Stand: 8.2.2007

# Einführung in die Informatik III

(für Studierende des 3. Fachsemesters)

Pflichtvorlesung für die Diplomstudiengänge "Informatik" und "Automatisierungstechnik in der Produktion" sowie wählbare Vorlesung für weitere Studiengänge, z. B. für Mathematik, Softwaretechnik, Lehramt Informatik usw.

Universität Stuttgart, Wintersemester 2006/2007

Dozent: Volker Claus



**Bitte beachten Sie:** Für die Richtigkeit der Inhalte und insbesondere der Programme wird keine Garantie übernommen.

## Gliederung





**Einordnung und Inhalte**

In den Vorlesungen "Einführung in die Informatik I und II" wurde der Rohstoff „Information“ und seine Darstellung, Verarbeitung, Speicherung und Übertragung vorgestellt. Im Vordergrund stand eine „sequenzielle imperative Denkweise“, d. h., Variablen werden als Behälter mit erlaubten Werten, Operatoren und einem Verfallsdatum aufgefasst, auf denen Algorithmen schrittweise, einzeln und nacheinander arbeiten, d. h., es gibt nur einen Prozessor. Datentypen können durch gängige Operationen zu sehr komplexen Gebilden zusammengesetzt werden, Verweise auf zugehörige Behälter, aber auch auf andere Programmeinheiten sind zulässig, die Zusammenfassung zu einem Modul (Paket), bestehend aus Spezifikation und Implementation, und deren Archivierung und Wiederverwendung erleichtert die Erstellung neuer Programme. Hinzu kamen Komplexitäts- und Verifikationsfragen.

In der hieran anschließenden Veranstaltung „Einführung in die Informatik III“ geht es um die Vermittlung andersartiger Informatik-Denkweisen, konkret um

- Nebenläufigkeit,
- systemnahes Programmieren,
- funktionales Programmieren und
- objektorientiertes Programmieren.

Hierzu werden neue Sprachen und Kalküle eingeübt und zwar

- gemeinsame Variable bzw. Kommunikationskanäle,
- abstrakter Assembler,
- Scheme,
- Java.



Leider erfolgt kein Ausflug in das prädikative Programmieren und weitere Paradigmen der Informatik. Es werden jedoch konkrete Grundlagen der Informatik vermittelt, die in späteren Grundlagenveranstaltungen (Betriebssysteme, Übersetzer, Software Engineering usw.) aufgegriffen werden.

**Rahmen der gesamten Veranstaltung**

Die Veranstaltung dauert 16 Wochen (in der Zeit vom 17.10.06 bis 15.2.07). Sie besteht aus (eine „Vorlesungs- oder Übungsstunde“ = 45 Zeit-Minuten):
Vorlesung: 3 Vorlesungsstunden pro Woche (insgesamt 48 Vorlesungsstunden),
Übungen: 2 Übungsstunden pro Woche (insgesamt 30 Übungsstunden).
In den Übungen sind Programme in den Sprachen Ada, Scheme und Java anzufertigen.
Die Zusatzveranstaltungen und begleitende Programmierübungen, die Sie aus dem ersten Studienjahr gewohnt waren, werden nicht mehr fortgesetzt. Wir erwarten, dass Sie evtl. fehlendes Wissen und fehlende Fertigkeiten selbst ausgleichen werden.

*Zeit und Ort im Wintersemester 2006/2007:*
Vorlesung: Dienstag und Donnerstag 14:00 bis 15:30 Uhr, Hörsaal 38.01, manchmal 14-tägige Termine, Details: siehe in der Vorlesung verteilte Informationen.
Dozent: Prof. Dr. Volker Claus.
Übungen und Programmierübungen: Je nach Übungsgruppe. Wöchentlich (15 Termine).
Es gibt zwei zur Übung zählende Zwischenklausuren (im Dezember und Ende Januar).
Betreuung (bis 10.1.07): Dipl.-Inf. Dietmar Lippold
Betreuung (ab 11.1.07): Dipl.-Inf. Botond Draskoczy.
Klausur: Mittwoch, 7.3.07, 10 Uhr im großen Hörsaal 53.01.

**Anforderungen an Sie (beachten Sie: *Anwesenheit reicht nicht!*)**
Regelmäßige Mitarbeit, insbesondere Einüben und Festigen des Stoffs der Vorlesung durch Nacharbeit und Übungen. Studentischer Aufwand für die Lehrveranstaltung wöchentlich rund 11 Zeitstunden (5 SWS, ca. 7,5 ECTS-Punkte). Wird diese Zeit regelmäßig (!) in der Vorlesungszeit aufgewendet, so reichen weitere 50 Stunden zur Prüfungsvorbereitung aus. Gesamter Zeitaufwand (einschl. der Vorlesungsstunden und Übungsgruppen): rund 225 Zeitstunden.
Die Teilnehmer(innen) sind aufgefordert, den Kontakt zu den Betreuern und zu den Tutor(inn)en zu suchen. Hierfür können z.B. E-Mails und die Sprechstunden und die direkte Ansprache in und nach den Lehrveranstaltungen genutzt werden.
Wir bemühen uns darum, alle Hörer(innen) auf eine erfolgreiche Prüfung vorzubereiten. Defizite zeigen sich meist schon frühzeitig, vor allem beim Abschneiden in den Übungen und in den Zwischenklausuren. Leider haben wir nicht genügend Personal, um auf jede(n) einzelne(n) einzugehen. Lassen Sie Ihr Studium nicht schleifen, sondern überwachen Sie sich selbst und gehen Sie **sofort** in die Sprechstunden der Mitarbeiter oder des Dozenten, sobald Ihre Leistungen nicht mehr über den Minimalanforderungen liegen, d. h., sobald Sie erkennen, dass Sie weniger als 50 % der Übungsaufgaben alleine lösen können. *Die größte Gefahr, im Informatikstudium zu scheitern, liegt im schleichenden Ausklinken aus den Veranstaltungen!* Der Stoff baut stets auf dem bereits Gelernten auf und so steigern sich einzelne Versäumnisse schnell zu großen, kaum noch zu überbrückenden Wissenslücken.



## Literatur:

Manuskripte von Frau Prof. König (WS 05/06) und Herrn Prof. Lehmann (WS 04/05).
Abelson, Harold, and Gerald J, Sussman with J. Sussman, “Structure and Interpretation of Computer Programms”, MIT, 2. Auflage, 1996 (es gibt auch eine deutsche Ausgabe)
Cousineau, Guy and Michel Mauny, „The Functional Approach to Programming“, Cambridge University Press, ISBN: 0521576814 (1998)
Goos, Gerhard und Wolf Zimmermann, „Vorlesungen über Informatik Band 1: Grundlagen und funktionales Programmieren“, Springer, ISBN: 3540244050 (2005)
Goos, Gerhard und Wolf Zimmermann, „Vorlesungen über Informatik Band 2: Objektorientiertes Programmieren und Algorithmen“, Springer, ISBN: 3540244034 (2006)
Loeckx, J., Mehlhorn, K., Wilhelm, R., „Grundlagen der Programmiersprachen“, Teubner-Verlag, Stuttgart 1986
Ottmann, T., und Widmayer, P., „Algorithmen und Datenstrukturen“, Spektrum Verlag, Heidelberg, 4. Auflage 2002
Scott, Michael L., „Programming Language Pragmatics“, Morgan Kaufmann Publishers, ISBN: 0126339511 (2000)
Thiermann, Peter, „Grundlagen der funktionalen Programmierung“, Teubner Verlag, ISBN: 3519021374 (1994)
Weitere Literatur wird bei Bedarf in den Text eingestreut, insbesondere bzgl. Java.



## Konkrete Wissens-Inhalte:

In dieser Vorlesung werden unter anderem vermittelt:

Kalküle und Sprachen:
- Petrinetze (Stellen-Transitions-Netze)
- Register- und Stackmaschinen
- Attributierte Grammatiken
- Umgang mit der Programmiersprache Scheme
- Umgang mit der Programmiersprache Java

Algorithmen:
- Peterson-Algorithmus (softwaremäßiger wechselseitiger Ausschluss)
- Beliebige genaue Wurzelberechnung (mit Pellscher Gleichung)
- Schneeflocken-Zeichnungen (nach Koch)
- Wiederholung: Baumsortieren und Sortieren durch Mischen
- Median in linearer Zeit
- Teilworterkennung (substring-problem)

**Warnung:** Manche Fehler sind in den Folien natürlich noch enthalten.



# 1. Nebenläufigkeit

## 1.1 Stellen-Transitions-Netze

Bisher: Sequentielle Programmierung, d.h., höchstens eine Stelle im Programm wird in jedem Augenblick bearbeitet. Jeder Ablauf wird hierbei in eine Folge nacheinander auszuführender Aktivitäten zerlegt.

