre Ausdrücke: diese bestehen aus dem Kommandonamen, Optionen und Dateinamen. Ein Beispiel ist das UNIX-Kommando `grep`:

    ```
    grep [options] regexp [files]
    ```

    `grep` erhält als Argument wiederum einen regulären Ausdruck. Das Kommando sucht nun nach Zeichenketten in den angegebenen Dateien `files`, die dem regulären Ausdruck `regexp` entsprechen.
  - Notation von Variablen in der Programmierung. Variablennamen sind typischerweise immer in Form eines regulären Ausdrucks:

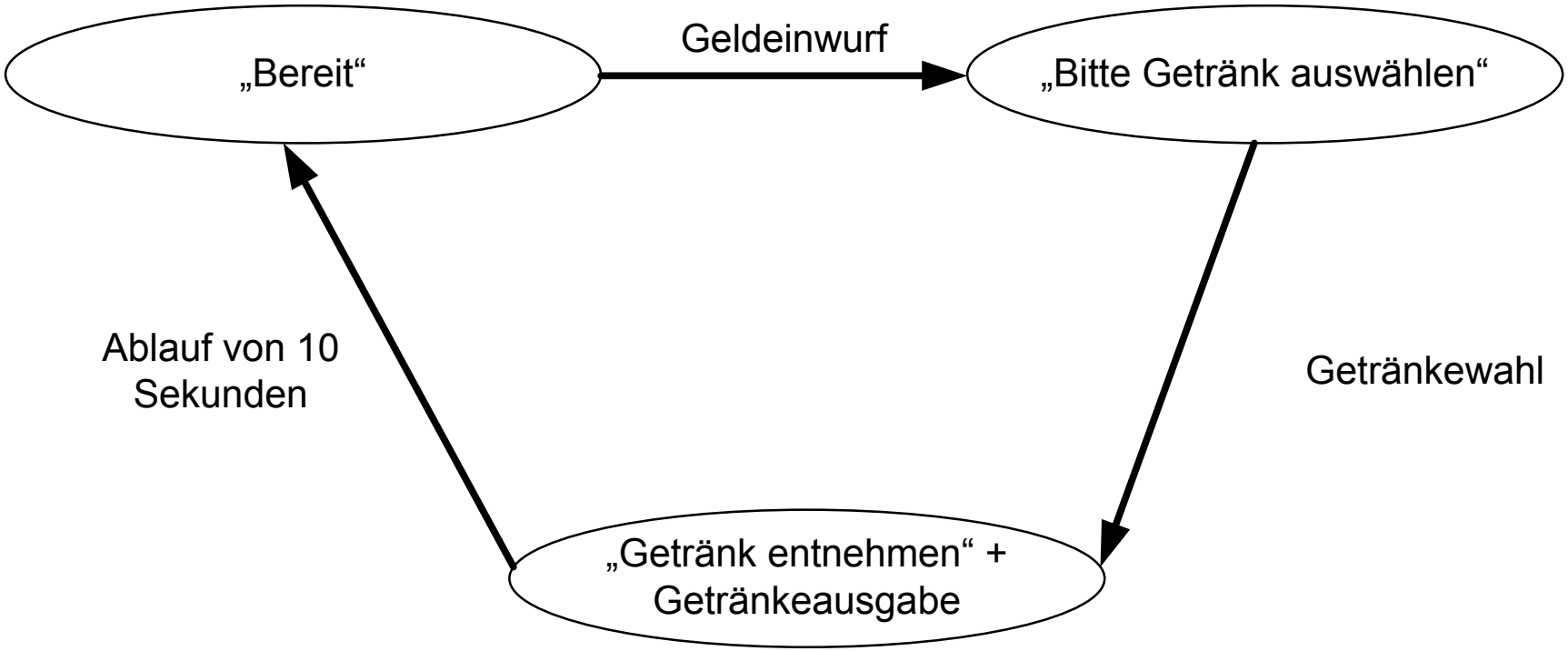
    z.B. `(a|b|…|Z)(a|b|…|Z|0|1|…|9|_)*`

Endliche Automaten

Motivation

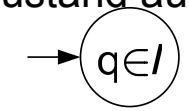
- Im Alltag kommen viele endliche Automaten vor:

  z.B. Colaautomat (stark vereinfacht)

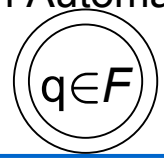


Endliche Automaten

- Vereinfachte Annahme in der Informatik:
  - Ein Automat besteht aus einer endlichen Zahl von Zuständen Q.
  - Der Automat startet in einem Zustand aus der Menge der Startzustände $I \subseteq Q$.



  - Eingaben beschränkt sich auf ein Alphabet Σ.
  - Ein Automat besitzt keine Ausgabe.
  - Für jeden Zustand wird eine Menge von Zustandsübergangen $\delta \subseteq Q \times \Sigma^* \times Q$ definiert. Der Zustandsübergang (p,w,q) bedeutet, daß der Automat beim Lesen eines Wortes w im Zustand p in den Zustand q übergeht.)
  - Die Definition der Zustandsübergänge erlaubt auch Übergänge der Form (p,ε,q), d.h. der Automat kann ohne Eingabe vom Zustand p in den Zustand q wechseln.
  - Eine Eingabe w ist dann gültig, wenn sie den Automaten in einen Zustand aus der Menge der Endzustände $F \subseteq Q$ bringt.



EA: Zustandsübergangsrelation

- Ähnlich der Definition bei Sprachen definieren wir die transitiv-reflexive Hülle δ^* (also das Ergebnis des Ausführens mehrerer Zustandsübergänge in Folge) der Zustandsrelation δ als die kleinste Relation mit:
  - $\delta \subseteq \delta^*$
  - $\forall\, q \in Q: (q,\varepsilon,q) \in \delta^*$
  - $\forall\, p,q,r \in Q,\ u,v \in \Sigma^*: (p,u,q) \in \delta^* \wedge (q,v,r) \in \delta^* \rightarrow (p,uv,r) \in \delta^*$.
- Die akzeptierte Sprache $L(A)$ eines endlichen Automaten $A=(Q,\Sigma,\delta,I,F)$ ist damit definiert durch:

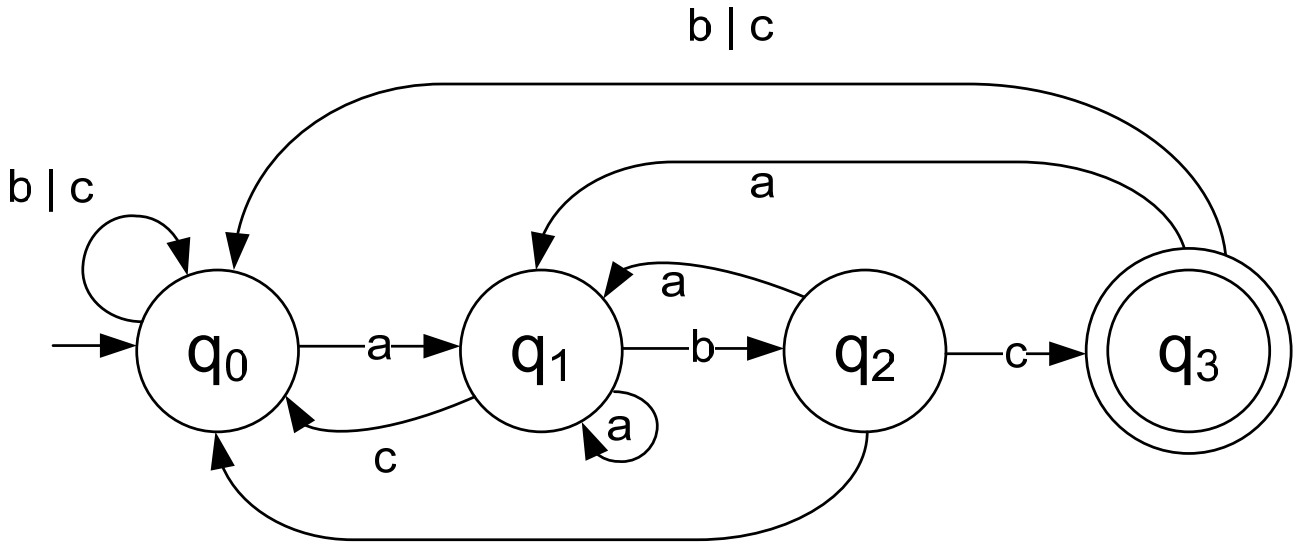
  $$L(A)=\{w \in \Sigma^* \mid \exists p \in I, q \in F: (p,w,q) \in \delta^*\}$$

  Ein endlicher Automat akzeptiert also ein Wort w, wenn sich der Automat nach Einlesen des Wortes in einem Endzustand befindet.
- Ein schönes Applet zur Funktionsweise von endlichen Automaten findet sich unter:

  http://www.cs.montana.edu/webworks/projects/fsa-old/fsa.html

Beispiel: Endlicher Automat

- Automat, der alle Wörter über dem Alphabet $\sum$ = {a,b,c} akzeptiert, die mit "abc" enden.



- Endzustände werden durch einen doppelten Kreis gekennzeichnet.

Definition: Deterministischer, vollständiger Automat

- Der Automat auf der vorangegangenen Folie ist deterministisch und vollständig:
  - Ein Automat ist **buchstabierend**, falls gilt $\delta \subseteq Q \times \Sigma \times Q$, also ein Zustandsübergang schon beim Einlesen eines *einzelnen* Zeichens erfolgt
  - Ein Automat ist **deterministisch (DEA)**, wenn er genau einen Startzustand besitzt und für jede Eingabe in jedem Zustand genau ein oder kein Nachfolgezustand existiert, d.h. es gilt:

$$\forall p,q,r \in Q \; \forall a \in \Sigma: (p,a,q) \in \delta \wedge (p,a,r) \in \delta \Rightarrow (q=r)$$

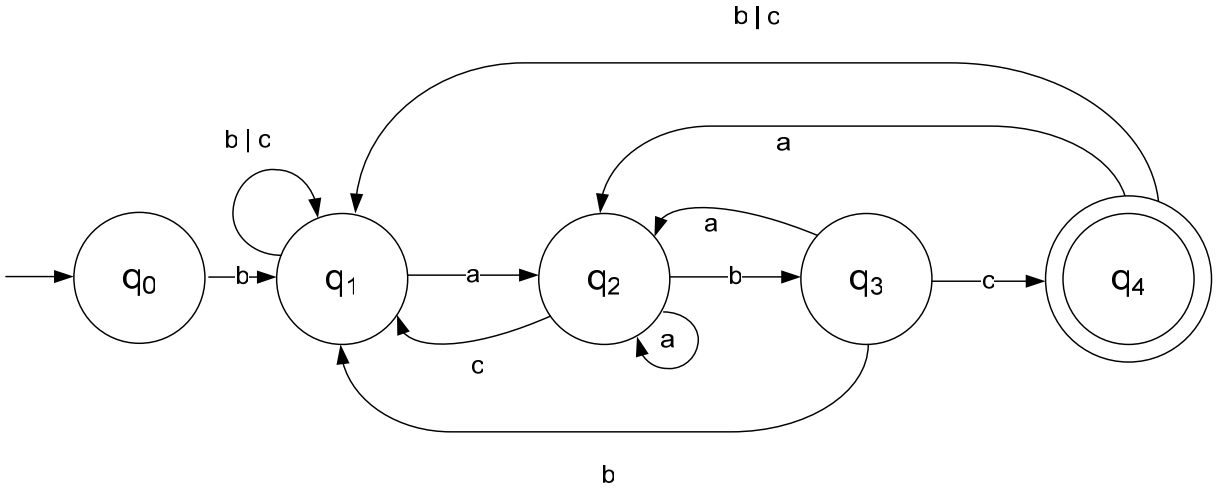
  - Ein Automat ist **vollständig**, falls gilt:

$$\forall p \in Q \; \forall a \in \Sigma: \exists q \in Q \text{ mit } (p,a,q) \in \delta$$

- Deterministische, vollständige Automaten erleichtern die Prüfung, ob ein Wort vom Automaten akzeptiert wird. Andererseits wird die Darstellung schnell unübersichtlich.

Beispiel

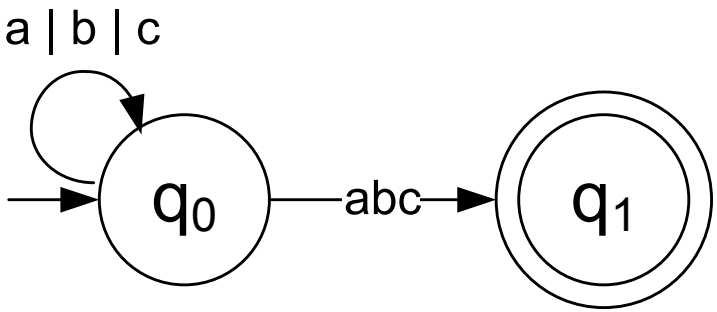
- Beispiel für einen buchstabierenden, deterministischen, aber nicht vollständigen Automaten:



- Der Automat akzeptiert alle Wörter, die mit "b" anfangen und mit "abc" enden.
- Für den Zustand q_0 sind nur die Übergänge für das Zeichen 'b' eingezeichnet. Beginnt das Wort mit einem anderen Zeichen, so akzeptiert der Automat das Wort nicht.

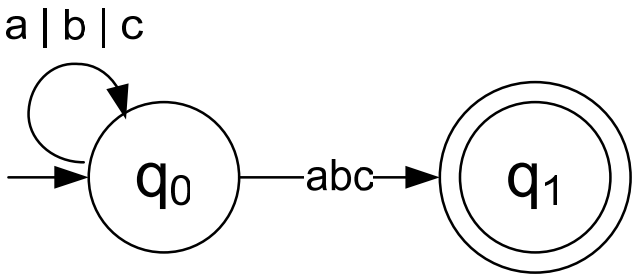
Unvollständiger, nicht-deterministischer Automat

- Unvollständiger, nicht-deterministischer Automat, der ebenfalls alle Wörter über dem Alphabet $\sum$ = {a,b,c} akzeptiert, die mit "abc" enden.



Bemerkungen zu nicht-deterministischen Automaten

- Bei deterministischen Automaten befindet sich der Automat nach einer Eingabe immer in genau einem Zustand.
- Ein nicht-deterministischer Automat kann sich nach einer Eingabe stattdessen in mehreren Zuständen gleichzeitig befinden. Dabei ist er jedoch optimistisch: ist einer der Zustände ein Endzustand, so wird die Eingabe akzeptiert.
- Beispiel: nach Eingabe von "abc" befindet sich der Automat im Zustand q_0 und q_1, er akzeptiert die Eingabe. Folgt nun die Eingabe von einem weiteren a, so erkennt der Automat, dass er sich momentan nur noch im Zustand q_0 befindet.
- Problem: Das Merken der Zustände kann sehr aufwendig werden.

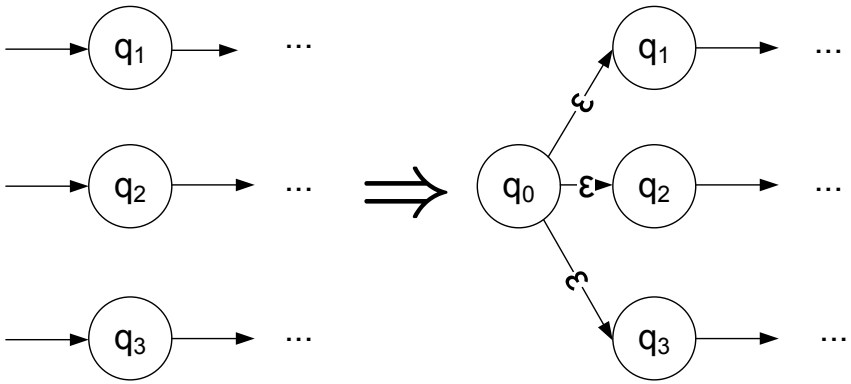


Äquivalenz von Automaten

- Zwei Automaten heißen **äquivalent**, wenn sie die gleiche Sprache akzeptieren.
- **Satz**: Zu jedem endlichen Automaten gibt es einen äquivalenten deterministischen, vollständigen, endlichen Automaten.
- Beweis durch Konstruktion mittels der folgenden Schritte:
  1. Reduzierung der Startzustände auf einen Startzustand
  2. Aufspaltung aller Zustandsübergänge $(p,w,q)\in\delta$ mit $|w|>1$.
  3. Eliminierung aller Zustandsübergänge $(p,\varepsilon,q)\in\delta$.
  4. Erstellung eines deterministischen Automaten durch Potenzautomatenkonstruktion.