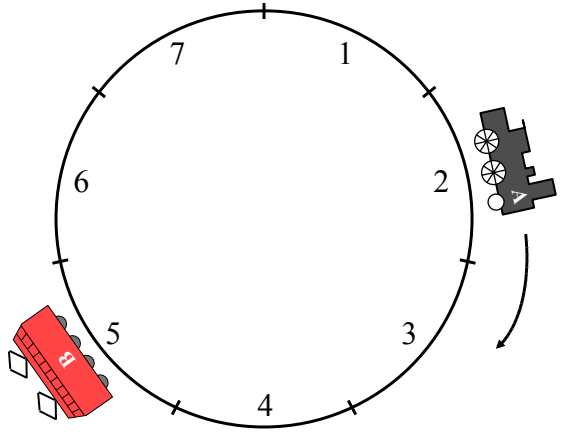
Im Folgenden wollen wir unabhängig voneinander ablaufende Programme (Prozesse, Objekte) und deren Kommunikation beschreiben. Man spricht von Nebenläufigkeit (engl.: concurrency) und von nebenläufigen, von verteilten und von parallelen Systemen. Wir beginnen mit einem Kalkül.

*Ansatz:* Verallgemeinere endliche Automaten, in denen mehrere Zustände gleichzeitig oder nacheinander aktiv sind. Als Beispiel wählen wir Züge auf einer Kreisstrecke.



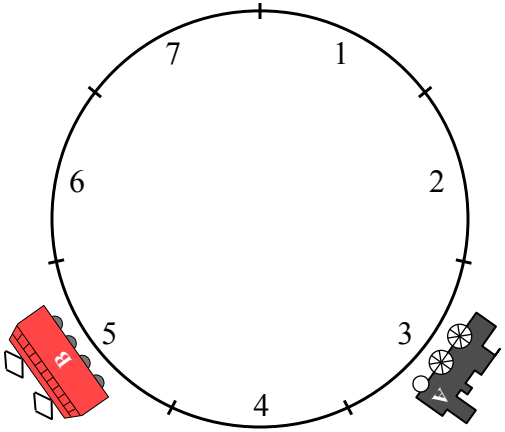
1.1.1 Beispiel: Züge auf einer kreisförmigen Strecke.

Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.



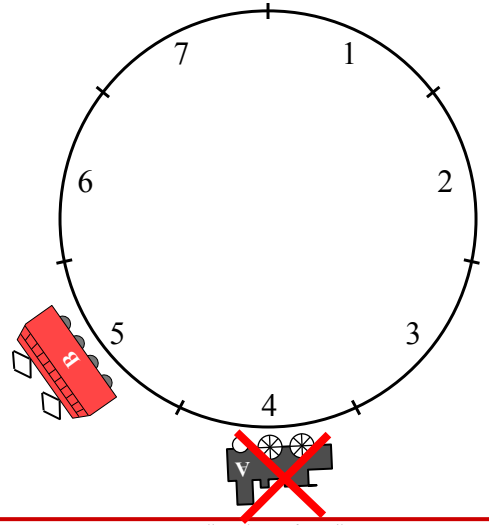



Damit die Züge nicht zusammenstoßen, verlangen wir, dass zwischen ihnen mindestens ein Streckenabschnitt frei bleibt.

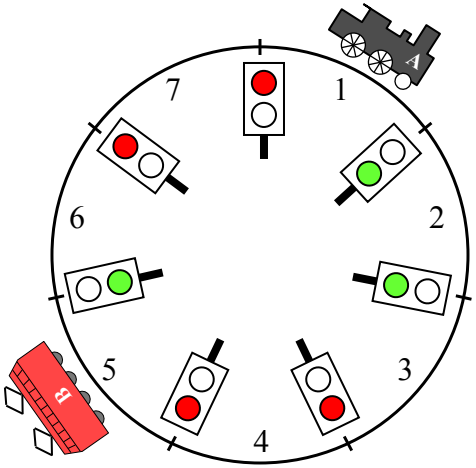





Diese Situation darf also nicht eintreten. Wie kann man dies sicherstellen?



Steuerung durch Ampeln: Grünes Signal bedeutet, dass in den Streckenabschnitt hineingefahren werden darf.

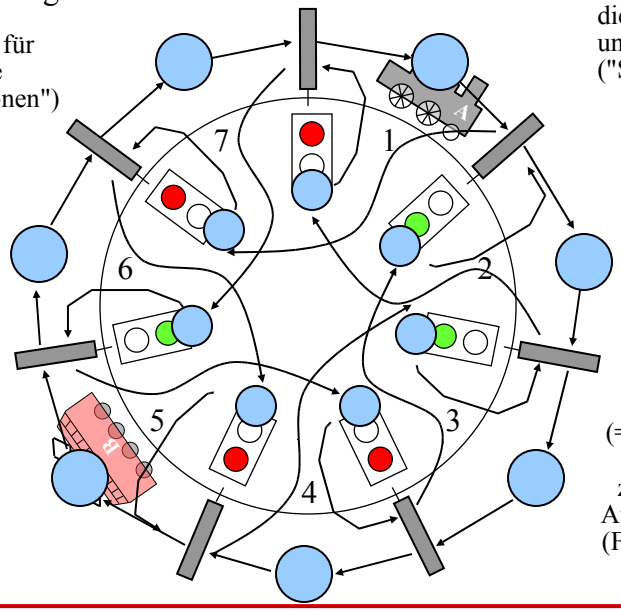





Umwandlung in ein Netz:

Rechtecke für Übergänge ("Transitionen")

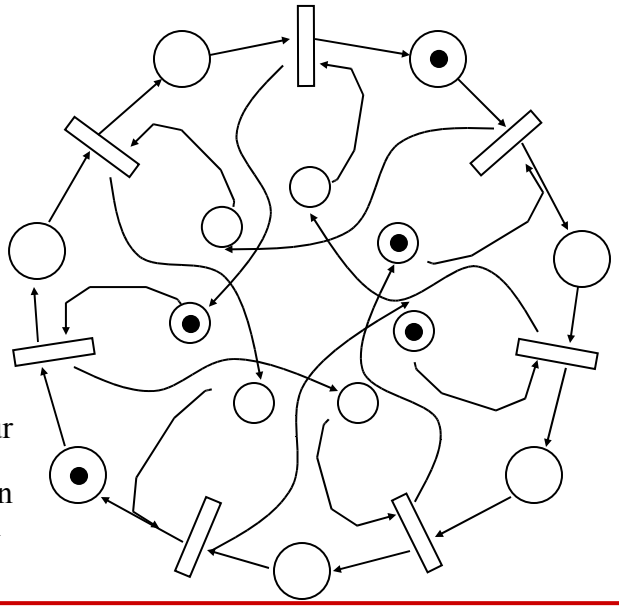
Kreise für die Strecken und Ampeln ("Stellen")




Pfeile (=Kanten) für Vorausset-zungen und Auswirkungen (Flussrelation)



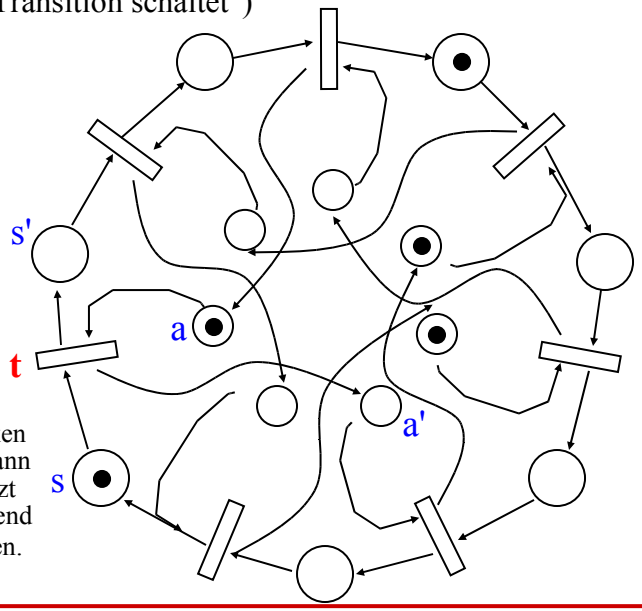
Das resultierende Netz:



Marken für mögliche Aktivitäten einfügen



Eine Aktivität durchführen ("eine Transition schaltet")

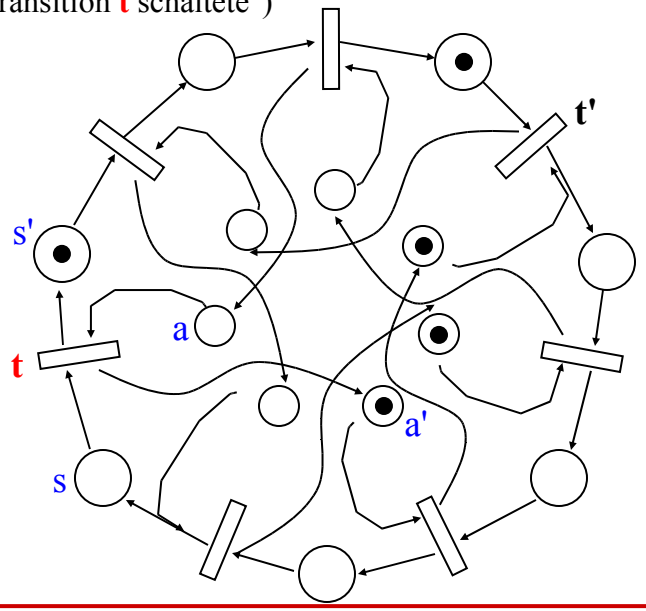



Die Marken werden dann umgesetzt entsprechend den Kanten.

*Die Transition **t** "schaltet":*

Ein Zug kann von Strecke s nach Strecke s' fahren (= in Stelle s liegt eine Marke). Die Ampel steht auf grün (= in der Stelle a liegt eine Marke). Der Zug fährt nun von s nach s' (= die Transition t schaltet).

Eine Aktivität durchführen
("die Transition t schaltete")




*Ergebnis des Schaltens:*

Der Zug ist nun auf der Strecke s' (= in Stelle s' liegt eine Marke). Die Ampel a steht auf rot (= in der Stelle a liegt keine Marke). Dafür wird aber a' auf grün gestellt (= in Stelle a' liegt eine Marke).



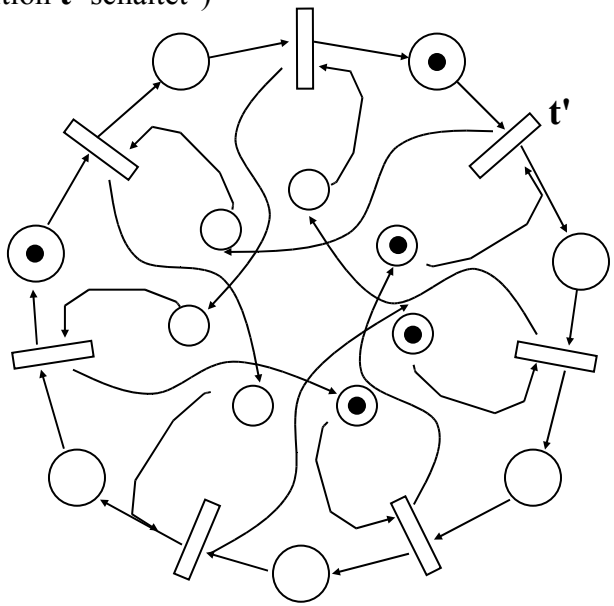
*Eine Transition t schaltet* bedeutet also:
Voraussetzung: Auf allen Stellen, von denen eine Kante nach t führt, muss mindestens eine Marke liegen.
Aktion: Von jeder dieser Stellen wird eine Marke abgezogen. Danach wird zu jeder Stelle, zu der eine Kante von t führt, eine Marke hinzugefügt.

Man nennt dies die "Schaltregel". Wir werden sie im Folgenden exakt definieren.

In unserem Beispiel: Der linke Zug kann nicht weiterfahren, weil die zugehörige Ampel keine Marke enthält. Aber der rechte Zug kann weiterfahren, d.h., die Transition t' kann jetzt schalten. Das Ergebnis (= die neue Verteilung der Marken) finden Sie auf der nächsten Folie.

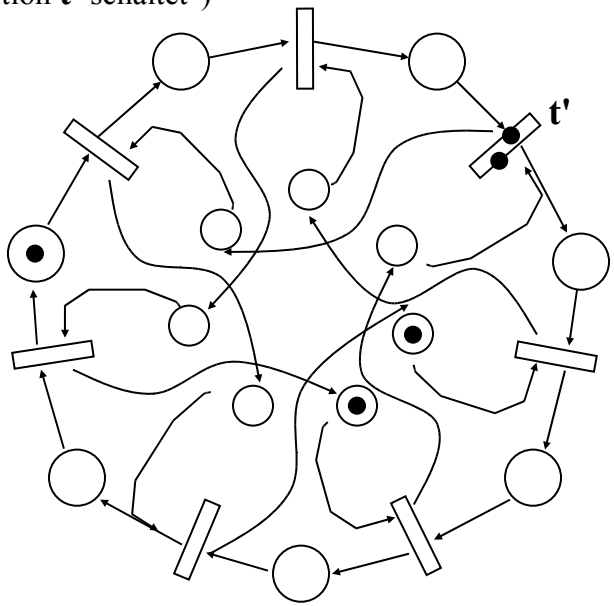


Nächste Aktivität durchführen
("die Transition **t'** schaltet")

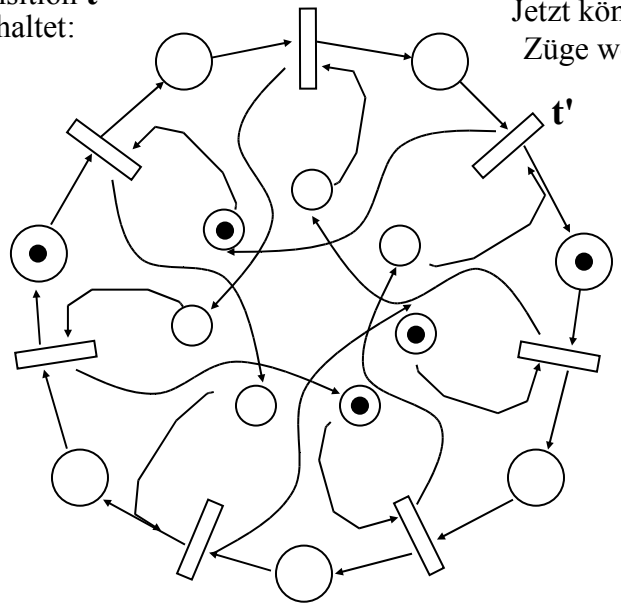





Nächste Aktivität durchführen
("die Transition **t'** schaltet")



Die Transition **t'** hat geschaltet:




Jetzt könnten beide Züge weiterfahren usw.