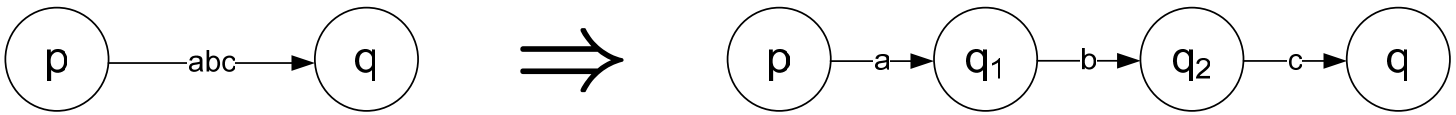
1. Schritt: Reduzierung der Startzustände

- Idee:
  - Einführung eines neuen Startzustandes q_0
  - Hinzufügen von neuen Zustandsübergangen (q_0,ε,q) für alle $q \in I$.
  - Wir wählen q_0 als einzigen Startzustand.
  - $A=(Q,\Sigma,\delta,I,F) \Rightarrow A'=(Q \cup q_0,\Sigma,\delta+\{(q_0,\varepsilon,p)|p \in I, q_0,F\}$
  - Dieser Schritt ändert nichts an der akzeptierten Sprache $\Rightarrow L(A)=L(A')$ und der Automat hat nun nur noch einen Startzustand



2. Schritt: Aufspaltung von Zustandsübergängen

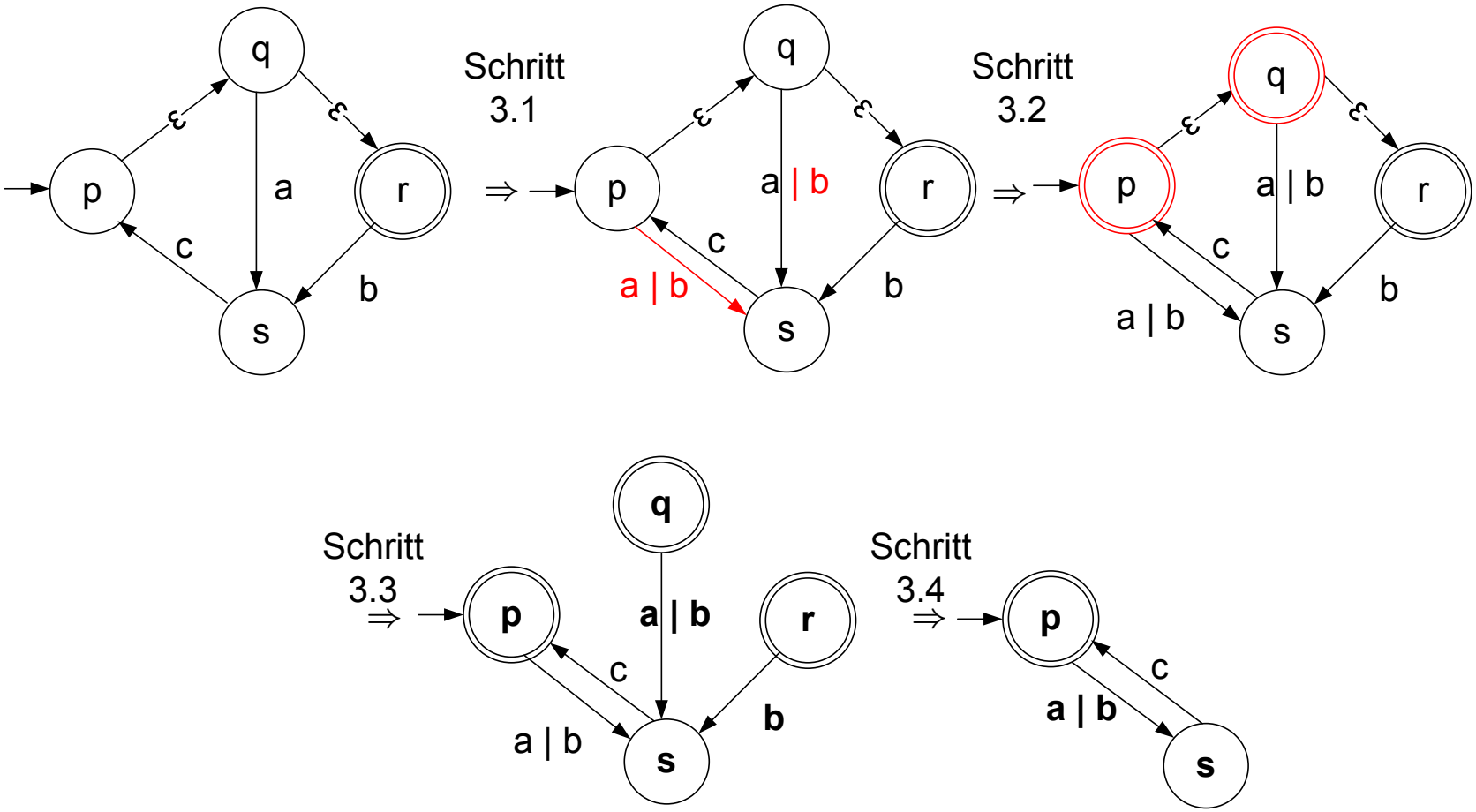
- Für jeden Übergang $(p,w,q)\in\delta$ mit $w = a_1a_2\ldots a_n$ und $n>1$ werden n-1 neue Zwischenzustände $p_1,\ldots,p_{n-1}$ eingeführt.
- Der Übergang $(p,w,q)\in\delta$ mit $w = a_1a_2\ldots a_n$ wird durch die Übergänge (p,a_1,p_1), (p_1,a_2,p_2), $\ldots,(p_{n-1},a_n,q)$ ersetzt.
- Dieser Schritt ändert nichts an der akzeptierten Sprache und es existieren nur noch Zustandsübergänge der Form (p,w,q) mit $w = \sum \cup \varepsilon$



3. Schritt: Eliminierung von Zustandsübergängen

- Ziel: Ersetzen aller ε-Übergänge
- 1. Schritt: Für alle Zustandsübergänge $(p,\varepsilon,q) \in \delta$ und $(q,a,r) \in \delta$ mit $a \in \sum$ fügen wir einen neuen Zustandsübergang (p,a,r) ein.
- 2. Schritt: Für alle Zustandsübergänge (p,ε,q) mit $q \in F$ wird auch p ein Endzustand.
- **Anmerkung:** Die Schritte 1 und 2 müssen mehrfach durchlaufen werden, damit auch ε-Ketten, also Folgen von Zustandsübergängen ohne das Einlesen von Zeichen berücksichtigt werden.
- 3. Schritt: Löschen aller Zustandsübergänge $(p,\varepsilon,q) \in \delta$.
- 4. Schritt: Löschen aller unnötigen Zustände q mit $(\forall (p,a,q) \in \delta \rightarrow (p \neq q)) \wedge q \notin F$, also alle Zustände, die nicht vom Startzustand erreichbar sind. Dieser Schritt ist eigentlich nicht nötig, vereinfacht aber den Automaten.

3. Schritt: Eliminierung von Zustandsübergängen

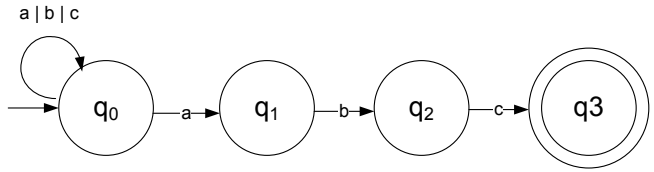


4. Schritt: Potenzautomatenkonstruktion

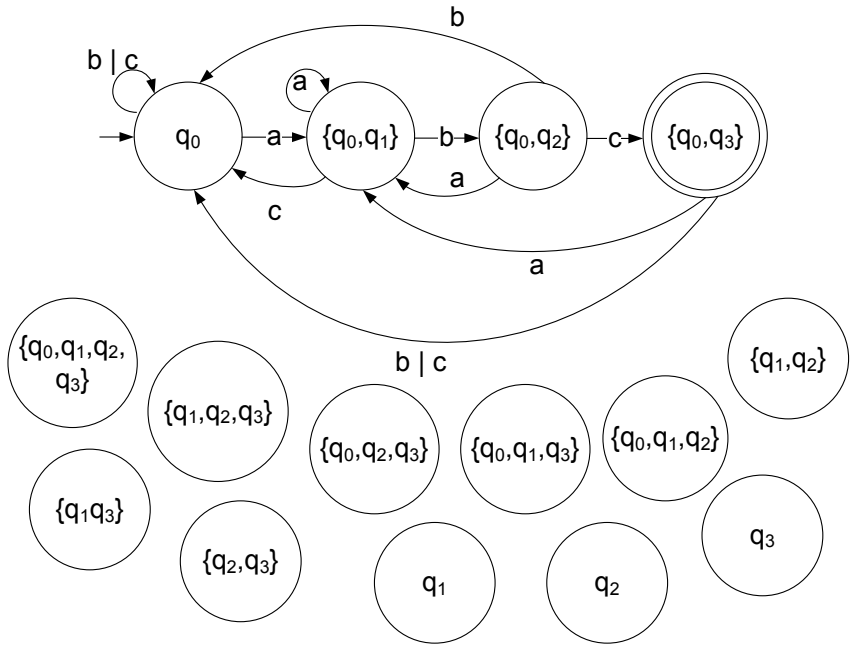
- Im letzten Schritt müssen wir nun aus einem nicht-deterministischen buchstabierenden endlichen Automaten $A=(Q,\Sigma,\delta,q_0,F)$ mit einem Startzustand q_0, einen deterministischen endlichen Automaten konstruieren.
- **Grundidee**: Erzeugung eines neuen Automaten mit $A=(2^Q,\Sigma,\delta,q_0,F)$, wobei 2^Q die Potenzmenge der ursprünglichen Zustände, d.h. die Menge aller Teilmengen von Q, ist.
- Existieren nun zwei Übergänge $(p,a,q)\in\delta$ und $(p,a,q')\in\delta$ mit $q\neq q'$, so werden diese Übergänge durch einen neuen Übergang $(p,a,\{q,q'\})$ ersetzt, wobei sicher gilt: $\{q,q'\}\in 2^Q$. Dieser Schritt wird solange durchgeführt, bis es keine Übergänge $(p,a,q)\in\delta$ und $(p,a,q')\in\delta$ mit $q \neq q'$ gibt.
- Der Automat ist nun deterministisch und, nachdem die Potenzmenge endlich ist, ist der neue Automat auch endlich.
- **Problem**: viele der neu erzeugten Zustände sind nicht nötig.

Beispiel Potenzautomatenkonstruktion

nicht deterministischer, buchstabierender Automat:



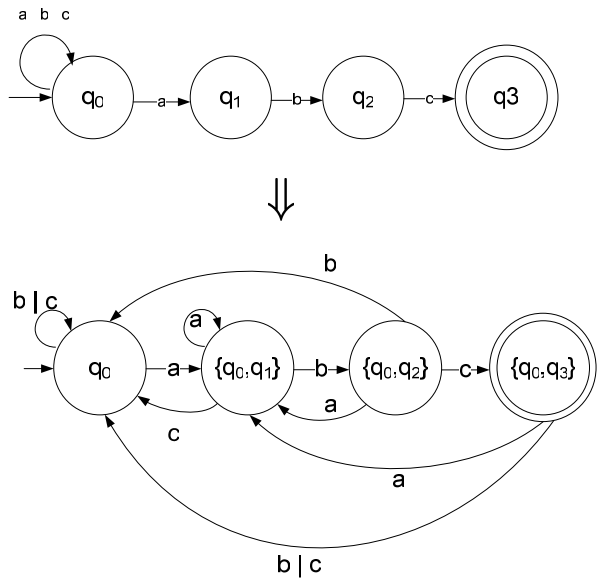
deterministischer, buchstabierender Automat: (viele unnötige Zustände)



Potenzautomatenkonstruktion: Myhill-Verfahren

- Effektivere Lösung: Konstruiere ausgehend vom Startzustand unter Test aller möglichen Eingaben einen deterministischen endlichen Automat. Existieren für eine Eingabe zwei mögliche Zustände, konstruiere einen neuen Zustand, der beide Zustände umfaßt. Zustandsmengen, die einen Endzustand enthalten, werden zu Endzuständen.

Zustände \ Eingabe	a	b	c
q_0	$\{q_0,q_1\}$	q_0	q_0
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$	q_0
$\{q_0,q_2\}$	$\{q_0,q_1\}$	q_0	$\{q_0,q_3\}$
$\{q_0,q_3\}$	$\{q_0,q_1\}$	q_0	q_0



Umwandlung in Programmcode

- Deterministische, buchstabierende Automaten können sehr einfach in ausführbare Programme umgewandelt werden.
- Vorgehensweise:
  - Nummerierung der Zustände
  - Einführung von zwei Variablen: Position im Wort (initialisiert mit 0), aktueller Zustand (initialisiert mit der Startzustandsnummer)
  - Eine Schleife zum Durchlaufen des Wortes
  - Innerhalb der Schleife: Switch-Anweisung für die einzelnen Zustände
  - Innerhalb jedes case-Statement: if-Anweisung und Abfragen der einzelnen möglichen Zeichen, Zuweisung des neuen Zustandes, falls kein Zustandsübergang vorgesehen wird ein Fehlerzustand angenommen.
  - Nach Schleife: Falls der aktuelle Zustand einem Endzustand entspricht, akzeptiert der Automat das Wort, andernfalls nicht.

Beispiel: Programm, das alle Wörter akzeptiert, die mit "abc" enden

```
int main (int argc, char** argv)
{
	int state=0;
	char* word;
	int i=0;

	if(argc!=2)
	{
		printf("Das Programm braucht genau ein
				Wort als Argument\n");
		return -1;
	}
	word=argv[1];
	while(word[i]!='\0')
	{
		switch(state)
		{
```

```
		case 0:
			if(word[i]=='a')
				state=1;
			else if ((word[i]=='b')||(word[i]=='c'))
				state=0;
			else
				state=4; //error state
			break;
		case 1:
			if(word[i]=='b')
				state=2;
			else if (word[i]=='a')
				state=1;
			else if (word[i]=='c')
				state=0;
			else
				state=4; //error state
			break;
```

Beispiel: Programm, das alle Wörter akzeptiert, die mit "abc" enden - Fortsetzung

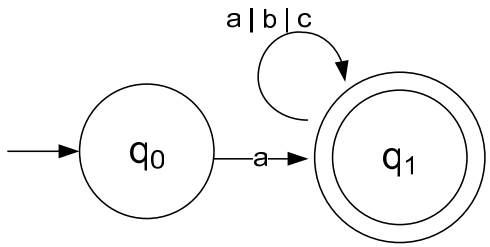
```
case 2:
        if(word[i]=='c')
                state=3;
        else if (word[i]=='a')
                state=1;
        else if (word[i]=='b')
                state=0;
        else
                state=4; //error state
        break;
case 3:
        if(word[i]=='a')
                state=1;
        else if (word[i]=='b')
                state=0;
        else if (word[i]=='c')
                state=0;
        else
                state=4; //error state
        break;
                default:
                        break;
                }
                i++;
        }
        if(3==state)
                printf("Der Automat akzeptiert das Wort\n");
        else
                printf("Der Automat akzeptiert das Wort
                        nicht\n");
        return 0;
}
```

Von EA erkannte Sprachen I

- Fragestellung: Welche Sprachen können mittels eines endlichen Automaten überprüft werden?
- Erinnerung: Wir hatten bei den Grammatiken drei Sprachen betrachtet:
  - $L(G_1)$: Alle Wörter, die mit a anfangen.
  - $L(G_2)$: Alle Palindrome.
  - $L(G_3)$: Alle Wörter, die 7mal den Buchstaben a enthalten.

Von EA erkannte Sprachen II

- $L(G_1)$: Alle Wörter, die mit a anfangen:
  - Der Automat kann leicht gebaut werden.

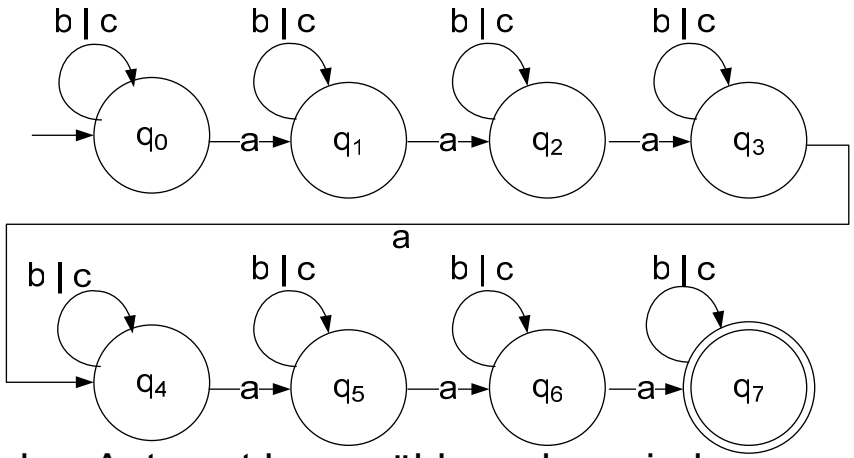


- $L(G_2)$: Alle Palindrome:
  - Um einen geeigneten Automaten zu konstruieren, müßten wir uns die Buchstaben bis zur Mitte des Wortes merken und ab der Mitte überprüfen, ob die Buchstaben wieder vorkommen.
  - Problem 1: Wie erkennen wir die Mitte? Gar nicht, aber wir können einen nicht-deterministischen Automaten bauen, der die Mitte errät.
  - Problem 2: Wie merken wir uns die Buchstaben. Einzige Möglichkeit durch Verwendung von Zuständen. Aber: es sind nur endlich viele Zustände verfügbar $\Rightarrow$ es gibt immer unendlich viele Palindrome, die der endliche Automat nicht erkennen kann.

  $\Rightarrow$ Die Sprache kann *nicht* durch einen endlichen Automaten dargestellt werden.

Von EA erkannte Sprachen III

- $L(G_3)$: Alle Wörter, die 7mal den Buchstaben „a“ enthalten.
  - kann durch folgenden Automaten dargestellt werden:



⇒ Ein endlicher Automat kann zählen, aber wieder nur endlich. Die Sprache, die alle Wörter mit gleicher Anzahl a und b enthält, kann nicht durch einen endlichen Automat dargestellt werden.

⇒ Die Sprache $a^n b^n$ kann also auch nicht dargestellt werden (wichtig in Bezug auf Programmiersprachen – siehe später)

Reguläre Sprachen

Reguläre Sprachen

- **Fragestellung:** Welche Sprachen können durch endliche Automaten dargestellt werden.
- Wir nennen diese Sprachen **reguläre Sprachen**.
- Wir wollen die regulären Sprachen durch reguläre Ausdrücke beschreiben, die wir induktiv definieren können:

- Syntax:
  1. $\emptyset$ ist ein regulärer Ausdruck.
  2. ε ist ein regulärer Ausdruck.
  3. Gilt $a \in \sum$ so ist a ein regulärer Ausdruck.
  4. Für zwei reguläre Ausdrücke *x* und *y* sind
     - $(x \mid y)$
     - (xy)
     - (x^*)

     reguläre Ausdrücke.

- Bedeutung:
  1. $\emptyset$ bezeichnet die leere Sprache $\emptyset$.
  2. ε bezeichnet die Sprache $\{\varepsilon\}$, also die Sprache bestehend aus dem leeren Wort.
  3. Gilt $a \in \sum$ bezeichnet $\{a\}$ die Sprache
  4. Bezeichnet *x* die Sprache *X* und *y* die Sprache *Y* so bezeichnen
     - $(x \mid y)$ die Sprache $X \cup Y$
     - (xy) die Sprache $X \circ Y$
     - (x^*) die Sprache X^*

Satz von Kleene

- Satz: Eine Sprache ist genau dann regulär, wenn sie von einem endlichen Automaten akzeptiert wird.
- Beweis: $\Rightarrow$ Für jeden Teilschritt der induktiven Definition kann ein Automat konstruiert werden.