Definition 1.1.2: Stellen-Transitionsnetz (S/T-Netz)

$N = (S, T, F, K, W, M_0)$ heißt Stellen-Transitions-Netz $\Leftrightarrow$

(1) S ist eine endliche Menge (Menge der "Stellen"),

(2) T ist eine endliche Menge (Menge der "Transitionen"),

(3) $F \subseteq (S \times T) \cup (T \times S)$ ist die "Flussrelation" (Kantenmenge),

(4) $K: S \to \mathbb{N} \cup \{\infty\}$ ist die Kapazität für jede Stelle,

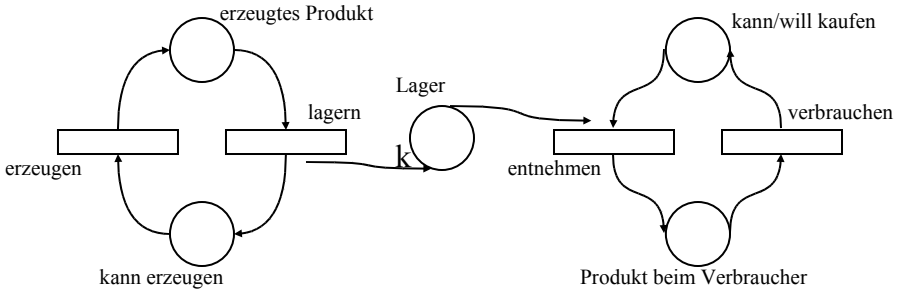
(5) $W: F \to \mathbb{N}$ ist die Gewichtsfunktion ("weight") der Kanten,

(6) $M_0: S \to \mathbb{N}_0 \cup \{\infty\}$ ist die Anfangsmarkierung, für die gelten muss: $\forall s \in S: M_0(s) \le K(s)$, d.h., in keiner Stelle dürfen mehr Marken liegen, als die Kapazität zulässt (die Markierungen schreibt man in der Regel als Vektoren).



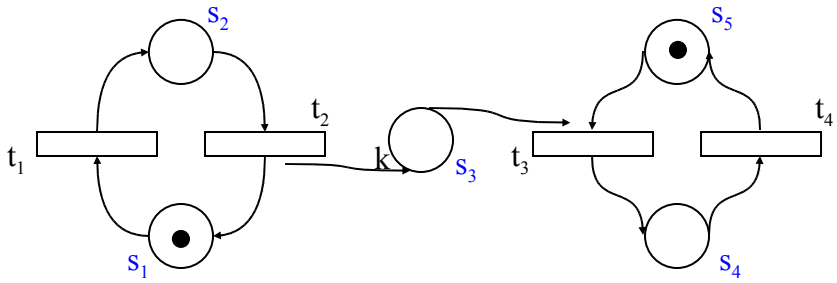
Beispiel 1.1.3: Erzeuger-Verbraucher-Kreislauf (Producer-Consumer-Cycle)

Ein Erzeuger erzeugt ein Produkt, legt dieses in einem Lager, das maximal $k \ge 1$ Stellplätze besitzt, ab und wiederholt diesen Prozess.
Ein Verbraucher entnimmt ein Produkt aus dem Lager, konsumiert dieses und wiederholt diesen Prozess.



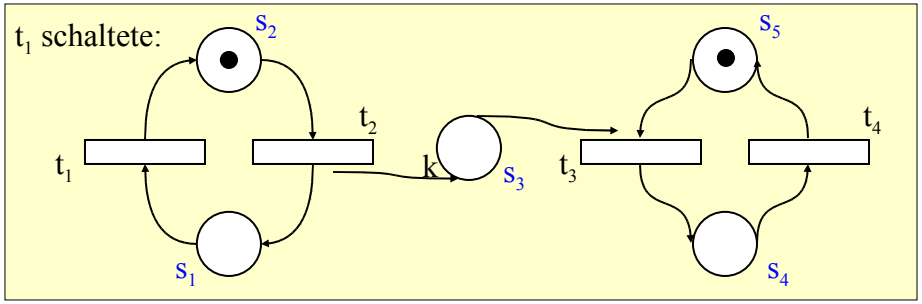

Hinweis: Das Lager hat die Kapazität k, alle anderen Stellen haben die Kapazität ∞.

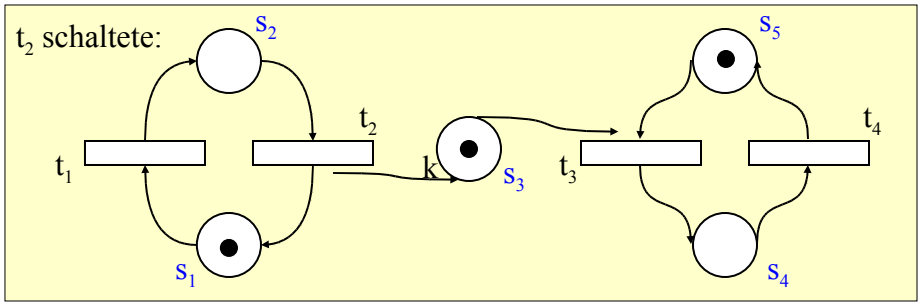





Verbraucher-Erzeuger-System
mit Anfangsmarkierung $M_0 = (1,0,0,0,1)$.

*Formal:* $S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $M_0 = (1,0,0,0,1)$,
$F = \{(s_1,t_1),(s_2,t_2),(t_1,s_2),(t_2,s_1),(t_2,s_3),(s_3,t_3),(s_5,t_3),(t_3,s_4),(s_4,t_4),(t_4,s_5)\}$,
$W((x,y))=1$ für alle Kanten (x,y), $K(s_1)=K(s_2)=K(s_4)=K(s_5)=\infty$, $K(s_3)=k$.
In dieser Anfangssituation kann nur die Transition $t_1$ schalten.
Aus der Anfangsmarkierung (1,0,0,0,1) ensteht dann die Folge-Markierung (0,1,0,0,1). Nun kann $t_2$ schalten und es entsteht die Markierung (1,0,1,0,1). Jetzt können $t_1$ oder $t_3$ schalten, wobei die Markierungen (0,1,1,0,1) bzw. (1,0,0,1,0) entstehen usw.

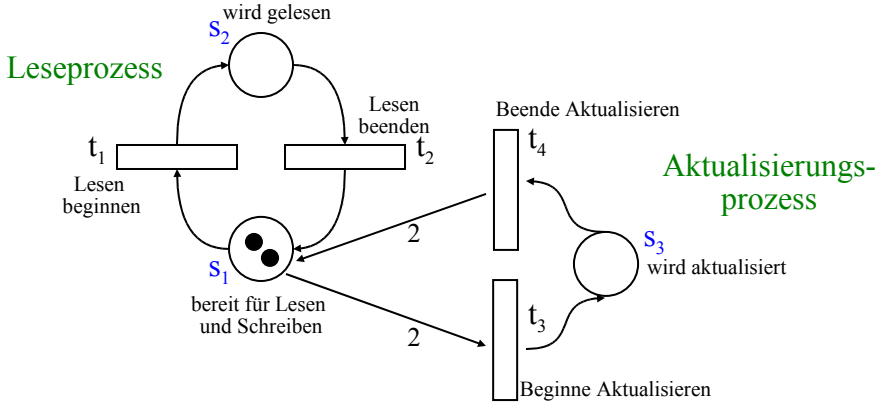


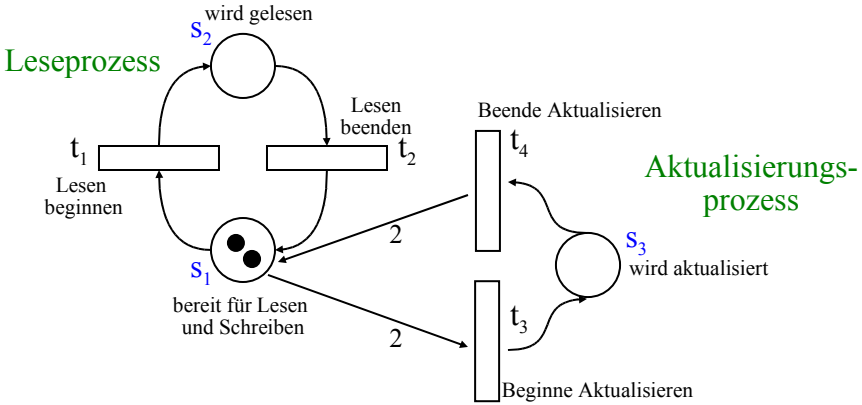



Beispiel 1.1.4: Lese-Schreib-Konflikt

Eine Datenbank steht zwei Benutzern zum Lesen zur Verfügung. Ab und zu sollen die Inhalte von einem Autor aktualisiert werden; zu diesem Zeitpunkt darf nur der Autor auf die Datenbank zugreifen können. Modell hierzu?