Automat akzeptiert nichts

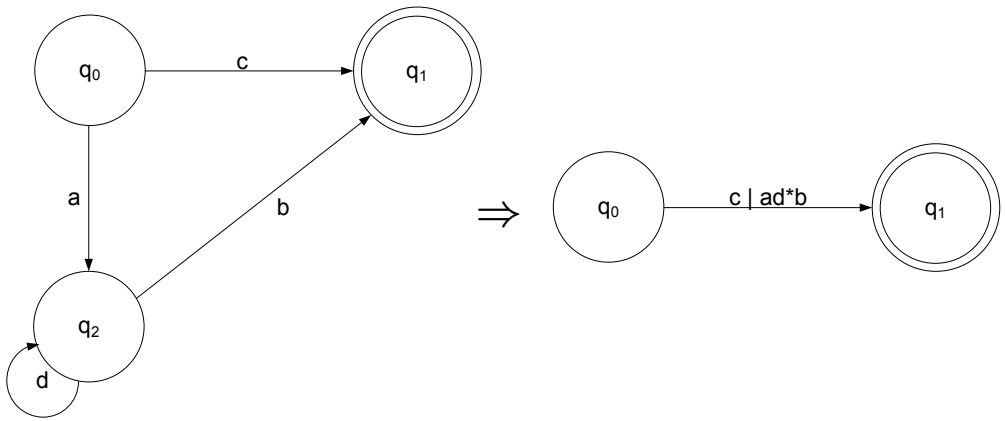
Automat akzeptiert das leere Wort

Automat akzeptiert das Wort "a"

  - (x | y) kann durch Parallelschaltung der Automaten realisiert werden.
  - xy kann durch Serienschaltung der Automaten realisiert werden.
  - x* kann durch Verbindung der Endzustände mit den Startzuständen mittels ε Übergangs und der Umwandlung aller Startzustände in Endzustände realisiert werden.

Satz von Kleene

- Satz: Eine Sprache ist genau dann regulär, wenn sie von einem endlichen Automaten akzeptiert wird.
- Beweis: $\Leftarrow$ Überführung des endlichen Automaten in einen Automaten mit einem Startzustand und einem Endzustand. Die Kante enthält genau den entsprechenden regulären Ausdruck.
- Idee zur Realisierung: Schrittweise Elimination von Zuständen unter der Berücksichtigung aller Pfade über den entsprechenden Zustand.
- Beispiel:



Grammatiken zu regulären Sprachen

- Mit dem bisherigen Wissen kann man leicht Regeln für Grammatiken zur Repräsentation von regulären Sprachen aufstellen.
- Endliche, deterministische, buchstabierende Automaten $A=(Q,\Sigma,\delta,q_0,F)$ können eins-zu-eins in Grammatiken $G=(V,\Sigma,P,S)$ umgesetzt werden.
  - Zustände $q_x \in Q$, werden zu Nonterminalen $v_x \in V$.
  - Das zum Startzustand q_0 gehörende Nonterminal wird zur Startvariablen S.
  - Die Übergänge $(p_x,a,p_y) \in P$ werden zu:
    - $p_x \rightarrow a v_y$, falls $p_y \notin F$
    - $p_x \rightarrow a$, sonst.

Rechtslineare Grammatik

- **Definition**: Eine Grammatik $G=(V,\Sigma,P,S)$ heißt rechtslinear, wenn jede Produktion die Form

  $$A \rightarrow wB \text{ oder } A \rightarrow w$$

  für $A,B \in V$ und $w \in \Sigma^*$ hat.
- Es ist klar, dass auch jede rechtslineare Grammatik direkt in einen Automaten umgesetzt werden kann $\Rightarrow$ rechtslineare Grammatiken beschreiben reguläre Sprachen.

- **Anmerkung**: Analog können linkslineare Grammatiken definiert werden. Da reguläre Sprachen unter der Spiegelung abgeschlossen sind (Invertierung der Start- und Endzustände, sowie der Kanten beim endlichen Automaten), beschreiben links- und rechtslineare Grammatiken die gleiche Klasse.

Eigenschaften regulärer Sprachen

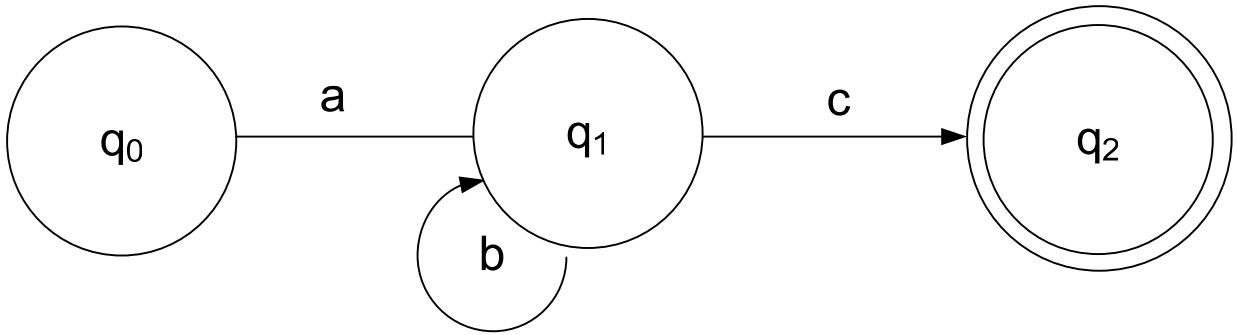
Eigenschaft regulärer Sprachen

Problem: Wie kann ich erkennen, daß eine Sprache (nicht) regulär ist?

1. Überlegung: Jede endliche Sprache kann durch einen endlichen Automaten dargestellt werden und ist somit regulär.
2. Überlegung: Da ein endlicher Automat nur begrenzt viele Zustände hat, muss jedes Wort, das gleich viele oder mehr Zeichen als der entsprechende buchstabierende Automat Zustände hat, einen Zyklus im Automaten durchlaufen haben.

   $\Rightarrow$ Für jedes Wort, das gleich viele oder mehr Zeichen hat, als der entsprechende buchstabierende Automat Zustände besitzt, muß man einen solchen Zyklus finden können. Durch zusätzliches oder weniger häufiges Durchlaufen der Zyklen können Wörter erzeugt werden, die in der Sprache enthalten sein müssen. ($\Rightarrow$ Pumping-Lemma, Aufpumpen des Zyklus)

Anschauliches Beispiel für Pumping-Lemma



- Der Automat akzeptiert das Wort abbbc. Wir können nun das Wort in den Teil vor dem Zyklus (a), den Zyklus (bbb) bzw. in diesem Fall die dreifache Anwendung im Zyklus und den Teil nach dem Zyklus (c) aufteilen.
- Die Wörter ac (bei Reduktion der Anzahl der Durchläufe) und die Wörter abbbbbbc (bei Erhöhung der Anzahl der Durchläufe) müssen auch in der Sprache enthalten sein.
- Allgemein gilt: a(bbb)*c muss in der Sprache enthalten sein.

Pumping Lemma (uvw-Theorem)

- Formale Formulierung des Pumping Lemmas: Für jede reguläre Sprache L gibt es ein $n \in \mathbb{N}$, so daß für jedes $x \in L$ mit $|x| \geq n$ eine Zerlegung $x=uvw$ mit $|v| \geq 1$ und $|uv| \leq n$ existiert, so dass $uv^*w \subset L$ gilt.
- Erläuterung:
  - n ist die Anzahl der Zustände des buchstabierenden Automaten.
  - Jedes Wort mit n oder mehr Buchstaben muß mindestens einen Zyklus durchlaufen haben $\Rightarrow$ Man findet deshalb eine entsprechende Zerlegung.
  - $|v| \geq 1$ gilt, da der Zyklus mindestens aus einem Buchstaben besteht.
  - $|uv| \leq n$ gilt, da der erste Zyklus spätestens n Buchstaben abgeschlossen sein muss.

Anwendung des Pumping-Lemma

- Wir hatten bereits gesehen, dass die Sprache aller Palindrome L_P keine reguläre Sprache ist.
- Dies kann durch das Pumping-Lemma bewiesen werden.
- Beweis durch Widerspruch:
  - Annahme: Jedes Wort w mit mehr als *n* Buchstaben kann entsprechend dem Pumping-Lemma in *uvw* zerlegt werden, mit $|v| \geq 1$ und $|uv| \leq n$, so dass gilt $uv^*w \in L_P$.
  - Wir betrachten das Wort $w = a^n b a^n$ mit $|w| = 2n+1$
  - Für jede mögliche Zerlegung *uvw* von *w* mit $|v| \geq 1$ und $|uv| \leq n$ gilt: $v = a^+$.
  - Es muss entsprechend der Annahme gelten, dass $uv^*w \in L_P$. Gegenbeispiel $uv^2w = a^m b a^n$ mit $m>n$ und somit $uv^2w \notin L_P$. $\Rightarrow L_P$ ist keine reguläre Sprache.

Abschlusseigenschaften

- Wir haben gesehen, dass nicht jede Sprache durch endliche Automaten dargestellt werden kann.
- Interessant ist nun, ob die Sprachklasse der endlichen Automaten bezüglich der Operatoren ($\cup, \cap, \neg, \circ, ^*$) abgeschlossen ist.
- Bezüglich einer Menge *M* abgeschlossen ist eine Funktion *f* genau dann, wenn für alle Argumente arg$\in M$ das Ergebnis ebenfalls in der Menge *M* liegt.
- Die Abschlusseigenschaften für von endlichen Automaten akzeptierten Sprachen können einfach durch Konstruktion von Automaten bewiesen werden.

Abschlusseigenschaften

Falls L_1 und L_2 zwei Sprachen sind, die von endlichen Automaten akzeptiert werden, so gibt es auch einen endlichen Automaten für:

- $L_1 \cup L_2$: Parallelschaltung der Automaten, Wort wird akzeptiert, wenn *einer* der beiden Automaten im Endzustand ist.
- $L_1 \cap L_2$: Parallelschaltung der Automaten, Wort wird akzeptiert, wenn *beide* Automaten im Endzustand sind.
- $\neg L_1$: Konstruktion eines deterministischen und vollständigen endlichen Automaten, Invertierung der Zustände (Endzustand $\Leftrightarrow$ kein Endzustand)
- $L_1 \circ L_2$: Serienschaltung der Automaten, von jedem Endzustand des ersten Automaten wird ein ε –Übergang zu den Startzuständen des zweiten Automaten eingefügt.
- L_1^*: Ähnlich $L_1 \circ L_2$ jedoch darf ε nicht vergessen werden.

akzeptiert.

Entscheidungsverfahren

- Entscheidungsverfahren in Bezug auf reguläre Sprachen L_R:
  - Wortproblem: effiziente Lösung des Wortproblems $w \in L_R$ in $O(|w|)$. Verfahren: Testen des Wortes mit Hilfe des endlichen Automatens.
  - Endlichkeit / Unendlichkeit: einfache und effiziente Lösung: Konstruktion des endlichen Automatens und suche nach einem Zyklus. Ist ein Zyklus vorhanden, so ist die Sprache unendlich, ansonsten ist sie endlich.
  - Disjunktheit $L_{R1} \cap L_{R2} = \emptyset$: Der Beweis der Disjunktheit kann durch Konstruktion des entsprechenden Automaten und Test auf Leerheit (Pfad von Startzustand zu einem Endzustand) durchgeführt werden.
  - Inklusion $L_{R1} \subseteq L_{R2}$: Die Inklusion kann auf die Betrachtung der Leerheit von $L_{R1} \cap \neg L_{R2}$ zurückgeführt werden. Der entsprechende Automat kann konstruiert werden und auf Leerheit getestet werden.
  - Äquivalenz: Die Äquivalenz $L_{R1} = L_{R2}$ kann durch Beweis der gegenseitigen Inklusion $L_{R1} \subseteq L_{R2}$ und $L_{R2} \subseteq L_{R1}$ bewiesen werden.

Kontextfreie Sprachen

Motivation

- Wir haben bisher die regulären Sprachen kennengelernt.
- Größter Vorteil der regulären Sprachen ist die effiziente Lösung des Wortproblems.
- Nachteil in Bezug auf Informatik: verschachtelte Strukturen beliebiger Tiefe lassen sich nicht als reguläre Sprache formulieren. Beweisidee: eine der einfachsten verschachtelten Strukturen sind ineinander verschachtelte geschweifte Klammern {{{…}}}, was der Sprache $\{^n\}^n = a^n b^n$ entspricht. Auf Seite 413 wurde bereits gezeigt, dass diese Sprache keine reguläre Sprache ist.
- Verschachtelte Strukturen sind jedoch essentiell in verschiedenen Programmiersprachen (verschachtelte Schleifen, If-Abfragen,…)
- Kontextfreie Grammatiken erlauben verschachtelte Strukturen und sind daher das Mittel zur Beschreibung der Syntax von Programmiersprachen.
- Die bereits kennengelernte Backus-Naur-Form ist eine Variante der kontextfreien Grammatiken.

Kontextfreie Grammatik

- **Definition**: Eine Grammatik $G=(V,\sum,P,S)$ heißt **kontextfrei**, wenn gilt $P \subset V$ x $(V \cup \sum)^*$. Eine Sprache L heißt kontextfrei, wenn es eine kontextfreie Grammatik G gibt, die L erzeugt (d.h. $L(G)=L$).
- Kontextfreie Grammatiken beziehen ihren Namen aufgrund der Tatsache, dass ein Nonterminales Zeichen unabhängig vom Kontext ersetzt werden kann.
- Beispiel für kontextfreie Grammatik:

  $V=\{R,O\}$,

  $\sum=\{a,b,c,+, *, ()\}$,

  $S= R$,

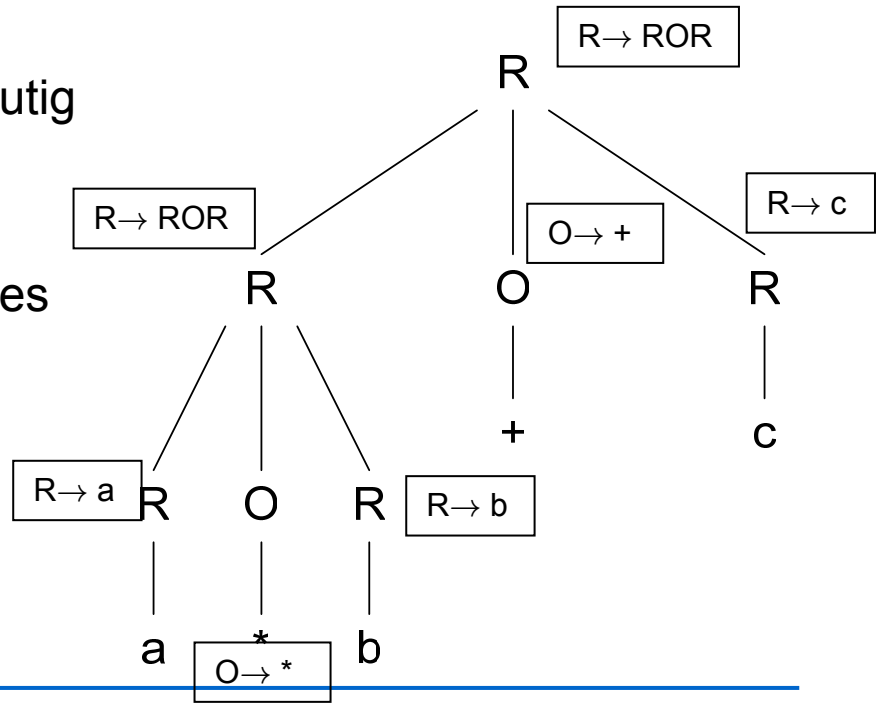
  $P=\{ \quad R \rightarrow a \mid b \mid c \mid (R) \mid R\ O\ R$,

  $O \rightarrow + \mid * \}$
- Achtung: "(", ") ", "*", "+" sind hier Terminale und keine Metazeichen.

Ableitungsbaum

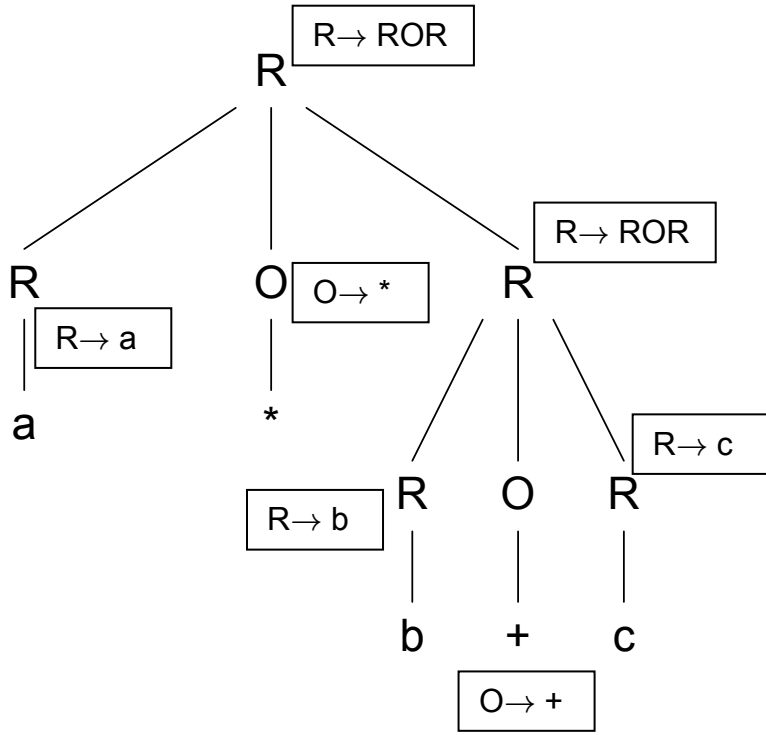
- Zur Darstellung der Herleitung eines Wortes eignet sich der Ableitungsbaum (auch Syntaxbaum).
- Das Wort a*b+c kann durch folgenden Baum hergeleitet werden:

- Frage: ist dieser Ableitungsbaum eindeutig oder gibt es noch weitere Ableitungsbäume?
- Welche Konsequenzen hätte es, wenn es weitere Ableitungsbäume gebe?



Mehrdeutigkeit

- Zum Beispiel auf vorangeganger Folie gibt es einen zweiten Ableitungsbaum:
- Ableitungsbäume stellen die Auswertungsreihenfolge dar, deshalb sind Mehrdeutigkeiten nicht erwünscht.
- **Definition**: Eine kontextfreie Grammatik heißt **mehrdeutig**, wenn es mindestens ein Wort der erzeugten Sprache mit zwei verschiedenen Ableitungsbäumen gibt; andernfalls heißt die Grammatik **eindeutig**. Eine Sprache heißt inhärent mehrdeutig, wenn jede Grammatik, die diese Sprache erzeugt, mehrdeutig ist.



Eindeutige Grammatik

- Für das Beispiel kann auch eine eindeutige Grammatik gefunden werden:

  V={R,O,L},

  $\sum$={a,b,c,+, *},

  S= R,

  P={ R $\rightarrow$ a | b | c | (R) | L,

  L $\rightarrow$ a O | b O | c O,

  O $\rightarrow$ + R | * R }

- Mit Hilfe dieser Grammatik werden die Ausdrücke immer von links nach rechts unter Berücksichtigung der Klammern ausgedrückt.
- Diese Grammatik hat eine andere Semantik als typische arithmetische Ausdrücke (Stichwort: Punkt-vor-Strich zusätzlich zur Auswertung von links-nach-rechts). Die Ausdrücke werden hier nur stur von links-nach-rechts ausgewertet.
- Aufgabe: Wie sieht eine Grammatik für beliebig geklammerte arithmetische Terme aus? (Lösung später)

Eindeutigkeit / Mehrdeutigkeit

- Wie bereits gezeigt, gibt es für viele kontextfreie Sprachen eindeutige Grammatiken.
- Eine eindeutige Grammatik ist nötig, vor allem in Bezug auf Parser:

  Beispiel: Wie kann folgender Code interpretiert werden:

  if x **then if** y **then** a **else** b

  (Zu welchem **if** gehört das **else**)
- **Problem**: es gibt keine Möglichkeit zu erkennen, ob eine Sprache eine eindeutige Grammatik hat.
- **Relativierung**: Für die meisten praktisch relevanten kontextfreien Sprachen gibt es eine eindeutige Grammatik, die jedoch häufig deutlich komplizierter ist.
- **Problemlösung**:
  - Sprachspezifisch: Belegen der Operatoren mit Prioritäten bezüglich der Auswertungsreihenfolge, z.B. Punkt-vor-Strich bei arithmetischen Ausdrücken.

Normalformen - Motivation

- Kontextfreie Grammatiken $G=(V,\Sigma,P,S)$ können beliebig kompliziert sein:
  - Sie können **Kettenproduktionen** enthalten z.B. $P=\{S\rightarrow A, A\rightarrow B, B\rightarrow C, C\rightarrow a\}$.
  - Sie können nicht **produktive** Variablen/Nonterminale enthalten. Eine Variable A ist produktiv, wenn gilt: $\exists\, w\in\Sigma^*$ mit $A\Rightarrow^* w$. Aus einer nicht produktiven Variablen kann also kein Wort aus ausschließlich terminalen Zeichen abgeleitet werden.
  - Sie können nicht **erreichbare** Variablen enthalten. Eine Variable A ist erreichbar, falls gilt: $\exists\, \gamma,\gamma' \in (V\cup\Sigma)^*$ mit $S\Rightarrow^* \gamma A\gamma$. Für eine nicht erreichbare Variable gibt es also keinen Ableitungspfad von der Startvariablen zu der Variablen.
  - Sie können nicht **nützliche** Variablen enthalten. Eine Variable ist nützlich, falls gilt $\exists\, w\in\Sigma, \exists\, \gamma,\gamma' \in (V\cup\Sigma)^*$ mit $S\Rightarrow^* \gamma A\gamma'\Rightarrow^* w$. Für eine nicht nützliche Variable kann also kein Wort mittels einem Ableitungsbaum der die Variable enthält abgeleitet werden.

    Beispiel für eine produktive, erreichbare aber nicht nützliche Variable:

    $$P=\{S\rightarrow AB,\ A\rightarrow a\}$$

    (Da parallel zu A auch immer das nicht produktive Variable B erzeugt wird, kann kein Wort produziert werden.)

Normalformen - Motivation

- Um festzustellen, ob ein Wort *w* in der von der Grammatik G beschriebenen Sprache L_G enthalten ist, ist es nützlich zu wissen, wann die Suche abgebrochen werden kann.
- **Voraussetzung**: Grammatiken sollten **monoton** sein. Eine Grammatik ist monoton, wenn für jede Produktion $p \in P$ mit $\alpha \rightarrow \beta$ gilt: $|\alpha| \leq |\beta|$. Da in kontextfreien Grammatiken für jede Produktion gilt $\alpha \in V$, muss nur noch sichergestellt werden, dass keine Produktion der Form $\alpha \rightarrow \varepsilon$ in der Grammatik enthalten ist. Um trotzdem Grammatiken zu erlauben, die das leere Wort ε enthalten, wird für die Startvariable eine Ausnahme gemacht, d.h. $S \rightarrow \varepsilon$ ist erlaubt, solange S nie auf der rechten Seite einer Produktion vorkommt.
- **Vorteil**: Um zu beweisen, dass ein Wort *w* nicht von der monotonen Grammatik akzeptiert wird, reicht es aus, alle Ableitungen $S \rightarrow^* x$ mit $x \in (V \cup \Sigma)^*$ und $|x| \leq w$ zu testen.

Normalformen

- Noch besser wäre es für die Lösung des Wortproblems, wenn die genaue Anzahl der Ableitungsschritte im Voraus bestimmbar wäre.
- **Chomsky-Normalform CNF**: Eine Grammatik ist in Chomsky-Normalform, wenn alle Produktionen die Form $A \rightarrow BC$ oder $A \rightarrow a$ mit $A,B,C \in V$ und $a \in \sum$ haben und nur nützliche Variablen enthält. Um auch das leere Wort zu akzeptieren, ist die Produktion $S \rightarrow \varepsilon$ erlaubt, solange S nie auf der rechten Seite vorkommt.
- Man kann zeigen, dass jedes Wort w mit $|w|=n$ in $2n$-1 Ableitungsschritten mittels einer Grammatik in CNF erzeugt werden kann.
- **Satz**: Jede nicht-leere kontextfreie Sprache kann durch eine Grammatik in CNF erzeugt werden.
- **Beweis** durch Konstruktion:

Umwandlung einer Grammatik in CNF I

1. Schritt: Entfernung aller nicht-produktiven Variablen und aller Produktionen, in denen diese Variablen vorkommen.

   Die Menge V_i der produktiven Variablen kann iterativ bestimmt werden:

   $$V_0=\{A\in V \mid \exists\, w\in\Sigma^*\colon (A\rightarrow w)\in P\}$$

   $$V_i=\{A\in V \mid \exists\, w\in(V_{i-1}\cup\Sigma)^*\colon (A\rightarrow w)\in P\}$$

   Erläuterung: Ausgehend von der Menge V_0 der Variablen, die mit einem Ableitungsschritt zu einem Wort ausschließlich bestehend aus Terminalen abgeleitet werden können, wird die Menge der produktiven Variablen bestimmt.

   In jedem Schritt wird die Menge V_{i-1} um die Variablen erweitert, die zu einem Wort bestehend aus der ausschließlich Terminalen, sowie den Variablen aus der Menge der bereits als produktiv erkannten Variablen, also V_{i-1}, abgeleitet werden können.

   Die Iteration wird abgebrochen, wenn sich die Menge V_i nicht mehr verändert, also der Fixpunkt der Funktion erreicht ist.

   Anmerkung: Durch das Entfernen der nicht-produktiven Variablen und entsprechender Produktionen wird die akzeptierte Sprache nicht verändert.

Umwandlung einer Grammatik in CNF II

2. Schritt: Entfernung aller nicht-erreichbaren Variablen und aller Produktionen, in denen diese Variablen vorkommen.

   Die Menge der erreichbaren Variablen V_i kann iterativ bestimmt werden:

$$V_0=\{S\}$$

$$V_i=\{Y\in V \mid \exists\, X\in V_{i-1},\, w_1, w_2\in(\Sigma\cup V)^*: (X\rightarrow w_1 Y w_2)\in P\}$$

   Erläuterung: Ausgehend von der Menge V_0 die lediglich aus der Startvariablen besteht, wird die Menge der erreichbaren Variablen bestimmt.

   In jedem Schritt wird die Menge V_{i-1} um die Variablen erweitert, die über *einen* Ableitungsschritt aus der Menge der erreichbaren Variablen, also V_{i-1}, abgeleitet werden können.

   Die Iteration wird abgebrochen, wenn sich die Menge V_i nicht mehr verändert, also der Fixpunkt der Funktion erreicht ist.

   Anmerkung: Durch das Entfernen der nicht-erreichbaren Variablen und entsprechender Produktionen wird die akzeptierte Sprache nicht verändert.

Umwandlung einer Grammatik in CNF III

3. Schritt: Entfernung aller ε-Produktionen.

   Die Menge der zu ε ableitbaren Variablen V_i kann iterativ bestimmt werden:

$$V_0=\{A\in V \mid (A\rightarrow\varepsilon)\in P\}$$

$$V_i=\{A\in V \mid \exists\, w\in V_{i-1}^*: (A\rightarrow w)\in P\}$$

   Erläuterung: Ausgehend von der Menge V_0 die direkt zu ε abgeleitet werden kann, wird die Menge der zu ε ableitbaren Variablen bestimmt.

   In jedem Schritt wird die Menge V_{i-1} um die Variablen erweitert, die über einen Ableitungsschritt in ein Wort bestehend aus ausschließlich zu ε ableitbaren Variablen, also V_{i-1}, abgeleitet werden können.

   Die Iteration wird abgebrochen, wenn sich die Menge V_i nicht mehr verändert, also der Fixpunkt der Funktion erreicht ist.

Umwandlung einer Grammatik in CNF IV

3. Schritt: Entfernung aller ε-Produktionen (Fortsetzung).

   Einzelschritte:

   3.1. Hinzufügen von neuen Produktionen:

   Für jede Produktion $X \rightarrow Y_1 Y_2 \ldots Y_n$ mit einer zu ε ableitbaren Variable Y_i fügen wir eine Produktion $X \rightarrow Y_1 \ldots Y_{i-1} Y_{i+1} \ldots Y_n$ ein.

   3.2. Eliminieren von ε Produktionen.

   3.3. Hinzufügen einer neuen Variablen S' und der Produktionen $S' \rightarrow S$ und $S' \rightarrow \varepsilon$, falls S zu ε ableitbar war.

   3.4. Entfernen eventuell nutzlos gewordener Variablen.

   Anmerkung: Durch dieser Schritte wird die akzeptierte Sprache nicht verändert.

Umwandlung einer Grammatik in CNF V

4. Schritt: Entfernung von Kettenproduktionen (A→B→C→c).

   4.1. Hinzufügen von Abkürzungen für alle Kettenproduktionen:

   $P' = P \cup \{X \to w \mid \exists\, X, Y \in V: X \Rightarrow^* Y \to w \text{ mit } |w| \geq 2 \text{ oder } w \in \Sigma^*\}$

   Im obigen Beispiel also A→c und B→c

   4.2. Entfernen aller Produktionen der Form $X \to Y$ mit $X, Y \in V$.

   Im obigen Beispiel also A→B und B→C

   4.3. Entfernen aller eventuell neu entstandenen nutzlosen Variablen.

Anmerkung: Durch das Entfernen von Kettenproduktionen wird die akzeptierte Sprache nicht verändert.

Umwandlung einer Grammatik in CNF VI

Endgültiges Umwandeln der Grammatik in CNF. Zur Erinnerung alle Regeln müssen die Form $A \rightarrow BC$ oder $A \rightarrow a$ haben. Wir haben bisher eine Grammatik ohne nutzlose Variablen und ε-Produktionen. Alle Produktionen sind von der Form $V \rightarrow w$ mit $|w| \geq 2$ oder $w \in \Sigma$.

5. Schritt: Einfügen von neuen Variablen X_a für jedes Terminalzeichen a und einer neuen Produktion $X_a \rightarrow a$.

   Ersetzen sämtlicher Terminalzeichen a in w bei Produktionen der Form $A \rightarrow w$ mit $|w| \geq 2$ durch X_a.

   $\Rightarrow$ Die einzigen Produktionen, auf deren rechten Seite nun Terminalzeichen a stehen sind in der Form $A \rightarrow a$.

6. Schritt: Ersetzen aller Produktionen $A \rightarrow A_1, A_2, \ldots A_n$ mit $n>2$ durch n-1 Produktionen:

$$A \rightarrow A_1 Y_1,\ Y_1 \rightarrow A_2 C_2,\ \ldots,\ C_{n-2} \rightarrow A_{n-1} A_n$$

wobei $C_1, \ldots C_{n-2}$ neue Zwischenvariablen sind.

Die Grammatik ist nun in Chomsky-Normalform.

Weitere Normalform: Greibach-Normalform

- Eine weitere Normalform für kontextfreie Grammatiken ist die **Greibach-Normalform**.
- Alle Produktionen einer Grammatik der Greibach-Normalform sind in der Form:

$$A \rightarrow aB_1 \ldots B_k$$

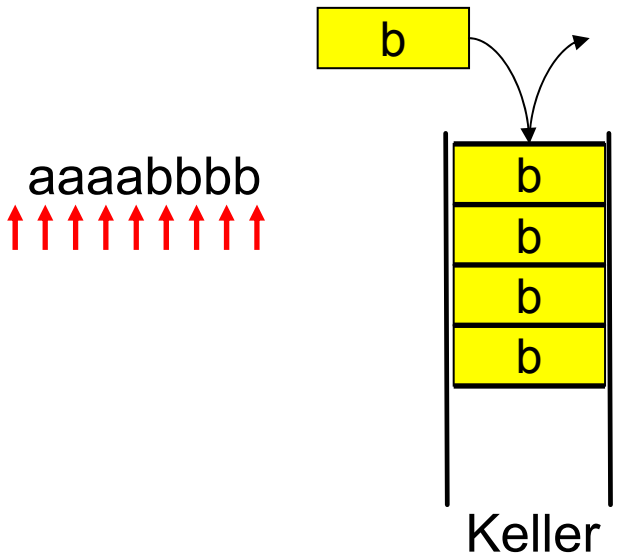
$$\text{mit } a \in \Sigma \text{ und } A, B_1, \ldots B_k \in V$$

- Es gilt die gleiche Ausnahmeregel bzgl. dem leeren Wort wie bei der Chomsky-Normalform.
- Vorteil der Greibach-Normalform: in jedem Ableitungsschritt wird genau ein Terminalzeichen erzeugt $\Rightarrow$ Die Anzahl der Ableitungsschritte ist wie bei der CNF in Bezug auf das Wortproblem bekannt und zusätzlich kann die Menge der potentiellen Ableitungen stark eingeschränkt werden.
- Jede kontextfreie Sprache besitzt eine Grammatik in Greibach-Normalform.
- Eine Beschreibung zur Umformung einer Grammatik in CNF in eine Grammatik in Greibach-Normalform findet sich beispielsweise unter http://de.wikipedia.org/wiki/Greibach-Normalform.

Kellerautomaten

Motivation

- Wie bei regulären Sprachen mit den endlichen Automaten wird ein Automatenmodell für kontextfreie Sprachen gesucht.
- Wie bereits bei dem Wort $a^n b^n$ gesehen, muss der Automat über die Fähigkeit des Zählens verfügen können.
- Nimmt man an, dass der Automat über einen unendlich großen Speicherplatz verfügt, so könnte er beispielsweise beim oben genannten Wort bei jedem a ein b auf diesen Speicherplatz legen (push). Folgt ein b, so könnte dieses b wieder von dem Speicherplatz gelöscht (pop) werden.
- Unendlich große Speicherplätze, bei denen immer nur auf das zuletzt eingefügte Element zugegriffen wird, werden in der Informatik Keller genannt.
- Der Automat besitzt nun neben den Zuständen wie bei endlichen Automaten einen Keller, den er zum Zählen verwenden kann.



Kellerautomat

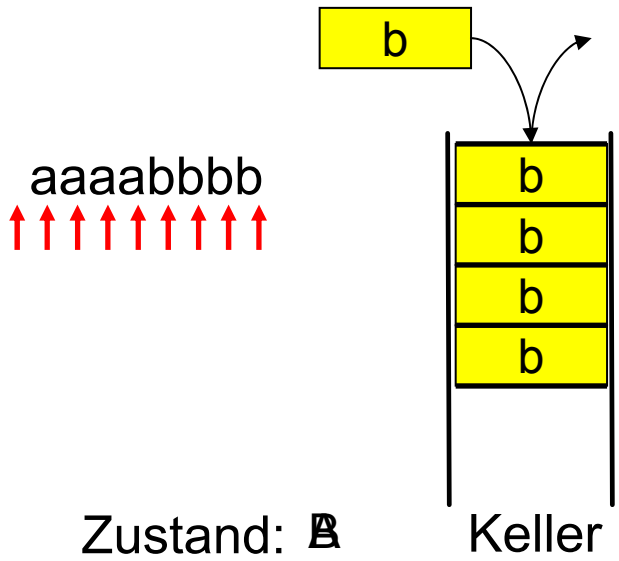
- Formale Definition: Ein Kellerautomat A über das Alphabet Σ kann durch das Tupel $(Q,q_0,\Gamma,Z_0,F,\delta)$ beschrieben werden. Dabei ist:
  - Q, eine endlichen Zustandsmenge $Q\cap\Sigma=\emptyset$
  - $q_0\in Q$ der Startzustand
  - Γ, das Kelleralphabet mit $\Gamma\cap Q=\emptyset$, $\Gamma=\Sigma$ ist möglich
  - $Z_0\in\Gamma$ das Anfangssymbol, typischerweise $\perp$ und wird ausschließlich für den leeren Keller verwendet.
  - $F\subseteq Q$ die Menge der Endzustände
  - $\delta\subseteq (Q \times \Sigma\cup\{\varepsilon\} \times \Gamma) \times (Q \times \Gamma^*)$ eine Übergangsrelation
- Erläuterung der Übergangsrelation $(q,a,\gamma)\rightarrow(q',\gamma')$: Als Eingabe bekommt der Automat jeweils den aktuellen Zustand q, das Eingabezeichen a und das oberste Element des Kellers γ. Erfolgt der Übergang, so ist das Ergebnis ein neuer Zustand q, und das Zeichen γ wird durch die Zeichenkette γ' auf dem Keller ersetzt ($\gamma'=\varepsilon$ bedeutet, dass das letzte Zeichen gelöscht wird).
- Die Übergangsrelation $(q,a,x)\rightarrow(q',ax)$ entspricht also beispielsweise einen `push`, während $(q,a,x)\rightarrow(q',\varepsilon)$ einem pop entspricht.
- Die von einem Kellerautomaten akzeptierte Sprache sind alle Wörter, nach deren Eingabe sich der Kellerautomat in einem Endzustand befindet und der Keller leer ist.