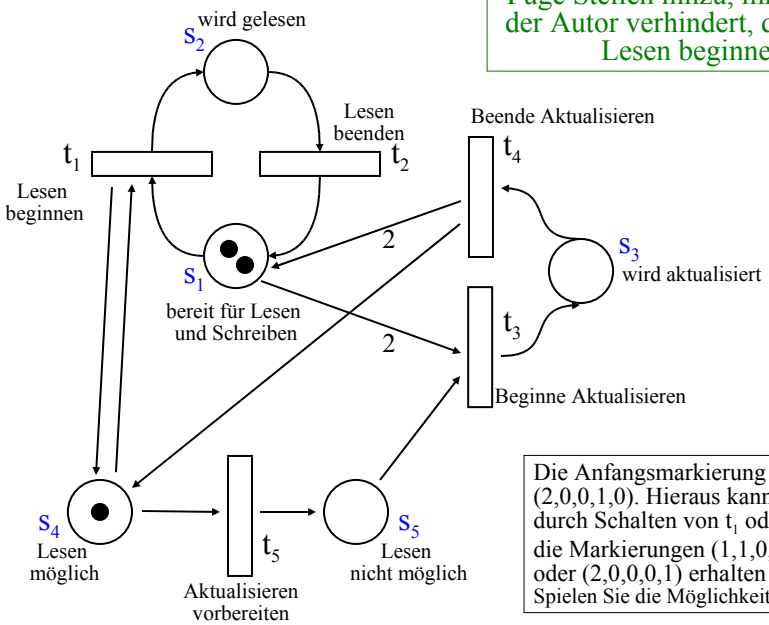

Nun kann eine "unfaire Aktionsfolge" auftreten:

$t_1\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2\, t_1\, t_2 \ldots$

Hierbei ist stets mindestens eine Marke in der Stelle $s_2$, so dass niemals aktualisiert werden kann. Wie kann man erzwingen, dass der Autor das Lesen unterbrechen kann?



Füge Stellen hinzu, mit denen der Autor verhindert, dass das Lesen beginnen kann.




Die Anfangsmarkierung lautet: (2,0,0,1,0). Hieraus kann man durch Schalten von $t_1$ oder von $t_5$ die Markierungen (1,1,0,1,0) oder (2,0,0,0,1) erhalten usw. Spielen Sie die Möglichkeiten durch!

Fragen 1.1.5 a:

1. Erreichbarkeitsproblem: Wie kann man feststellen, ob eine angestrebte Situation (= Markierung) von einer gegebenen Situation (= Markierung) aus erreicht werden kann?
2. Beschränktheit: Wie kann man nachweisen, dass die Zahl der Marken in allen Stellen beschränkt bleibt?
3. Fairness: Wie kann man ermitteln und sicherstellen, dass keine "unfairen Aktionsfolgen" (= unfaire Folge von schaltenden Transitionen) auftreten kann?
4. Wir kann man beweisen, dass das Netz nicht in "Verklemmungen" gerät, also in eine Markierung, von der aus es nicht mehr weitergeht?
5. Wie stellt man sicher, dass das Netz nicht in "sinnlose Schleifen" (= Iteration von schaltenden Transitionen, die aus Sicht des Problems keinen Sinn machen) gerät?

Um diese Fragen beantworten zu können, müssen wir zuerst die Arbeitsweise und die erforderlichen Begriffe exakt definieren.



Definition 1.1.5 b: Begriffe und Schreibweisen

$N = (S, T, F, K, W, M_0)$ sei ein Stellen-Transitions-Netz.

(1) Jede Abbildung $M: S \to \mathbb{N}_0 \cup \{\infty\}$ heißt Markierung von N. M heißt zulässig, wenn für alle $s \in S$ gilt: $M(s) \le K(s)$.

(2) Eine Markierung schreibt man in der Regel als Spaltenvektor (oder in Texten auch als Zeilenvektor). Hierbei wird vorausgesetzt, dass die Menge der Stellen S geordnet ist: $S = \{s_1, s_2, s_3, \ldots, s_n\}$ mit $s_1 < s_2 < s_3 < \ldots < s_n\}$

(3) Für $x \in S \cup T$ heißen ${}^\bullet x = \{(y,x) \mid (y,x) \in F\}$ der Vorbereich von x und $x^\bullet = \{(x,y) \mid (x,y) \in F\}$ der Nachbereich von x.

(4) Die Flussrelation $F \subseteq (S \times T) \cup (T \times S)$ zusammen mit der Gewichtsfunktion W wird meist auf die gesamte Menge $(S \times T) \cup (T \times S)$ wie folgt fortgesetzt zu $W': S \to \mathbb{N}_0$
$W'((x,y)) :=$ if $(x,y) \in F$ then $W((x,y))$ else 0 fi.
(W' beschreibt F und W eindeutig.)



Definition 1.1.6: Arbeitsweise von S/T-Netzen

$N = (S, T, F, K, W, M_0)$ sei ein Stellen-Transitions-Netz.

(1) Eine Transition $t \in T$ heißt unter der Markierung M aktiviert $\Leftrightarrow \forall s \in {}^\bullet t: W((s,t)) \le M(s)$ und $\forall s \in t^\bullet: W((t,s)) + M(s) \le K(s)$. Man schreibt hierfür auch: $M[t>$

(2) *Schaltregel:* Es sei t eine Transition, die unter der zulässigen Markierung M aktiviert ist. Dann kann t schalten und es entsteht aus M die Folge-Markierung M' mit:

$$M'(s) = \begin{cases} M(s) & s \notin {}^\bullet t \text{ und } s \notin t^\bullet, \\ M(s) - W((s,t)) & s \in {}^\bullet t \text{ und } s \notin t^\bullet, \\ M(s) + W((t,s)) & s \notin {}^\bullet t \text{ und } s \in t^\bullet, \\ M(s) - W((s,t)) + W((t,s)) & s \in {}^\bullet t \text{ und } s \in t^\bullet. \end{cases}$$

[Wir hätten auch $M'(s) = M(s) - W'((s,t)) + W'((t,s)), \forall s \in S$ schreiben können, siehe Definition 1.1.5 (4).]



noch Definition 1.1.6:

(3) Schreibweise: Wenn t unter M aktiviert ist und nach dem Schalten von t die Markierung M' entsteht, so schreibt man $M[t>M'$ oder $M \xrightarrow{t} M'$.

(4) Fortsetzung der Relation $[>$ auf Folgen von Transitionen (also auf $T^*$; Wörter über T nennen wir auch Schaltfolgen):
$M[t_1 t_2 t_3 \ldots t_r> \Leftrightarrow$
es gibt Markierungen $M_1, M_2, \ldots, M_{r-1}$ mit $M[t_1>M_1$, $M_1[t_2>M_2$, $M_2[t_3>M_3$, ..., $M_{r-2}[t_{r-1}>M_{r-1}$, $M_{r-1}[t_r>$.

noch Definition 1.1.6: Aktiviertheit und Erreichbarkeit

(5) Fortsetzung der Relation $[>$ auf Schaltfolgen (also auf Folgen von Transitionen):
$M[t_1 t_2 t_3 \ldots t_r > M' \Leftrightarrow$ es gibt Markierungen $M_1, M_2, \ldots, M_{r-1}$ mit $M[t_1 > M_1$, $M_1[t_2 > M_2$, $M_2[t_3 > M_3$, ..., $M_{r-1}[t_r > M'$.
(Im Falle r=0 muss M=M' sein.)

(6) $ERR(M) = \{M' \mid$ es existiert $t_1 t_2 t_3 \ldots t_r$ mit $M[t_1 t_2 t_3 \ldots t_r > M'\}$ heißt Erreichbarkeitsmenge bzgl. M.
(Beachte: $M \in ERR(M)$, insbesondere ist ERR(M) nie leer.)
Wenn M' in ERR(M) liegt, so sagt man auch, M' ist von M aus erreichbar.
$ERR(N) = \{M' \mid$ es existiert $t_1 t_2 t_3 \ldots t_r$ mit $M_0[t_1 t_2 t_3 \ldots t_r > M'\}$ heißt Erreichbarkeitsmenge des Netzes N.



Die Erreichbarkeit stellt die Bedeutung der S/T-Netze dar. Kennt man also den Erreichbarkeitsgraphen (siehe unten) eines S/T-Netzes im Detail, so kann man hieraus alle seine Eigenschaften ableiten. Wir werden dies an den Begriffen Beschränktheit, Lebendigkeit und Fairness demonstrieren.