Kellerautomat: Beispiel I

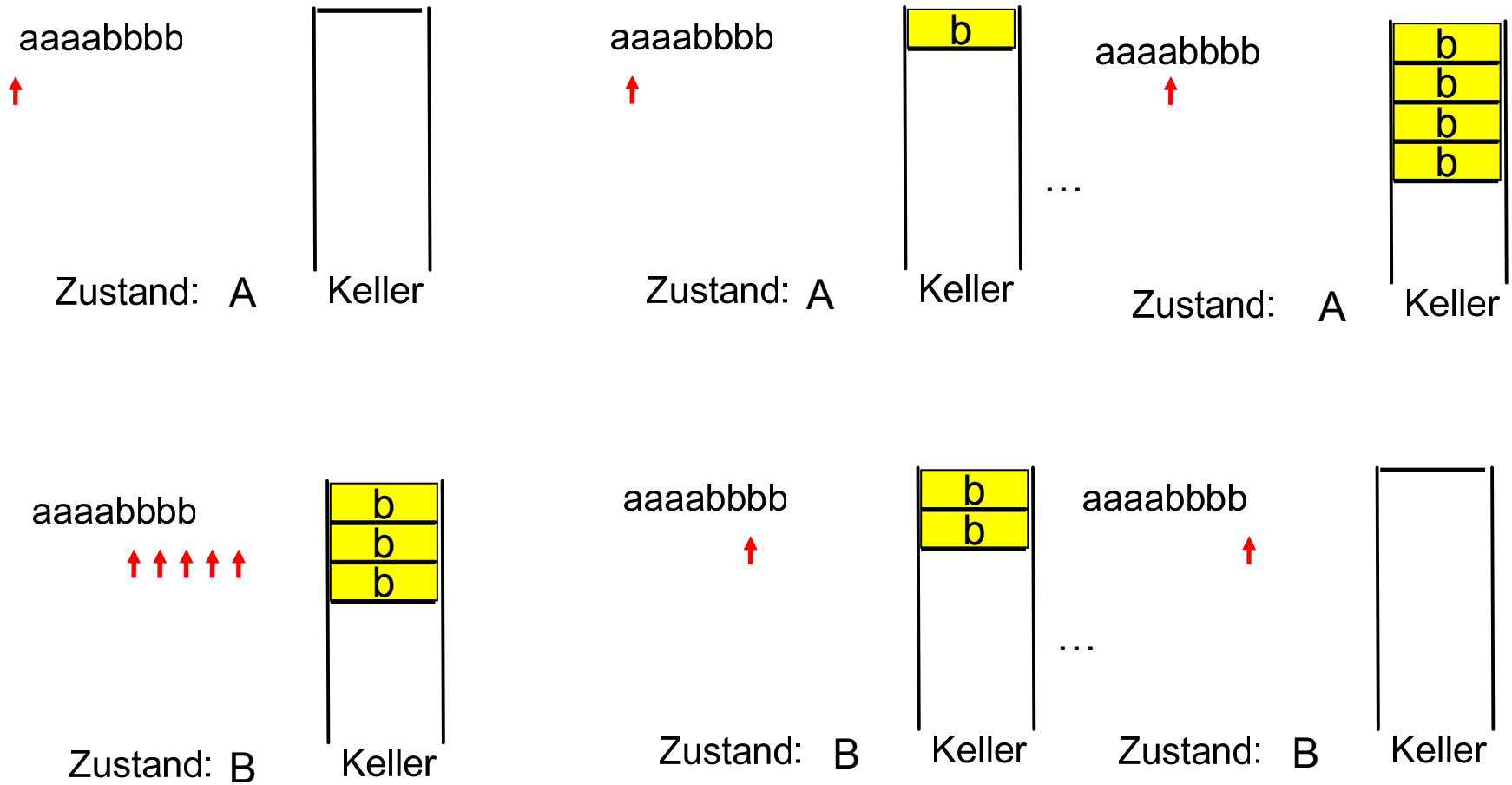
- Kellerautomat $(Q,q_0,\Gamma,Z_0,F,\delta)$, der alle Wörter a^nb^n akzeptiert:
  - $\Sigma=\{a,b\}$
  - $q_0=A$
  - $Q=\{A,B\}$
  - $\Gamma=\{b,\perp\}$
  - $Z_0=\perp$
  - $F=\{A,B\}$ A wegen leerem Wort
  - δ:

    $\{((A,a,x)\rightarrow(A,bx)|x\in\{b,\perp\})\} \cup$

    $\{((X,b,b)\rightarrow(B,\varepsilon)|X\in\{A,B\})\}$



Kellerautomat - Animation



Kellerautomat: Beispiel II

- Kellerautomat $(Q,q_0,\Gamma,Z_0,F,\delta)$, der als Wörter alle Palindrome mit den Buchstaben a,b akzeptiert:
  - $\Sigma=\{a,b\}$
  - $Q=\{q_0,q_1\}$ q_0 der Automat liest die erste Hälfte des Wortes, q_1 der Automat liest die 2. Hälfte
  - $\Gamma=\{a,b,\perp\}$
  - $Z_0=\perp$
  - $F=\{q_0,q_1\}$
  - δ:

$\{((q_0,x,y)\rightarrow(q_0,xy))\|x,y\in\Sigma\} \cup$	(1) In der ersten Hälfte werden die Buchstaben auf dem Keller gesichert.
$\{((q_0,x,y) \rightarrow(q_1,xy))\|x,y\in\Sigma\} \cup$	(2) Entscheidung Mitte (gerade Länge), Wechsel des Zustands und schreiben des Zeichens auf den Keller
$\{((q_0,x,y)\rightarrow(q_1,y))\|x,y\in\Sigma\} \cup$	(3) Entscheidung Mitte (ungerade Länge), Wechsel des Zustands und ignorieren des Zeichens
$\{((q_1,x,x)\rightarrow(q_1,\varepsilon))\|x\in\Sigma\}$	(4) Keller mit Eingabezeichen vergleichen und bei Gleichheit Zeichen vom Keller löschen

Äquivalenz von Kellerautomaten I

- Das Beispiel zur Sprache der Palindrome ist ein nicht-deterministischer Kellerautomat (siehe erste drei Regeln).
- Bei endlichen Automaten sind deterministische und nicht deterministische Automaten gleich mächtig. Gilt dies auch bei Kellerautomaten?

Äquivalenz von Kellerautomaten I

- Das Beispiel zur Sprache der Palindrome ist ein nicht-deterministischer Kellerautomat.
- Bei endlichen Automaten sind deterministische und nicht deterministische Automaten gleich mächtig. Gilt dies auch bei Kellerautomaten?
- **Nein**! Das Beispiel der Palindrome zeigt warum: Um die Mitte zu erkennen muss der Automat die ganze Eingabe kennen. Der Automat müsste also beim Lesen vorausschauen können.
- Vorausschauen kann **prinzipiell** realisiert werden: durch Speichern der gelesenen Zeichen in Zuständen kann die Auswertung der Zeichen zeitlich verzögert werden.
- **Aber**: da die Menge der Zustände endlich ist, ist die Anzahl der Zeichen, die ein Automat vorausschauen kann begrenzt. Im Fall der (prinzipiell nicht längenbeschränkten) Palindrome kann der Automat also nicht vorausschauen.
- Frage: Wieso werden die Zeichen zur Vorausschau nicht im Keller (unendlich groß) gespeichert?

Äquivalenz von Kellerautomaten II

- Die Zeichen können zur Vorschau auch nicht im Keller gespeichert werden, da dieser im Zugriff auf das zuletzt gespeicherte Element beschränkt ist.
- Nicht-deterministische Kellerautomaten sind somit mächtiger, als deterministische Kellerautomaten, allerdings nur als Gedankenexperiment realisierbar.
- Deshalb sind zur Konstruktion von Programmen zur Analyse von Wörtern im Bezug auf kontextfreie Grammatiken nicht-deterministische Kellerautomaten nicht geeignet: gewünscht sind Grammatiken, die mit maximal *k* Zeichen Vorschau, eine deterministische Auswahl der Produktionsregeln erlauben. Diese Grammatiken werden LR(*k*)-Grammatiken genannt.
- Ohne Beweis, weil zu kompliziert: LR(*k*) Grammatiken können immer zu LR(1)-Grammatiken umgewandelt werden. Aus diesen wiederum kann sehr leicht der passende Kellerautomat konstruiert werden.

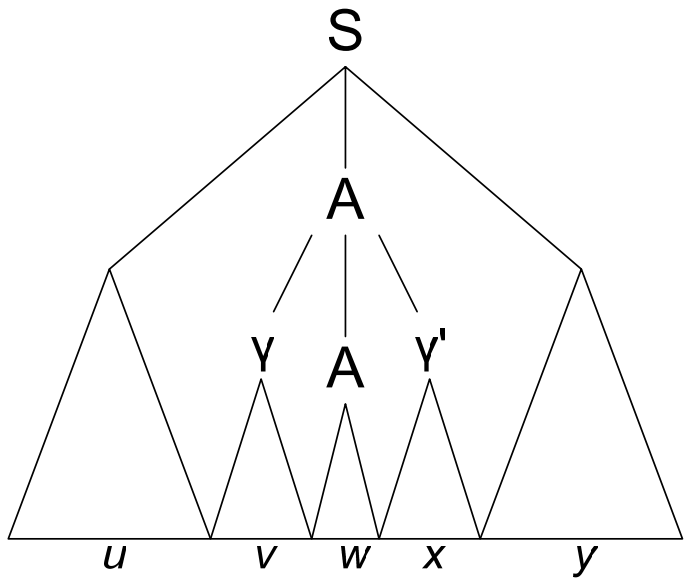
Eigenschaften kontextfreier Sprachen

Pumping-Lemma für kontextfreie Sprachen I

- Ähnlich wie bei dem Pumping-Lemma für reguläre Sprachen kann man auch bei kontextfreien Sprachen argumentieren:
  - Eine kontextfreie Sprache kann nur dann unendlich viele Elemente besitzen, wenn in der die Sprache beschreibende Grammatik G eine Ableitung $A \Rightarrow \gamma A \gamma'$, also ein Zyklus im Ableitungsbaum, gefunden werden kann.
  - Die Existenz einer solchen Ableitung entspricht dem Zyklus im endlichen Automaten.
  - Durch mehrfaches Durchlaufen dieses Zyklus können weitere Wörter gefunden werden, die zur Sprache gehören.
  - Man kann nun den dazugehörigen Ableitungsbaum betrachten, um den Effekt zu verdeutlichen.

Pumping-Lemma für kontextfreie Sprachen II

- Um eine unendliche Sprache L_∞ zu beschreiben, muss es Zyklen innerhalb der Produktionen geben $\Rightarrow$ es gibt Wörter, in deren Ableitungsbaum auf einem Pfad dieselbe Variable mehrfach vorkommt:
- Entsprechend der Abbildung können wir diese Worte in *uvwxy* aufteilen. Da man das Teilstück zwischen dem ersten Auftreten und dem zweiten Auftreten der Variable beliebig oft wiederholen können, gilt $uv^iwx^iy \in L_\infty$ für alle $i \in \mathbb{N}_0$.
- Überlegung: gibt es wieder ein entsprechendes *n* (analog zu dem Pumping-Lemma bei regulären Sprachen), so dass für alle Wörter *w* mit $|w| \geq n$ gilt, dass diese Wörter einen Zyklus enthalten?



Pumping-Lemma für kontextfreie Sprachen III

- Sei $G=(V,\Sigma,P,S)$ eine kontextfreie Grammatik in Chomsky-Normalform (ohne Beschränkung der Allgemeinheit) mit $|V|=m$.
- Es gelten folgende Eigenschaften:
  - Der Ableitungsbaum für ein Wort mit dieser Grammatik ist ein Binärbaum.
  - Das Wort enthält keine Zyklen, wenn auf jedem Pfad keine Variable doppelt vorkommt.
  - Die maximale Höhe für den Ableitungsbaum eines Wortes, das keine Zyklen enthält, kann also nur $h=m+1$ sein. (m Variablen + ein Terminalzeichen auf Pfad).
  - Die maximale Länge des Wortes (entspricht der maximalen Anzahl der Blätter) ist also auf 2^h-1 begrenzt.
  - Jedes Wort w mit einer Länge $|w| \geq 2^h = 2^{m+1} = n$ Blätter enthält mindestens einen Zyklus.
  - Entscheidener Punkt: Für eine gegebene Grammatik G mit $|V|=m$ haben wir nun ein n gefunden, für das gilt: jeder Ableitungsbaum für ein Wort w mit $|w| \geq n$ enthält einen Zyklus.

Pumping-Lemma für kontextfreie Sprachen IV

- Formalisierung des Pumping-Lemmas für kontextfreie Sprachen: Für jede kontextfreie Sprache L gibt es ein $n \in \mathbb{N}$, so dass für jedes $w \in$ L mit $|w| \geq n$ eine Zerlegung $w=uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$ existiert mit $uv^iwx^iy \in L$ für alle $i \in \mathbb{N}_0$.
- Erläuterungen:
  - n wird wie vorher beschrieben gewählt.
  - $|vx| \geq 1$ gilt, da Nullproduktionen der Form $A \rightarrow \varepsilon$ aufgrund der Chomsky-Normalform ausgeschlossen sind.
  - $|vwx| \leq$ n gilt, da wir die Zerlegung so wählen, dass auf dem Teilpfad zwischen ersten Vorkommen und zweiten Vorkommen der Variable, keine Variable doppelt vorkommt.

Anwendung des Pumping-Lemmas

- Beweis, dass die Sprache L $a^m b^m c^m$ nicht kontextfrei ist.
- Beweis durch Widerspruch:
  - Annahme: Jedes Wort w mit mehr Buchstaben als n kann entsprechend dem Pumping-Lemma in *uvwxy* zerlegt werden, mit $|vx| \geq 1$ und $|vwx| \leq n$, so dass gilt $uv^i wx^i y \in L$.
  - Wir betrachten nun das Wort $a^n b^n c^n$ mit |w|=3n.
  - Für jede beliebige Zerlegung von w in *uvwxy* mit $|vx| \geq 1$ und $|vwx| \leq n$, gilt wegen $|vwx| \leq n$, dass v und x nur a´s und b´s, b´s und c´s oder nur b´s enthalten können. Wegen $|vx| \geq 1$ müssen v oder x mindestens ein Zeichen haben.
  - Es muss entsprechend der Annahme gelten, dass $uv^*wx^*y \in L$. Gegenbeispiel uv^2wx^2y enthält entweder mehr a´s als c´s (falls v oder x a´s enthalten), mehr c´s als a´s (falls v oder x c´s enthalten) oder mehr b´s als a´s bzw. c´s (falls v und x nur b´s enthalten) und ist somit nicht in der Sprache L enthalten.

Abschlusseigenschaften

Kontextfreie Grammatiken sind abgeschlossen unter (Beweis durch Konstruktion der entsprechenden Grammatik):

- Vereinigung, Idee $S \rightarrow S_1 \mid S_2$
- Konkatenation, Idee $S \rightarrow S_1 S_2$
- Kleensche Hülle: $S \rightarrow \varepsilon \mid S_1 S$

Kontextfreie Grammatiken sind nicht abgeschlossen unter:

- Durchschnitt: $L_1 = a^m b^n c^n$, $L_2 = a^n b^n c^m$ sind beide kontextfrei, der Durchschnitt $L_1 \cap L_2 = a^n b^n c^n$ aber nicht.
- Komplement: Kann nicht abgeschlossen sein, da sonst auch der Durchschnitt abgeschlossen wäre, da gilt: $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$

Entscheidungsverfahren I

- Leerheit $L=\emptyset$: entscheidbar

  Verfahren: Bestimmung der produktiven Variablen wie bereits kennengelernt. Ist S in den produktiven Variablen enthalten, so ist die Sprache nicht leer, andernfalls ist die Sprache leer.

- Endlichkeit $|L| = m$: entscheidbar

  Verfahren: Konstruktion eines Graphen aus der Grammatik in CNF. Der Graph enthält die Kante (A,B), falls die Grammatik eine Produktion der Form $A \rightarrow BC$ oder $A \rightarrow CB$ besitzt. Enthält der Graph einen Zyklus, so ist die Sprache L unendlich.

- Nicht entscheidbar sind Disjunktheit, Inklusion und Äquivalenz (ohne Beweis).

Entscheidungsverfahren II

- Wortproblem:
  - Wie bereits gesehen, kann das Wortproblem durch Bestimmung aller Ableitungen einer bestimmten Länge gelöst werden. Problem: sehr ineffizient.
  - Effizientere Lösung mit Hilfe des Algorithmus von Cocke, Younger und Kasami (CYK-Algorithmus)
  - Prinzip: Aufbau des Ableitungsbaums von unten
  - Voraussetzung: Grammatik $G=(V,\Sigma,P,S)$ in CNF
  - Idee: Für das Wort $w=a_1 \ldots a_n$ bestimmt man für alle $j \geq i \geq 1$ die Mengen $V_{i,j}=\{A \in V \mid A \Rightarrow^* a_i \ldots a_j\}$
  - Es gilt $w \in L$, genau dann, wenn $S \in V_{1,n}$
  - Die Mengen $V_{i,j}$ können systematisch bestimmt werden:

$$V_{i,i} = A \in V | A \rightarrow a_i$$

$$V_{i,j} = \bigcup_{k=i}^{j-1} \{A \in V | A \rightarrow BC \in P, B \in V_{i,k}, C \in V_{k+1,j}\}$$

CYK-Algorithmus am Beispiel

- Sprache: $a^n b^m c^n$
- Grammatik:

  $S \rightarrow ASC \mid ABC$

  $A \rightarrow a$

  $B \rightarrow BB \mid b$

  $C \rightarrow c$
- Grammatik in CNF:

  $S \rightarrow AD$

  $D \rightarrow SC \mid BC$

  $A \rightarrow a$

  $B \rightarrow BB \mid b$

  $C \rightarrow c$

- Test des Wortes: aabcc

	a	a	b	c	c
i \ j	1	2	3	4	5
1	A	-	-	-	S
2		A	-	S	D
3			B	D	-
4				C	-
5					C

Pseudo-Code des CYK-Algorithmus

```
for i:= 1 to n do                    //Initialisierung der Diagonale
    V_{i,i} := { A | A→a_i ∈P}
for d :=1  to n-1 do                 //d: Unterschied zwischen i und j
    for i :=1 to n-d do
        begin
            V_{i,i+d} := ∅
            for k := i to i+(d-1) do
                V_{i,i+d} := V_{i,i+d} ∪ {A | A→ BC ∈ P, B∈V_{i,k}, C∈V_{k+1,i+d}}
        end
```

Compilerbau

Compilerbau

- Ein **Compiler** / **Übersetzer** ist ein Computerprogramm, das ein in Quellcode geschriebenes Programm in ein semantisch übereinstimmendes Programm der Zielsprache umwandelt.
- Im Unterschied dazu arbeitet ein **Interpreter** ein Programm Schritt für Schritt durch die Ausführung der einzelnen Schritte ab.
- Die Ausführung eines Compilers kann in zwei Phasen unterteilt werden:
  - die **Analysephase** / **Frontend**: in dieser Phase wird das Programm auf syntaktische und semantische Korrektheit überprüft und ein Ableitungsbaum / Syntaxbaum erzeugt.
  - die **Synthesephase** / **Backend**: aus dem Syntaxbaum wird in der Synthesephase Code in der Zielsprache erzeugt. In dieser Phase kann der Code auch optimiert werden.
- Der erste Compiler wurde 1952 von Grace Hopper entwickelt.

Analysephase

- Die Analysephase dient zur Erstellung des Syntaxbaums und kann in 3 Teilphasen unterteilt werden:
  - **Lexikalische Analyse**: Unterteilung des Programms in einzelne Token verschiedener Klassen (Schlüsselwörter, Operatoren, Bezeichner, Konstanten,…). Das Programm Lex hilft bei der Erzeugung eines Programms zur lexikalischen Analyse eines Programms.
  - **Syntaktische Analyse**: Aufbau des Syntaxbaums, falls möglich. Andernfalls ist der Programmcode syntaktisch fehlerhaft. Diese Phase wird auch als Parsen bezeichnet. Das Programm Yacc hilft bei der Erzeugung eines Programms zur syntaktischen Analyse eines Programms.
  - **Semantische Analyse**: Überprüfung der statischen Semantik eines Programms: z.B. es müssen alle Variablen vor ihrer ersten Benutzung deklariert sein. Die Überprüfung wird durch Hinzufügen von Attributen (z.B. Liste aller deklarierten Variablen) an einen Syntaxbaum erreicht. Der mit Attributen versehene Syntaxbaum wird deshalb dekorierter Syntaxbaum genannt.

Lex

- Lex (Lexical Analyzer Generator) erstellt ein Programm zur lexikalischen Analyse.
- Als Eingabe erwartet Lex eine Tabelle von regulären Ausdrücken und korrespondierenden Codefragmenten / Token.
- Lex generiert aus dieser Tabelle ein Programm zur lexikalischen Analyse in C oder Ratfor (welche wiederum in portables Fortran umgewandelt werden kann) auf der Basis eines endlichen Automaten (mit Ausgabe).
- Eine ausführlichere Erläuterung zu Lex und der Syntax zur Beschreibung der Tabelle bzw. der regulären Ausdrücke findet sich unter http://dinosaur.compilertools.net/lex/.
- Download von Flex (für Windows) unter http://gnuwin32.sourceforge.net/packages/flex.htm

Yacc

- Yacc (Yet another compiler- compiler) generiert einen Parser für eine spezifizierte Sprache.
- Die Sprache wird als eine LR(1) Grammatik in einer Notation ähnlich BNF übergeben.
- Der Benutzer kann spezifizieren welche Aktionen das Programm vollführen soll, wenn eine Programmstruktur erkannt wird.
- Yacc erzeugt einen Parser typischerweise in C Code.
- Eine genaue Beschreibung von Yacc und die Syntax zur Spezifikation der Grammatik findet man unter: http://dinosaur.compilertools.net/#yacc
- Als frei herunterzuladendes Programm steht die Yacc-Implementierung Bison (für Windows) von GNU zur Verfügung: http://www.gnu.org/software/bison/
- Auch für OCaml stehen Versionen von Lex und Yacc bereit, siehe:

  http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html

Synthesephase

- Analog zur Analysephase kann die Synthesephase wiederum in drei Teilphasen unterteilt werden:
  - **Zwischencodeerzeugung**: Häufig wird auf der Grundlage eines Syntaxbaums zunächst ein maschinennaher Zwischencode erzeugt. Vorteile des Zwischencodes sind Möglichkeiten zur Optimierung, sowie Vorzüge bei der Übersetzung für unterschiedliche Plattformen.
  - **Programmoptimierung**: Typischerweise optimiert ein Compiler den entstehenden Code in Bezug auf die Laufzeit und den Speicherbedarf. Möglichkeiten sind eine optimierte Umsetzung von typischen Programmkonstrukten, Optimierung von Schleifen, Eliminierung von nicht genutzen Variablen und Programmcode, statische Auswertung von Ausdrücken.
  - eigentliche **Codegenerierung** in Form von ausführbaren Code, Bibliotheken oder Objektdateien.

Beispiel

- Mittels Lex und Yacc soll nun ein Taschenrechner implementiert werden.
- Der Taschenrechner soll alle Grundrechenarten (+,-,*,/) beherrschen, sowie beliebig geklammerte Ausdrücke verstehen.
- Die Auswertung soll nach gültigen Regeln erfolgen: Punkt-vor-Strich und sonst Auswertung von links nach rechts.
- Es muss nun zunächst eine eindeutige Grammatik entwickelt werden.
- Vorüberlegungen:
  - Zur Realisierung von Punkt-vor-Strich wird in der Grammatik ein Symbol benötigt, dass dafür sorgt, dass Multiplikationen und Divisionen vorrangig behandelt werden.
  - Die Auswertung des Terms soll sonst von links nach rechts erfolgen $\Rightarrow$ die Grammatik muss von rechts nach links expandieren (der oberste Knoten im Syntaxbaum wird schließlich zuletzt ausgeführt).

Grammatik für Taschenrechner

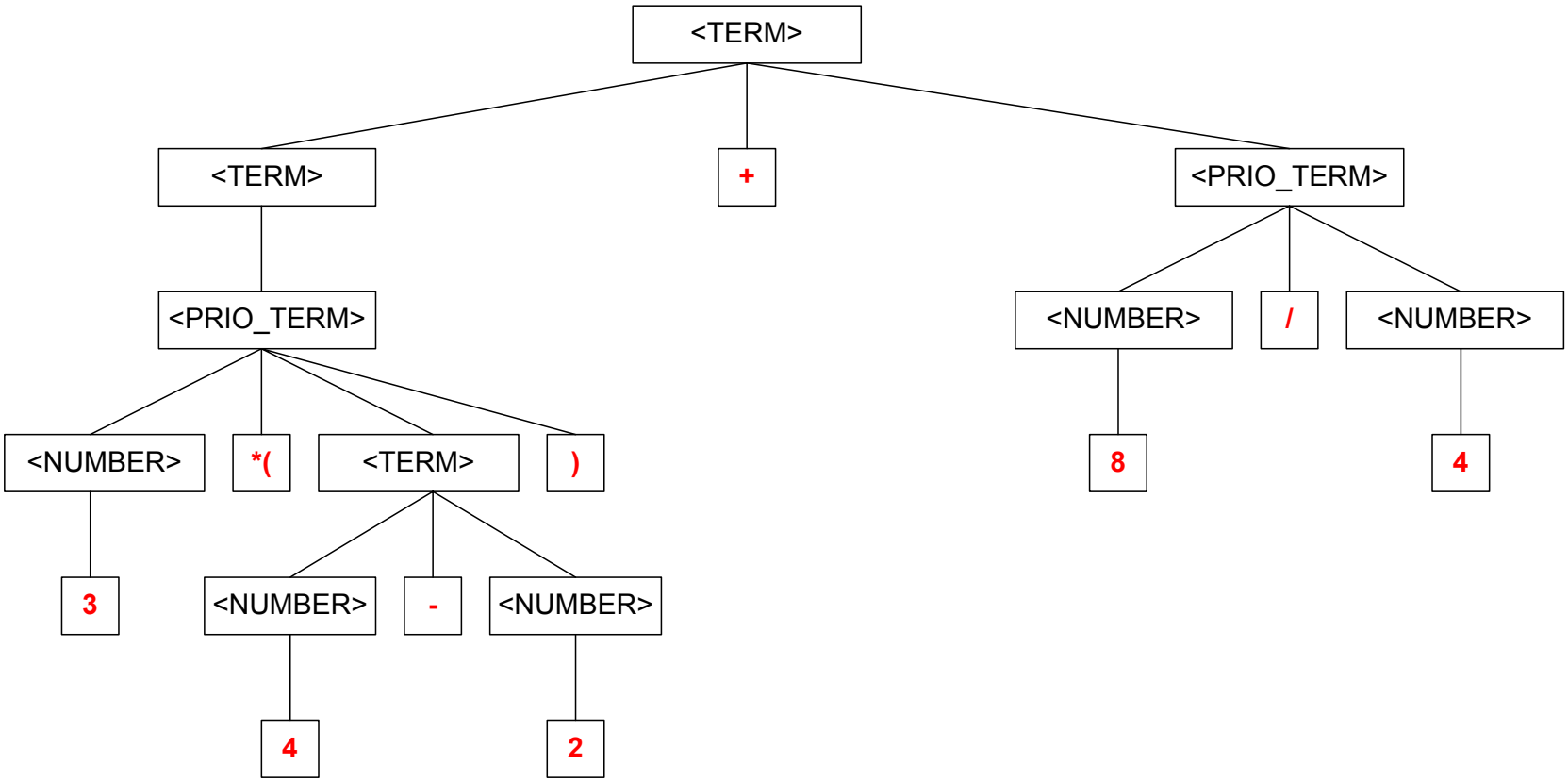
<TERM> := <TERM>+<NUMBER>
| <TERM>+<PRIO_TERM>
| <TERM>-<NUMBER>
| <TERM>-<PRIO_TERM>
| <PRIO_TERM>
| <NUMBER>

<PRIO_TERM>:= <PRIO_TERM>*<NUMBER>
| <NUMBER>*<NUMBER>
| <PRIOTERM>*(<TERM>)
| <NUMBER >*(<TERM>)
| <PRIO_TERM>/<NUMBER>
| <NUMBER>/<NUMBER>
| <PRIOTERM>/(<TERM>)
| <NUMBER >/(<TERM>)
| (<TERM>)

<NUMBER>:= $\{0,1,2,3,4,5,6,7,8,9\}^+$

Syntaxbaum: Beispiel

- Für den Term 3*(4-2)+8/4 ergibt sich somit folgender Baum:



Lexikalische Analyse mit Lex (Flex unter Windows)

- Ein Lex-Programm hat die folgende Form:

```
%{
Deklarationen / Definitionen
%}
%%
Regeln
%%
Unterprogramme
```

- Deklarationen / Definitionen und Unterprogramme werden als C-Code angegeben.
- Die Regeln bestehen aus regulären Ausdrücken (den Tokens) und den dazu gehörigen Anweisungen. Die Anweisungen sind wieder als C-Code anzugeben und werden ausgeführt, sobald der reguläre Ausdruck gefunden wird.

Lex-Datei für Taschenrechner

```
%{
#include <stdlib.h>           /*Einbinden der Header-Datei stdlib für die
                              Funktionen printf und strtol*/
int tokenValue;     /*Variable zum Speichern des Wertes einer Zahl*/
%}
%%
[0-9]+    {tokenValue=(int)strtol(yytext,NULL,10); return NUM;} /*Zahl gefunden, der
                                        Wert wird in tokenValue gespeichert*/
"+"       {return ADD;}               /*Pluszeichen gefunden*/
"-"       {return SUB;}               /*Minuszeichen gefunden*/
"/"       {return DIV;}               /*Divisionszeichen gefunden*/
"*"       {return MUL;}               /*Multiplikationszeichen gefunden*/
"("       {return LEFT;}              /*Linke Klammer gefunden*/
")"       {return RIGHT;}             /*Rechte Klammer gefunden*/
"\n"      {return END;}               /*Zeilenende gefunden*/
%%
int yywrap(){
   return(1);
}
```

Beim Aufruf von flex wird die Datei lex.yy.c erzeugt.

Syntaktische Analyse mit Yacc (Bison unter Windows)

- Ein Yacc-Programm hat die folgende Form:

```
%{
C-Deklarationen / C-Definitionen
%}
Yacc-Deklarationen
%%
Regeln
%%
Unterprogramme
```

- Die C-Deklarationen / C-Definitionen werden direkt in die generierte Datei kopiert.
- Die Yacc-Deklarationen dienen zur Festlegung des Startsymbols und der Tokens.
- Die Yacc-Regeln werden ähnlich zu BNF notiert: Die linke und die rechte Seite werden durch „:“ getrennt, Alternativen werden mit „|“ notiert. Das Ende einer Regel wird durch ein Semikolon markiert.
- Jeder Produktionsregel werden C-Anweisungen zugeordnet, die im Fall der Anwendung der entsprechenden Regel ausgeführt wird.
- Der Ausführung der Regel kann ein Wert innerhalb der C-Anweisungen zugeordnet werden: $$ steht dabei für den Wert der Aktion, mit $1,$2,…$n werden die Werte der einzelnen Symbole auf der rechten Seite der Produktion bezeichnet.

Yacc-Datei für Taschenrechner

```
%{#include "lex.yy.c " void yyerror(char* s);%}
%start s
%token ADD SUB DIV MUL NUM RIGHT LEFT END
%%
s:              normal_term END                          {printf("Ergebnis: %d\n",$1); return 0;};
normal_term:    normal_term ADD number                   {$$=$1+$3;}
        |       normal_term ADD prio_term                {$$=$1+$3;}
        |       normal_term SUB number                   {$$=$1-$3;}
        |       normal_term SUB prio_term                {$$=$1-$3;}
        |       prio_term                                {$$=$1;}
        |       number                                   {$$=$1;};
prio_term:      prio_term MUL number                     {$$=$1*$3;}
        |       number MUL number                        {$$=$1*$3;}
        |       prio_term MUL LEFT normal_term RIGHT     {$$=$1*$4;}
        |       number MUL LEFT normal_term RIGHT        {$$=$1*$4;}
        |       prio_term DIV number                     {$$=$1/$3;}
        |       number DIV number                        {$$=$1/$3;}
        |       prio_term DIV LEFT normal_term RIGHT     {$$=$1/$4;}
        |       number DIV LEFT normal_term RIGHT        {$$=$1/$4;}
        |       LEFT normal_term RIGHT                   {$$=$2;};
number:         NUM                                       {$$=tokenValue;};
%%
void yyerror(char* s){printf("%s\n",s);}
int main(){yyparse();return 0;}
```

Erweiterter Taschenrechner

- Im Folgenden soll der Taschenrechner um Speichermöglichkeiten erweitert werden.
- Zur Speicherung eines Ausdrucks stehen vier Register R0 bis R3 zur Verfügung.
- Ein Programm besteht nun aus mehreren Zuweisungen, die sequentiell abgearbeitet werden.
- Da nun mehrere Zuweisungen hintereinander möglich sind, muss der Programmierer den Start und das Ende der Zuweisungen und des Programms markieren.
- Das Programm startet mit `BEGIN` und endet mit `END`. Jede Anweisung wird durch ein Semikolon beendet.
- Die Register können jeweils auf der linken und rechten Seite der Zuweisung positioniert sein.

taschenrechner.l

```
…
%%
[0-9]+    {tokenValue=(int)strtol(yytext,NULL,10); return NUM;}
"+"       {return ADD;}
"-"       {return SUB;}
"/"       {return DIV;}
"*"       {return MUL;}
"("       {return LEFT;}
")"       {return RIGHT;}
"="       {return ASSIGN;}
";"       {return END_ASSIGN;}
"R0"      {return REG0;}
"R1"      {return REG1;}
"R2"      {return REG2;}
"R3"      {return REG3;}
"BEGIN"   {return START;}
"END"     {return END;}
%%
…
```

taschenrechner.y

```
int my_register[4];
…
%start prog
%token START END ADD SUB DIV MUL NUM RIGHT LEFT ASSIGN END_ASSIGN REG0 REG1 REG2 REG3
%%
prog:          START assignment END      {printf("R0: %d\n",my_register[0]);
                                          printf("R1: %d\n",my_register[1]);
                                          printf("R2: %d\n",my_register[2]);
                                          printf("R3: %d\n",my_register[3]);
                                          return 0;};
assignment:      assignment regnumber ASSIGN term END_ASSIGN          {my_register[$2]=$4;}
               | regnumber ASSIGN term END_ASSIGN                    {my_register[$1]=$3;};
term:          …
prio_term:     …
number:          NUM          {$$=tokenValue;}
               | REG0         {$$=my_register[0];}
               | REG1         {$$=my_register[1];}
               | REG2         {$$=my_register[2];}
               | REG3         {$$=my_register[3];};
regnumber:       REG0         {$$=0;}
               | REG1         {$$=1;}
               | REG2         {$$=2;}
               | REG3         {$$=3;};
…
```

Beispielaufruf für erweiterten Taschenrechner

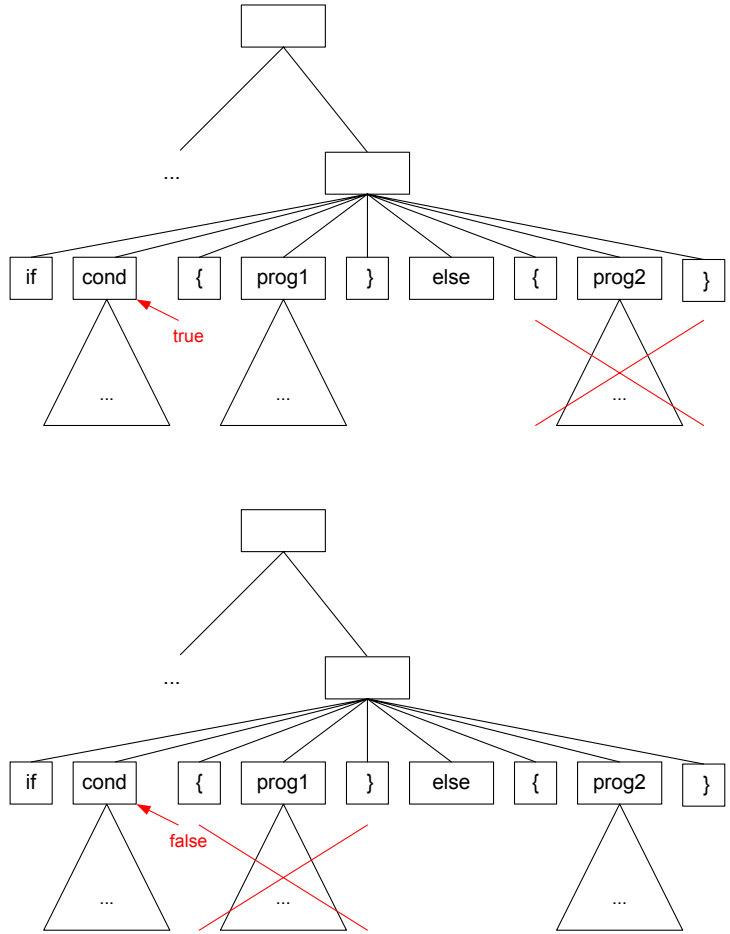
```
BEGIN R0=1; R1=2; R2=3; R3=(R0+R1)/R3; END
```

Ausgabe:

```
R0:1
R1:2
R2:3
R3:1
```

Einschränkungen bzgl. Lex und Yacc

- Lex und Yacc dienen nur der lexikalischen Analyse und der Erstellung des Ableitungsbaums. Ein kompletter Compiler kann mit ausschließlicher Verwendung von Lex und Yacc also nicht erstellt werden.
- Das Taschenrechnerbeispiel führt leicht zu der gegenteiligen Annahme. Zur Begründung der obigen Aussage kann der Versuch unternommen werden den Taschenrechner um if- und else-Anweisungen zu erweitern.
- Zur Erweiterung des Taschenrechners um beliebig verschachtelte `if cond {prog1} else {prog2}` Konstrukte, muss es möglich sein, abhängig vom Ergebnis der Auswertung von `cond` nur die Unterbäume `prog1` oder `prog2` auszuwerten.
- If- und else- Anweisungen können nicht mit Hilfe von Yacc realisiert werden, da es keine Möglichkeit gibt Unterbäume des Ableitungsbaums nicht auszuwerten.