*Hinweise:*
Oft schreibt man S/T-Netze nur in der Form $N = (S, T, F, M_0)$.
In diesem Fall ist $K(s) = \infty$ für alle Stellen s und $W((x,y)) = 1$ für alle Kanten (x,y) einzusetzen.
Dies gilt auch für Zeichnungen: Fehlen die Angaben für K oder W an einer Stelle bzw. Kante, so ist unbeschränkte Kapazität bzw. Kantengewicht 1 gemeint.
In diesem Abschnitt sprechen wir oftmals nur von einem "Netz" und meinen damit stets ein S/T-Netz.



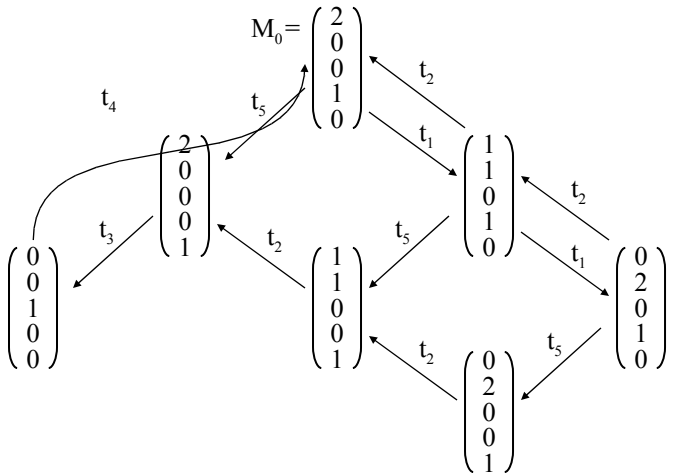
Um die Arbeitsweise eines Netzes im Ganzen zu verstehen, konstruiert man schrittweise alle Markierungen, die man von der Anfangsmarkierung $M_0$ aus erreichen kann. Das Ergebnis ist der "Erreichbarkeitsgraph" des Netzes.

Dieser Graph kann unendlich oder endlich sein. Unter den endlichen Graphen interessieren vor allem diejenigen, die nicht allzu groß werden; also: Wenn d die Länge der Darstellung eines Netzes ist, dann soll die Länge der Darstellung des Erreichbarkeitsgraphen höchstens polynomiell bzgl. d sein.

Leider ist dies jedoch nur selten der Fall. Wir betrachten hierzu einige Beispiele und definieren den Erreichsbarkeitsgraphen eines Netzes formal.



1.1.7: *Konstruktion des Erreichbarkeitsgraphen* des letzten S/T-Netzes aus Beispiel 1.1.4: Ausgehend von $M_0$ werden nacheinander alle im nächsten Schritt erreichbaren Markierungen notiert:

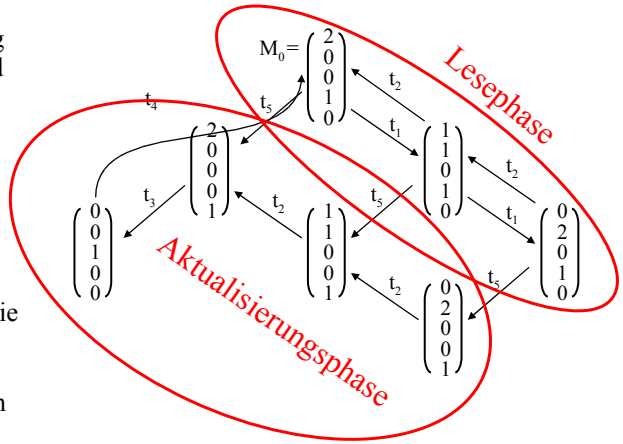


Dieser Graph besitzt einige Eigenschaften, die uns Hinweise geben, ob das angegebene Netz unser gestelltes Lese-Schreib-Problem tatsächlich löst:

1. Man erkennt die (korrekte) Trennung zwischen Lesen und Schreiben

2. Man entdeckt eine "Invariante" der erreichbaren Markierungen M: $(1,1,3,1,1)\cdot M = 3$.

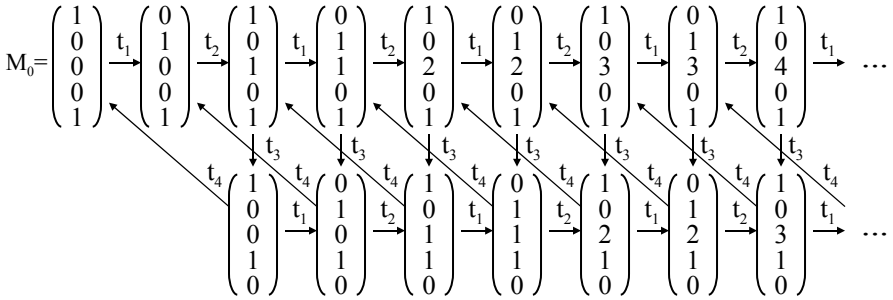
3. Man sieht, dass die Transitionenfolgen $(t_1t_2)^*$ und $(t_5t_3t_4)^*$ korrekt von $M_0$ nach $M_0$ führen.




4. Man erkennt, dass jede Transition irgendwann noch einmal schalten kann, dass also keine Transition irgendwann überflüssig wird.



## 1.1.8: Konstruktion des Erreichbarkeitsgraphen aus 1.1.3



Falls die in 1.1.3 angegebene Größe k eine natürliche Zahl ist, so besitzt der Erreichbarkeitsgraph genau 4k+2 Markierungen. Ist $k = \infty$, so ist der Erreichbarkeitsgraph unendlich groß.



**Definition 1.1.9:**

Es sei $N = (S, T, F, K, W, M_0)$ ein Stellen-Transitions-Netz mit der Erreichbarkeitsmenge ERR(N).

Der Erreichbarkeitsgraph G(N) des Netzes N ist ein gerichteter Graph mit der Knotenmenge ERR(N). Er besitzt in der Regel Mehrfachkanten, die dann aber mit *verschiedenen* Transitionen beschriftet sind. Eine Kante (M, M') mit der Beschriftung t existiert genau dann, wenn $M[t>M'$ gilt.

*Formal:* $G(N) = (\mathrm{ERR}(N), \{(M, t, M') \mid M[t>M'\})$, wobei (M, t, M') eine Kante von der Markierung M zur Markierung M' mit Beschriftung t ist.

(M, t, M') zeichnet man in der Form $M \xrightarrow{t} M'$

[Hinweis: Zu jedem Netz gibt es genau einen Erreichbarkeitsgraphen. Es ist klar, wie man ihn schrittweise aufbaut.]



## 1.1.10: Beschränktheit eines Netzes

Ein Netz soll beschränkt heißen, wenn es eine natürliche Zahl k gibt, so dass keine Markierung im Netz erreicht werden, in der eine Stelle mehr als k Marken besitzt.

**Definition:** Sei $N = (S, T, F, K, W, M_0)$ ein S/T-Netz.

(1) Es sei k eine natürliche Zahl. Eine Stelle s des Netzes N heißt k-beschränkt, wenn für jede von $M_0$ aus erreichbare Markierung M gilt, M(s) ist nicht größer als k, d.h., $\forall M \in \mathrm{ERR}(N)$: $M(s) \le k$.

(2) N heißt k-beschränkt, wenn jede Stelle k-beschränkt ist, d.h., $\forall s \in S\ \forall M \in \mathrm{ERR}(N)$: $M(s) \le k$.

(3) s heißt beschränkt, wenn es eine natürliche Zahl k gibt, so dass s k-beschränkt ist, d.h., $\exists k \in \mathbb{N}\ \forall M \in \mathrm{ERR}(N)$: $M(s) \le k$.

(4) N heißt beschränkt, wenn jede Stelle von N beschränkt ist, d.h., $\exists k \in \mathbb{N}\ \forall s \in S\ \forall M \in \mathrm{ERR}(N)$: $M(s) \le k$.

1.1.11: Folgerung

Ein Netz ist genau dann beschränkt, wenn sein Erreichbarkeitsgraph endlich ist.

Beweis: Sei S die Menge der Stellen des Netzes N.

"⇒" Wenn N beschränkt ist, so gibt es ein k, so dass alle Markierungen des Erreichbarkeitsgraphen nur Komponenten besitzen, die kleiner oder gleich k sind. Dann kann es aber höchstens $(k+1)^{|S|}$ Markierungen in ERR(N) geben, d.h., der Erreichbarkeitsgraph ist endlich.