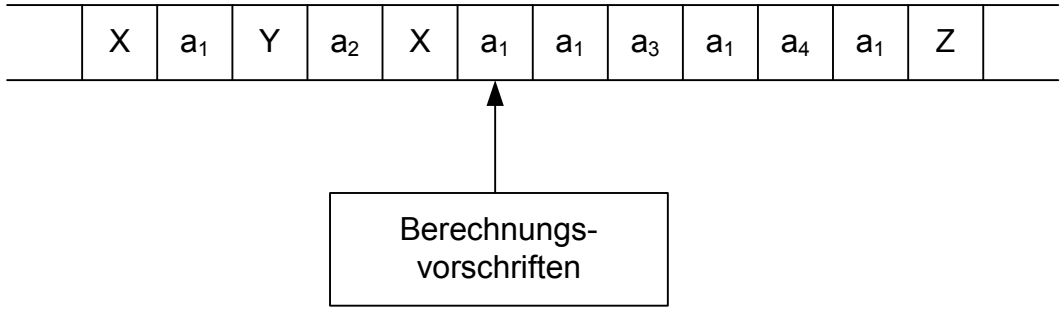


Turing-Maschine

Turing-Maschine

- Die passenden Automaten für kontextfreie Sprachen (Kellerautomat) und reguläre Sprachen (endl. Automat) wurden bisher vorgestellt.
- Die Turing-Maschine ist der Automat für beliebige Sprachen und stellt einen universellen Automaten (folgend der Church-Turing-These), also ein Automat, mit dem jeder andere praktisch realisierbare Rechner simuliert werden kann.
- Mit Hilfe der Turing-Maschine kann der Begriff der Berechenbarkeit erklärt werden: alle Funktionen die berechenbar sind, können mit der Turing-Maschine berechnet werden.
- Die Turing-Maschine zeichnet sich aus durch:
  - unendlichen Speicherplatz in Form eines Bandes, daher auch beliebig viel Platz für Nebenrechnungen
  - zu Beginn der Ausführung steht das vollständige Eingabewort auf dem Band, überall sonst ist das Band leer
  - Steuerung der Maschine durch **endliche** Berechnungsvorschriften
  - lokale Symbolmanipulation, d.h. die Turing-Maschine kann zu einem Zeitpunkt immer nur ein Zeichen lesen und verändern

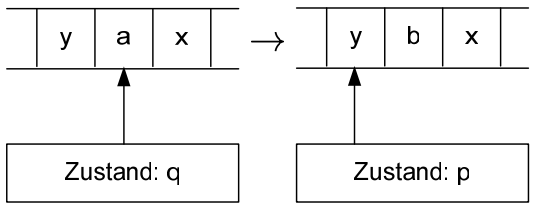
Turing-Maschine



- Die Maschine besteht aus einem Band mit unendlich vielen Feldern zum Speichern der Symbole und einem Lese- und Schreibkopf, der genau auf ein Feld zeigt.
- In jedem Schritt kann die Maschine also nur das Feld, auf das der Lese-/Schreibkopf zeigt, lesen und verändern.
- Der Kopf kann bei jedem Zustandsübergang um ein Feld nach links oder rechts bewegt werden, oder er bleibt stehen.
- Die Berechnungsvorschriften wird durch einen endlichen Automaten repräsentiert.
- D.h. die Regelbasis zur Manipulation des Bandfelder und zum Bewegen des Zeigers ist endlich.
- Der Zustandsübergang, also die Aktion der Maschine, ist nur vom aktuell gelesenen Zeichen und dem Zustand der Maschine abhängig.
- Applets zu Turing-Maschinen finden Sie unter http://ais.informatik.uni-freiburg.de/turing-applet/turing/TuringMachineHtml.html oder http://ironphoenix.org/tril/tm/

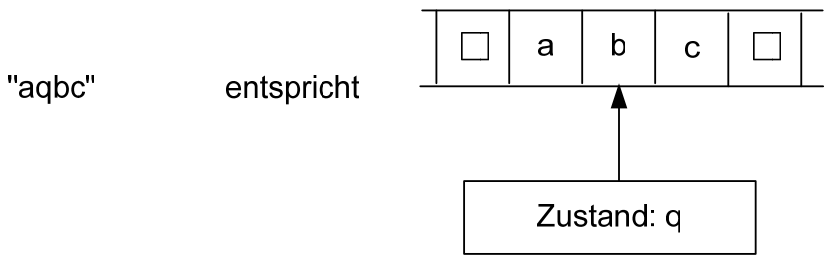
Turing-Maschine: Formale Definition

- Eine Turing-Maschine $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$ über dem Alphabet Σ besteht aus:
  - einer endlichen Menge von Zuständen Q,
  - einem Bandalphabet Γ mit $\Sigma\subseteq\Gamma$, $Q\cap\Gamma=\emptyset$ und $B\in\Gamma\setminus\Sigma$ wobei $B=\square$ Blank genannt wird,
  - einer Zustandsübergangsfunktion $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{l,r,n\})$,
  - einem Startzustand $q_0\in Q$,
  - und einer Menge von Endzuständen $F\subseteq Q$.
- Erläuterung der Zustandsübergangsfunktion $(q,a)\rightarrow(p,b,l)$
  - Vor Ausführen der Übergangsfunktion befindet sich die Turingmaschine im Zustand q und liest das Zeichen a.
  - Mit der Ausführung der Übergangsfunktion wechselt der Zustandsautomat der Turingmaschine in den Zustand p, überschreibt das Zeichen $a\in\Gamma$ mit b und bewegt danach den Schreibkopf nach links (alternativ auch nach rechts (r) oder keine Bewegung (n)).



Turing-Maschine: Konfiguration

- Der Status einer Turing-Maschine kann durch den Zustand des Bandes, die aktuelle Zeigerposition, sowie den aktuellen Zustand des endlichen Automaten beschrieben werden und wird *Konfiguration* genannt.
- Da $Q \cap \Gamma = \emptyset$ gilt, kann die Konfiguration durch das Wort $\alpha q \beta$ mit $\alpha, \beta \in \Gamma^*$ und $q \in Q$ beschrieben werden. Man markiert also die Position mit dem Zustand (links eingefügt neben Zeichen an Position).



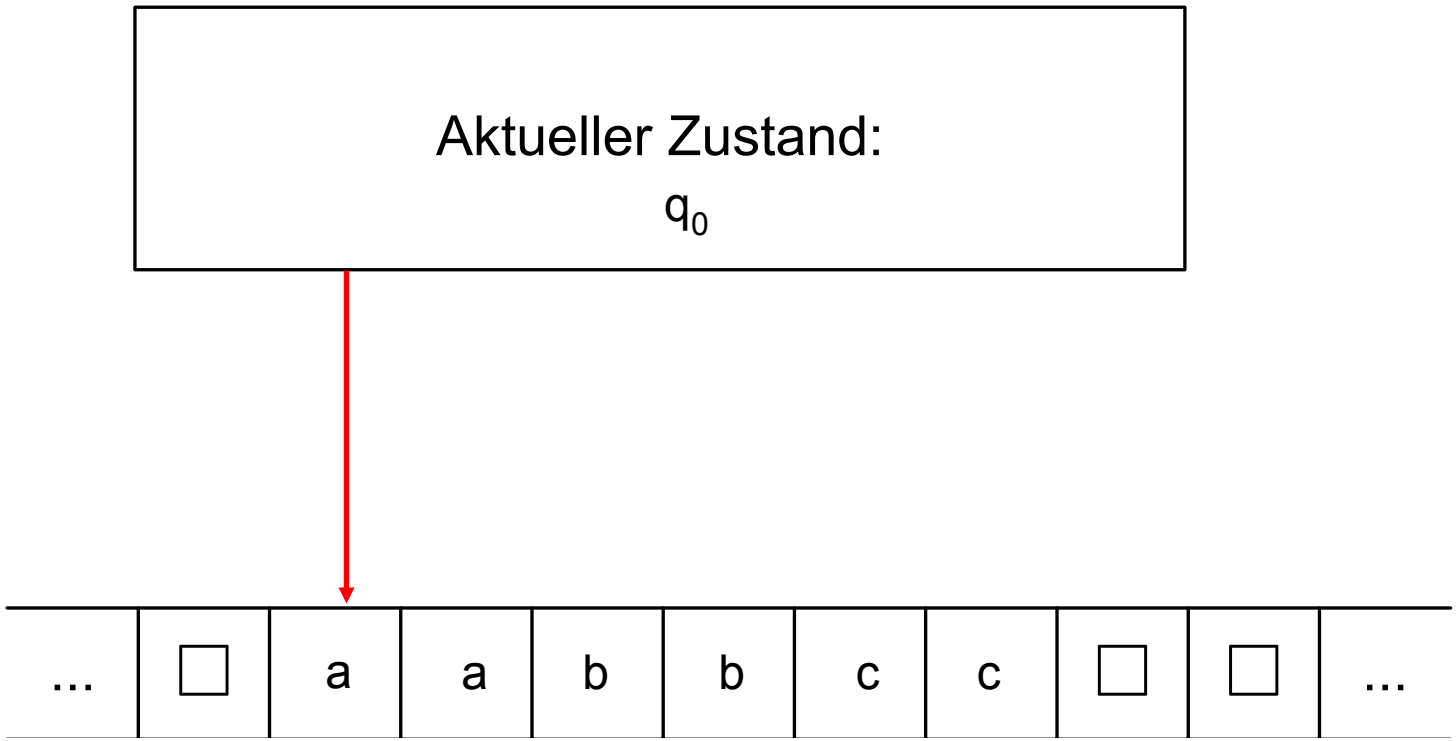
- Die Startkonfiguration für ein Wort *w* ist definiert als $q_0 w$.

Turing-Maschine: Akzeptierte Sprache

- Die Nachfolgekonfiguration einer Konfiguration ergibt sich aus der Zustandsübergangsfunktion:
  - Sei $\alpha=a_1\ldots a_n$ und $\beta=b_1\ldots b_m$ mit $n,m\geq 1$ und $((q,b_1),(p,b_1',l))\in\delta$, so ist $a_1\ldots a_{n-1}\mathbf{p}a_nb_1'b_2\ldots b_m$ Nachfolgekonfiguration von $\alpha q\beta$
  - Sei $\beta=b_1\ldots b_n$ mit $n\geq 1$ und $((q,b_1),(p,b_1',r))\in\delta$, so ist $\alpha b_1'\mathbf{p}b_2\ldots b_n$ von $\alpha q\beta$
  - Sei $\alpha=\varepsilon$ und $\beta=b_1\ldots b_n$ mit $n\geq 1$ und $((q,b_1),(p,b_1',l))\in\delta$, so ist $\mathbf{p}\square b_1'b_2\ldots b_n$ von $q\beta$
  - Sei $\beta=b_1\ldots b_n$ mit $n\geq 1$ und $((q,b_1),(p,b_1',n))\in\delta$, so ist $\alpha pb_1'b_2\ldots b_n$ Nachfolgekonfiguration von $\alpha q\beta$
  - γ ist Nachfolgekonfiguration von αq, wenn γ Nachfolgekonfiguration von $\alpha q\square$ ist.
- Ein Wort wird von der Turing-Maschine akzeptiert, wenn es ausgehend von der Startkonfiguration eine Folge von Übergängen in eine Konfiguration gibt, in der die Maschine sich in einem Endzustand befindet.
- Falls ein Wort durch eine Turing-Maschine akzeptiert wird, sagt man auch, die Maschine hält.

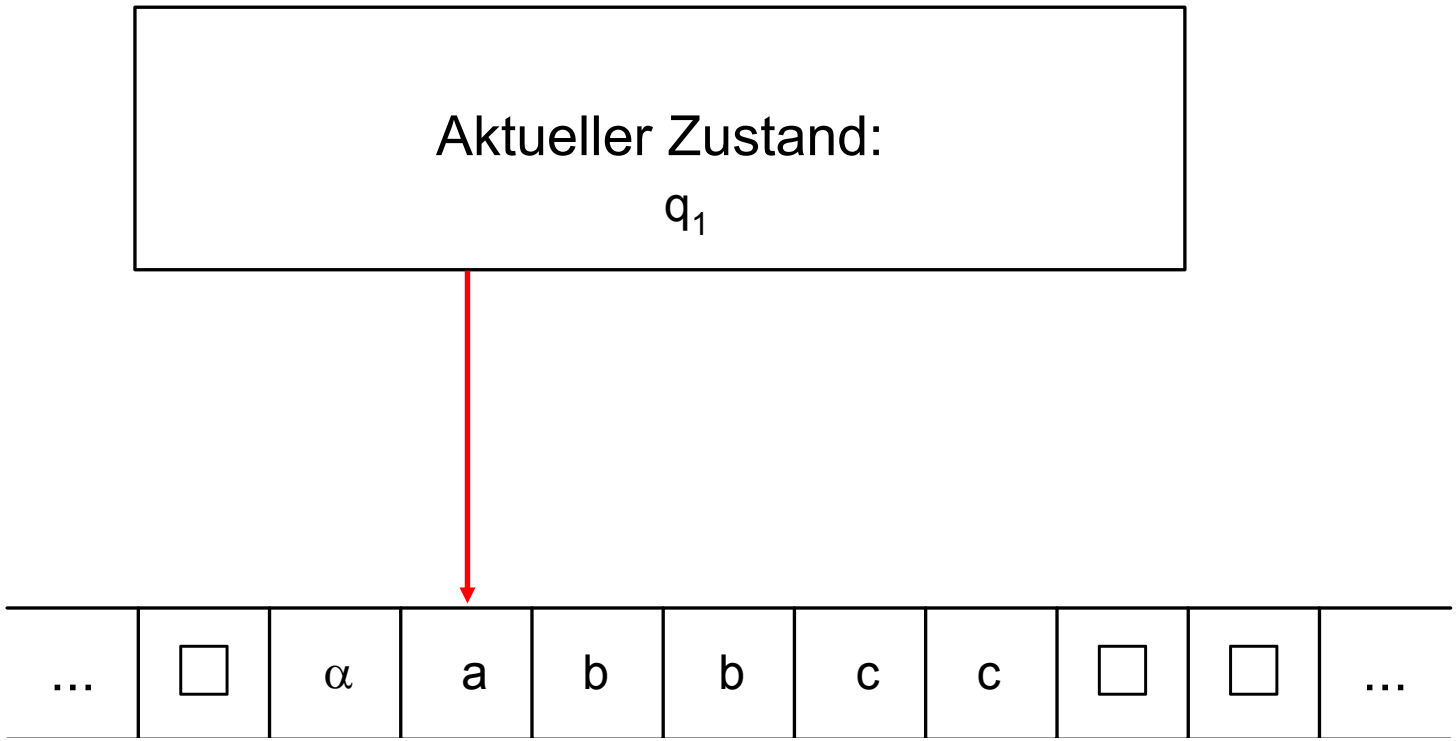
Turing-Maschine: Beispiel

- Turing-Maschine, die die Sprache $a^n b^n c^n$ akzeptiert.
- Idee: Man durchläuft das Wort und markiert pro Durchlauf jeweils ein a, b, c.
- Beispiel:



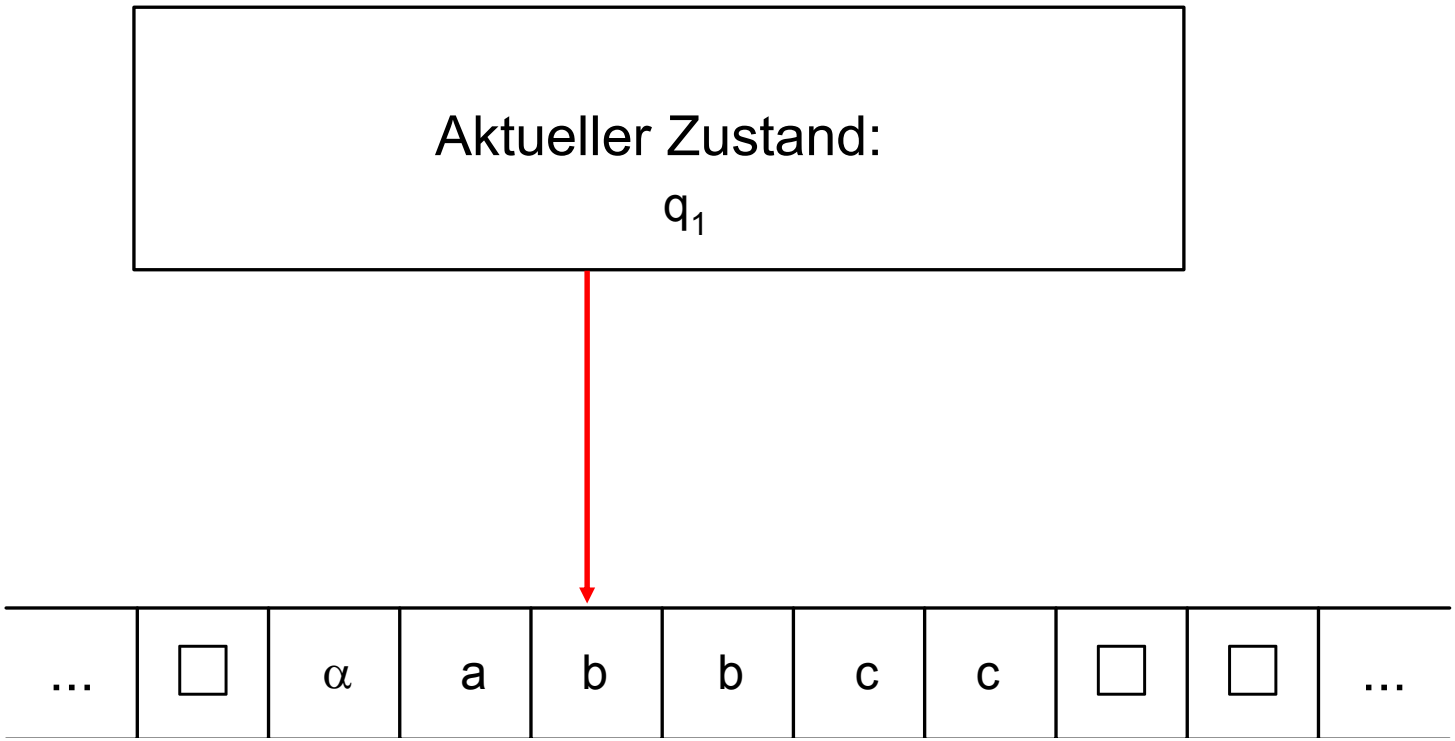
Turing-Maschine: Beispiel

- Turing-Maschine, die die Sprache $a^n b^n c^n$ akzeptiert.
- Idee: Man durchläuft das Wort und markiert pro Durchlauf jeweils ein a, b, c.
- Beispiel:



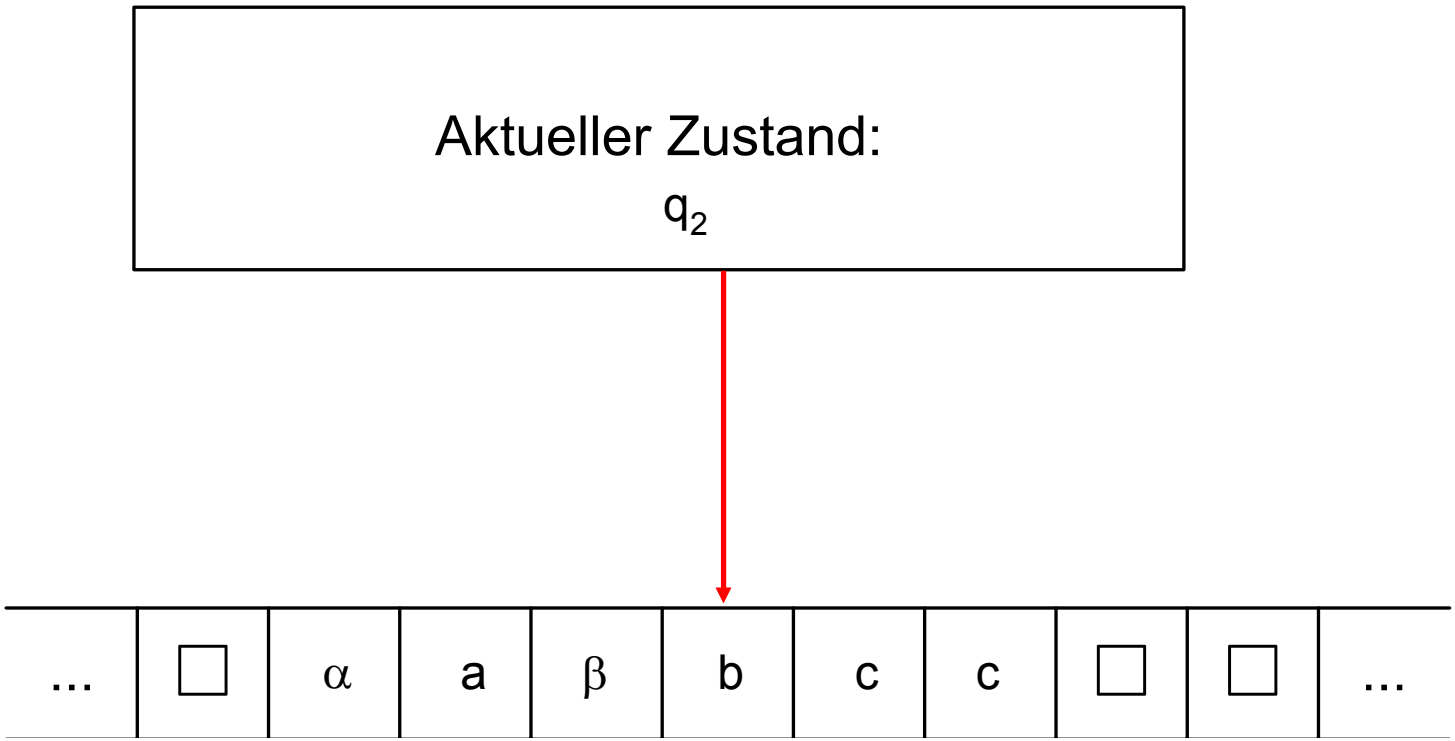
Turing-Maschine: Beispiel

- Turing-Maschine, die die Sprache $a^n b^n c^n$ akzeptiert.
- Idee: Man durchläuft das Wort und markiert pro Durchlauf jeweils ein a, b, c.
- Beispiel:



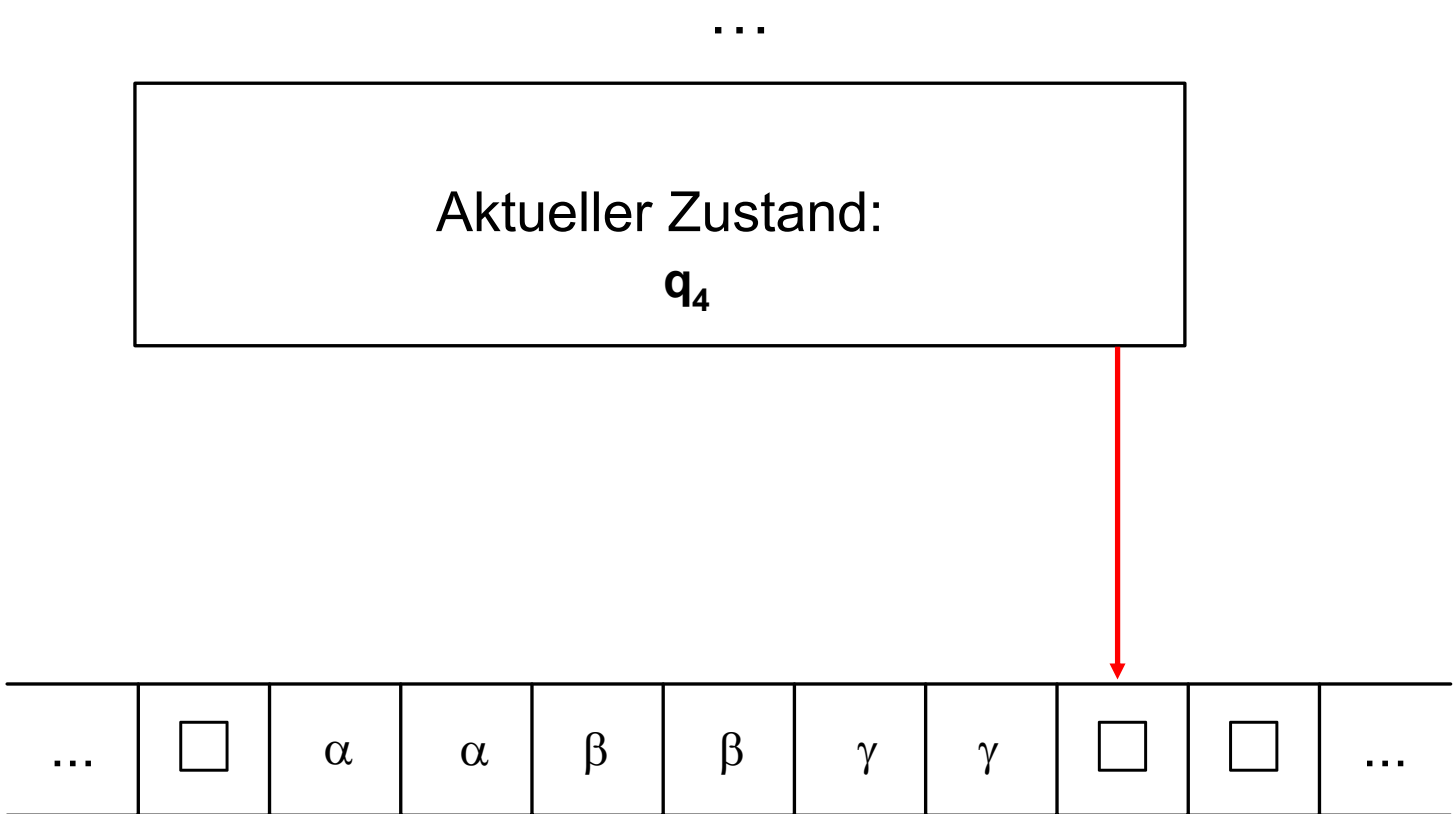
Turing-Maschine: Beispiel

- Turing-Maschine, die die Sprache $a^n b^n c^n$ akzeptiert.
- Idee: Man durchläuft das Wort und markiert pro Durchlauf jeweils ein a, b, c.
- Beispiel:



Turing-Maschine: Beispiel

- Turing-Maschine, die die Sprache $a^n b^n c^n$ akzeptiert.



Turing-Maschine: Beispiel

- Durch die Regeln wird sichergestellt, dass die Reihenfolge eingehalten wird.
- Mit folgender Turing-Maschine kann dieses Konzept realisiert werden:

  $M=(Q,\Sigma,\Gamma,\delta,q_0,\square,q_4\}$

  $Q=\{q_0,q_1,q_2,q_3,q_4\}$

  $\Gamma=\{a,b,c,\alpha,\beta,\gamma,\square\}$

  δ siehe nächste Folie

Turing-Maschine: Beispiel

Turing-Maschine, die die Sprache $a^n b^n c^n$ akzeptiert.

$\delta = \{((q_0,\square)\rightarrow(q_4,\square,n))\} \cup$	das leere Wort wird sofort akzeptiert und Maschine hält
$\{((q_0,a) \rightarrow(q_1,\alpha,r))\} \cup$	Markierung des ersten a´s
$\{((q_1,x) \rightarrow(q_1,x,r)) \mid x\in\{a,\beta\}\} \cup$	Weitere a´s und schon markierte b´s werden übersprungen
$\{((q_1,b) \rightarrow(q_2,\beta,r))\} \cup$	Das erste unmarkierte b wird markiert
$\{((q_2,x) \rightarrow(q_2,x,r)) \mid x\in\{b,\gamma\}\} \cup$	Weitere b´s und schon markierte c´s werden übersprungen
$\{((q_2,c) \rightarrow(q_3,\gamma,l))\}\cup$	Das erste unmarkierte c wird markiert und wir kehren zurück
$\{((q_3,x) \rightarrow(q_3,x,l)) \mid x\in\{a,b,\beta,\gamma\}\}\cup$	Unmarkierte a´s und b´s, sowie markierte b´s und c´s werden übersprungen
$\{((q_3,\alpha) \rightarrow(q_0,\alpha,r))\} \cup$	Das erste markierte a markiert den Umkehrpunkt
$\{((q_0,x) \rightarrow(q_0,x,r)\mid x\in\{\beta,\gamma\}\}$	Markierte b´s und c´s werden übersprungen, bis ein Blank gefunden wird (erste Regel)

Beispiel: Inkrementieren einer Binärzahl I

- Die Turing-Maschine erlaubt nicht nur die Überprüfung der Akzeptanz eines Wortes mit einer Sprache, sondern auch die Berechnungen.
- Beispiel: Programm zur Inkrementierung einer Binärzahl. Dabei steht das LSB (least significant bit) rechts.
- Beispiele: 1101 $\Rightarrow$ 1110, 1110 $\Rightarrow$ 1111, 1111 $\Rightarrow$ 10000
- Vorgehensweise:
  - Zunächst muss die Maschine das Ende der Eingabe finden.
  - Nun läuft die Maschine wieder nach vorne und überschreibt zunächst alle Einsen mit 0, solange bis eine 0 gefunden wird. Diese wird durch eine 1 überschrieben. Befindet sich in der ganzen Eingabe keine 0, so wird links von der Eingabe auf das leere Band eine 1 eingetragen.
  - Sobald das Band das linke Ende der Eingabe erreicht wird, nachdem evtl. die 1 eingetragen wurde, der Endzustand erreicht.
- Animation siehe: http://ais.informatik.uni-freiburg.de/turing-applet/turing/TuringMachineHtml.html

Beispiel: Inkrementieren einer Binärzahl II

- Mit folgender Turing-Maschine kann dieses Konzept realisiert werden:

$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,q_f\}$
$Q=\{q_0,q_1,q_2,q_f\}$
$\Gamma=\{0,1,\}$

δ :

$\{((q_0,x)\rightarrow(q_0,x,r)) \mid x\in\{0,1\}\} \cup$ Zu Beginn wird das rechte Ende der Eingabe gesucht

$\{((q_0,\square)\rightarrow(q_1,\square,l))\} \cup$ Rechtes Ende $\Rightarrow$ Zurücklaufen

$\{((q_1,1)\rightarrow(q_1,0,l))\} \cup$ Solange noch keine 0 gelesen wird, werden alle 1er überschrieben

$\{((q_1,0)\rightarrow(q_2,1,l))\} \cup$ Wird eine 0 gefunden, so wird diese mit einer 1 überschrieben.

$\{((q_1,\square)\rightarrow(q_f,1,n))\} \cup$ Wird das linke Ende der Eingabe gefunden, bevor eine 0 gelesen wird, so wird eine 1 ans linke Ende angefügt, die Berechnung ist fertig und die Maschine stoppt.

$\{((q_2,x)\rightarrow(q_2,x,l)) \mid x\in\{0,1\}\} \cup$ Das linke Ende wird gesucht

$\{((q_2,\square)\rightarrow(q_f,\square,n))\}$ Das linke Ende wird erreicht, die Maschine befindet sich im Endzustand und stoppt.

Problem der Fleißigen Biber (Busy Beaver)

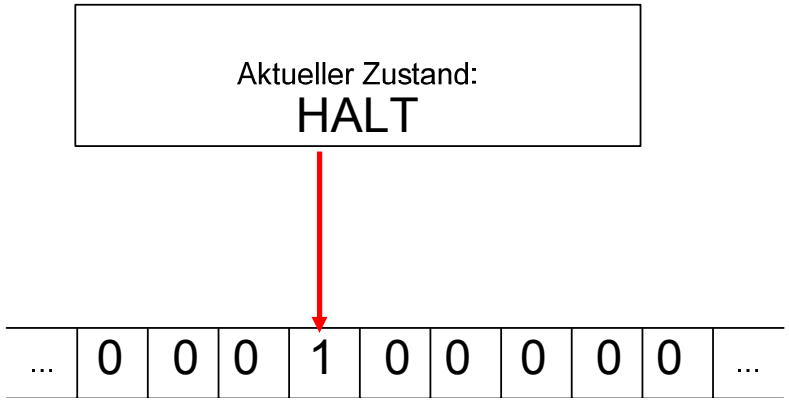
- Gegeben:
  - deterministische Turingmaschine
  - Arbeitsalphabet Γ={1,0} mit B=0
  - zu Beginn ist das Band leer, d.h. es enthält nur 0
  - die Zustandsübergangsfunktionen bewegen den Kopf immer nach links oder rechts, ein Halten ist nicht erlaubt.
  - Es gibt zusätzlich genau einen Haltezustand des Automaten der Turingmaschine, der nicht zu den Zuständen zählt.
- Busy Beaver Funktionen:
  - $\sum(n)$ (nicht zu verwechseln mit dem Bandalphabet): maximale, aber endliche Anzahl von Einsen, die eine Maschine mit *n*-Zuständen auf das Band schreiben kann und dann **hält**.
  - $S(n)$: maximale, aber endliche Anzahl der Schritte, die eine Maschine mit *n*-Zuständen ausführen kann, bevor sie **hält**.

Busy Beaver für verschiedene *n*

- n=1

	q_0
0	1-HALT
1	-

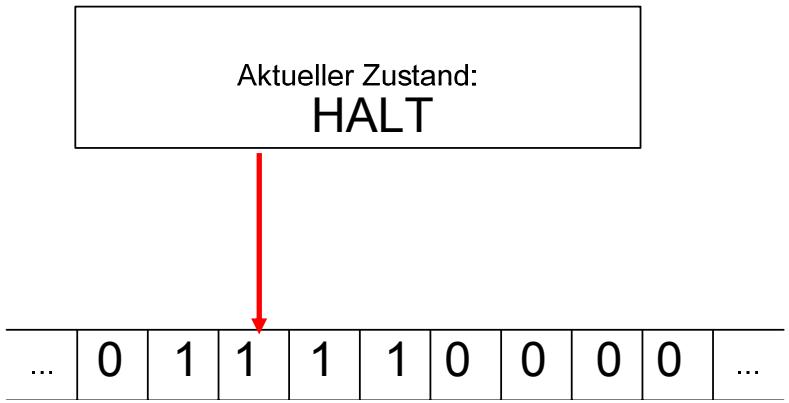
$\Sigma(1)=1$, $S(1)=2$



- n=2

	q_0	q_1
0	1-q_1-R	1-q_0-L
1	1-q_1-L	1-HALT

$\Sigma(2)=4$, $S(n)=6$



Busy Beaver

- Man kann sogar zeigen, dass die Busy Beaver Funktionen $\sum(n)$ und $S(n)$ nicht berechenbare Funktionen sind, da für eine Maschine mit n Zuständen nicht einmal entschieden werden kann, ob die Maschine hält (Halteproblem der Informatik).
- Die Busy Beaver-Funktionen wachsen sehr stark (sogar stärker als jede andere, berechenbare Funktion) und nur für n<5 ist die Lösung des Problems überhaupt bekannt.
- Die Geschichte des Problems und die aktuell besten Algorithmen (letzter Algorithmus wurde im Dezember 2005 veröffentlicht) können unter http://www.logique.jussieu.fr/~michel/ha.html eingesehen werden.

n	1	2	3	4	5	6
$\sum(n)$	1	4	6	13	$\geq$ 4.098	$\geq 1{,}2*10^{865}$
$S(n)$	2	6	21	107	$\geq$ 47.176.870	$\geq 3{,}0*10^{1730}$