"⇐" Wenn der Erreichbarkeitsgraph endlich ist, dann existiert das Maximum m für alle Komponenten von Markierungen in ERR(N). Jede Stelle ist dann m-beschränkt, d.h., das Netz ist beschränkt.



1.1.12: Größe eines Netzes. Wie groß können Erreichbarkeitsgraphen werden, bezogen auf die Größe des zugehörigen Netzes?

Hierzu müssen wir zunächst festlegen, was die Größe eines Netzes ist. Wie üblich ist dies die Länge einer Darstellung.

Meist wählt man eine normierte Darstellung. Eine relativ kurze Darstellung ist z.B. die Folgende, bei der die Stellen stets mit den Zahlen von 1 bis n = |S| und die Transitionen mit den Zahlen von 1 bis m = |T| bezeichnet werden; die i-te Stelle beschreibt man, indem man vor die Nummer i ein 's' setzt; analog sei 't'j die j-te Transition:

<Zahl der Stellen>; <Zahl der Transitionen>;
<Liste der Kanten in der Form (<Bezeichnung Knoten>,
<Bezeichnung Knoten>, <Gewicht der Kante>) >;
<Kapazitäten als n-stelliger Vektor>;
<Anfangsmarkierung als n-stelliger Vektor>;;

Hierbei werden alle Zahlen binär aufgeschrieben.

Definition: Die Größe des Netzes ist Länge dieser Darstellung.



Ein Netz wird hier also als ein Wort über dem 7-elementigen Alphabet {s, t, 0, 1, ,, ;, ∞} aufgefasst.

*Beispiel:* In Beispiel 1.1.3 hatten wir folgendes Netz vorgestellt:
$S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, $M_0 = (1,0,0,0,1)$,
$F=\{(s_1,t_1),(s_2,t_2),(t_1,s_2),(t_2,s_1),(t_2,s_3),(s_3,t_3),(s_5,t_3),(t_3,s_4),(s_4,t_4),(t_4,s_5)\}$,
$W((x,y))=1$ für alle Kanten (x,y), $K(s_1)=K(s_2)=K(s_4)=K(s_5)=\infty$, $K(s_3)=k$.

Dieses Netz schreiben wir also für k=9 in folgender Form auf:

101;100;s1,t1,1,s10,t10,1,t1,s10,1,t10,s1,1,t10,s11,1,s11,t11,1,
s101,t11,1,t11,s100,1,s100,t100,1,t100,s101,1;∞,∞,1001,∞,∞;
1,0,0,0,1;;

Dieses Wort besteht aus 134 Zeichen. Unser Netz besitzt also die Größe 134.



Einschub: Übungsaufgaben:

1. Welche Größen haben das "resultierende Netz" des Zug-Beispiels 1.1.1 und das letzte Netz aus Beispiel 1.1.4 (mit 5 Stellen und 4 Transitionen)?
2. Definieren Sie auf ähnliche Weise die Größe eines Erreichbarkeitsgraphen.
3. Welche Größen haben die Erreichbarkeitsgraphen der drei obigen Netze, für die die Größe bestimmt wurde? (Für zwei Beispiele wurden in 1.1.7 und 1.1.8 die Erreichbarkeits-graphen bereits angegeben; für das Zugbeispiel müssen Sie diesen Graphen selbst konstruieren.)

4. Definieren Sie, was es bedeuten soll, dass eine Klasse von Netzen *polynomiell konstruierbare* Erreichbarkeitsgraphen besitzt. Wieviel Platz benötigt man für diese Konstruktionen höchstens?
5. Geben Sie unendlich große Klassen von Netzen an, deren Erreichbarkeitsgraphen sich genau mit linearem, bzw. mit quadratischem Zeitaufwand konstruieren lassen.
6. Es gibt Netze, deren Erreichbarkeitsgraphen exponentiell größer sind als die Netze selbst. Versuchen Sie, solche Netze selbst zu entdecken, oder durchsuchen Sie die Literatur hiernach.
7. Gibt es Klassen von Netzen, deren Erreichbarkeitsgraphen zwar endlich sind, die aber viel stärker als exponentiell wachsen?



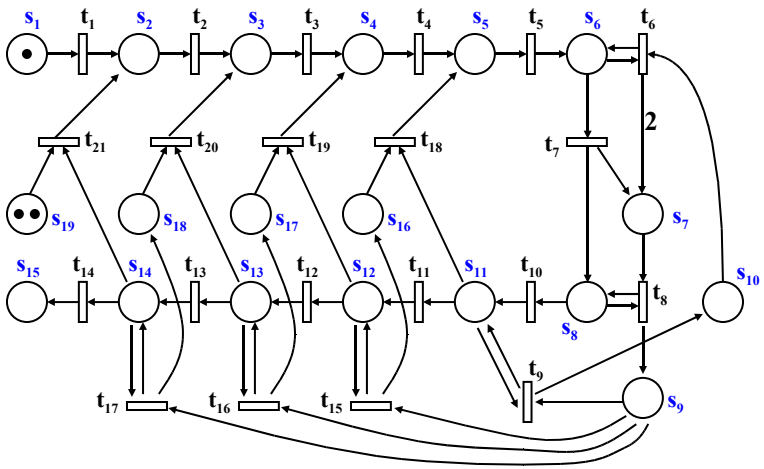
*Warnung zu Punkt 7:* Die Erreichbarkeitsgraphen von S/T-Netzen können gewaltig wachsen.

Das S/T-Netz auf der folgenden Folie mit 19 Stellen und 21 Transitionen vollzieht die Berechnung der sog. Ackermann-Funktion A (in der fünften Stufe) nach. Dies ist eine totale berechenbare Funktion $A: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, die schneller als jede Funktion wächst, die man nur mit elementaren Anweisungen, der Sequenz, der Alternative und der for-Schleife darstellen kann. Der Erreichbarkeitsgraph dieses S/T-Netzes ist endlich, besitzt aber mehr als

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad (65535 \text{ mal})$$

Knoten. Vollziehen Sie etwa 50 Schritte nach, um die Wirkungsweise des Netzes zu erahnen.








## 1.1.13: Lebendigkeit eines Netzes

Ein Netz soll lebendig heißen, wenn jede Transition irgendwann noch einmal schalten könnte. Genauer: Für jede Transition t muss gelten: Wenn man sich, ausgehend von $M_0$, in irgendeiner Markierung M befindet, dann muss von M aus eine Markierung M' erreichbar sein, unter der t aktiviert ist (also schalten kann).

Definition: Sei $N = (S, T, F, K, W, M_0)$ ein S/T-Netz.

(1) Eine Transition t des Netzes N heißt lebendig, wenn es zu jeder von $M_0$ aus erreichbaren Markierung M eine von M aus erreichbare Markierung M' mit $M'[t>$ gibt. Formal: $\forall M \in ERR(N)\ \exists M' \in ERR(M): M'[t>$.

(2) N heißt (stark) lebendig, wenn jede Transition von N lebendig ist.

Man könnte ein Netz auch lebendig nennen, wenn es immer weiterschalten kann. Im Netz darf es dann keine "Verklemmung" geben, d.h., es darf keine von $M_0$ aus erreichbare Markierung ohne Folge-Markierung geben.

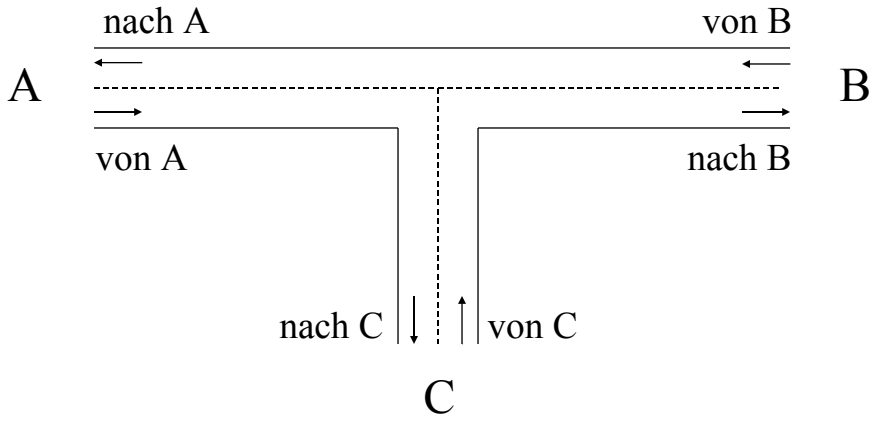
Fortsetzung der Definition:

(3) Eine Markierung M heißt Verklemmung ("Deadlock"), wenn unter M keine Transition aktiviert ist, d.h.
$\neg \exists t \in T: M[t>$.

(4) Das Netz N heißt schwach lebendig, wenn es keine von $M_0$ aus erreichbare Verklemmung besitzt, d.h.
$\forall M \in ERR(N) \exists t \in T: M[t>$.

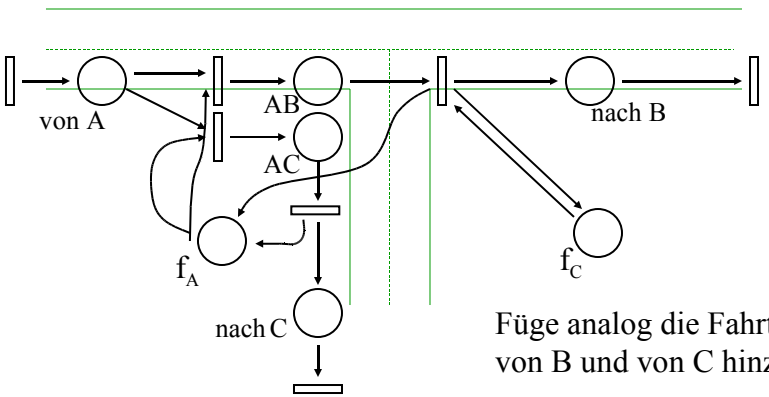


1.1.14: Beispiel:
Straßenkreuzung mit Vorfahrtsregel "rechts vor links".





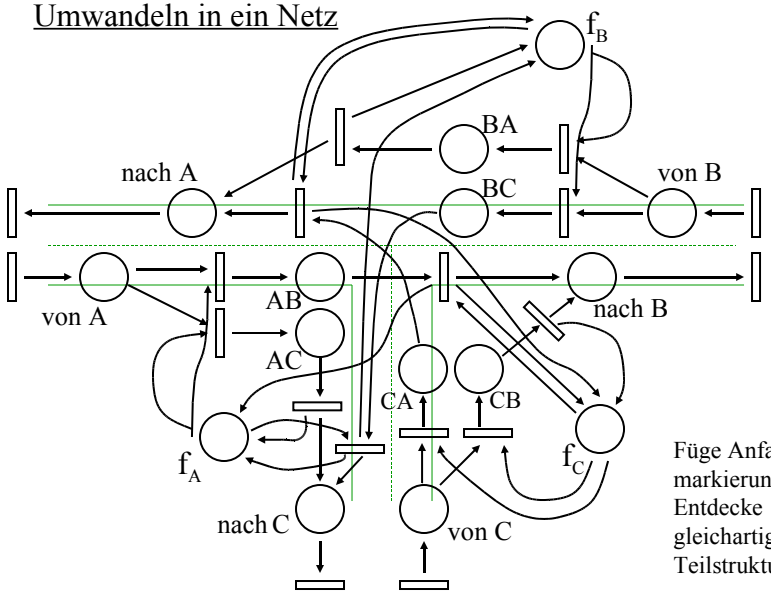

Umwandeln in ein Netz, wobei explizit "von rechts kommt niemand" durch die Stellen f modelliert wird. Wir betrachten zunächst nur die von A kommenden Fahrzeuge, für die nur wichtig ist, ob die Strecke von C zur Kreuzung frei ist ($f_C$).




Füge analog die Fahrten von B und von C hinzu.

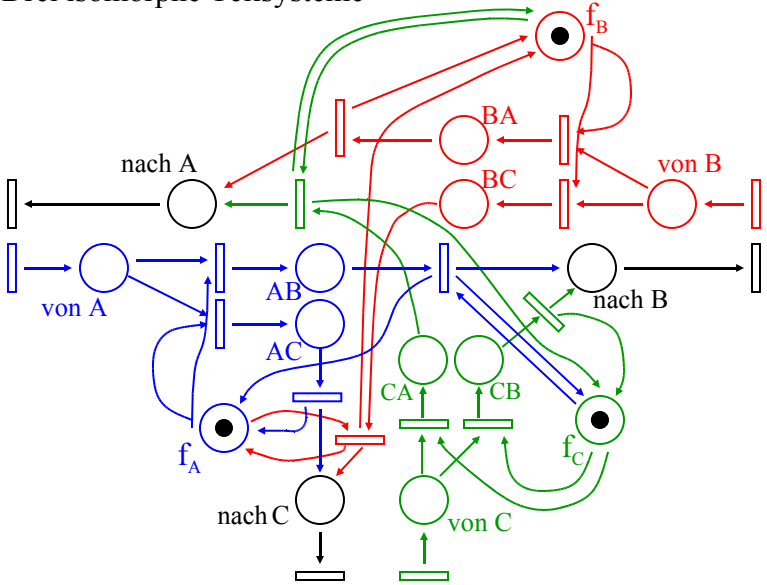


Umwandeln in ein Netz



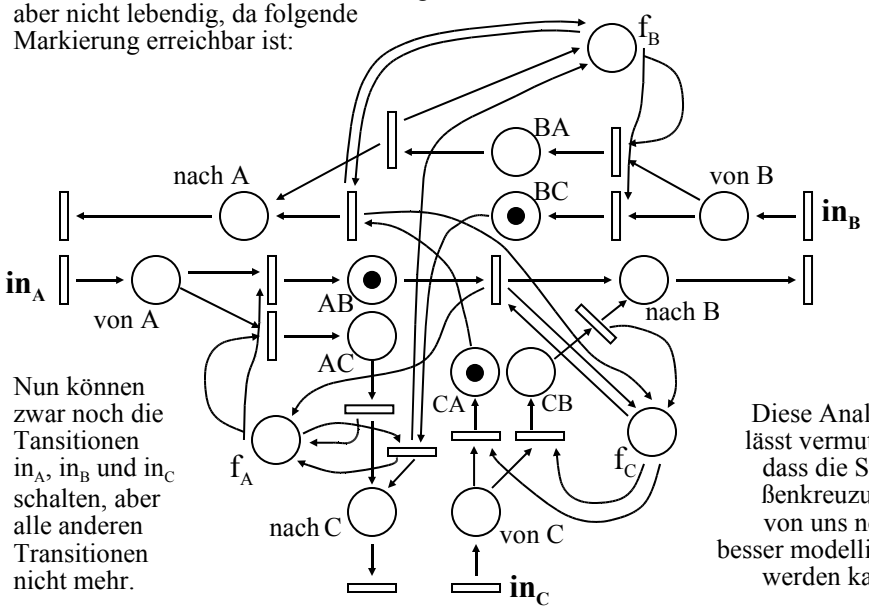

Füge Anfangsmarkierung hinzu. Entdecke gleichartige Teilstrukturen:

## Drei isomorphe Teilsysteme






Dieses Netz ist zwar schwach lebendig, aber nicht lebendig, da folgende Markierung erreichbar ist:



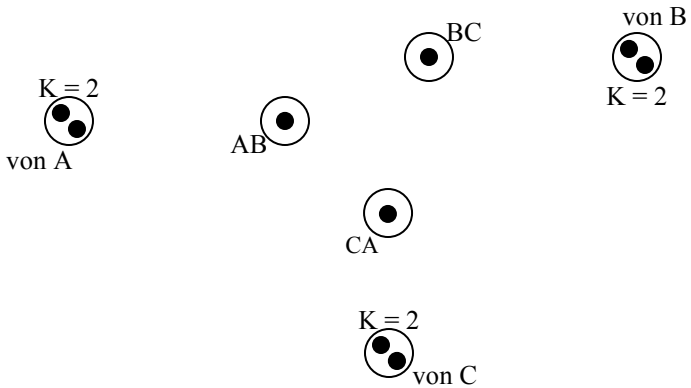

Nun können zwar noch die Tansitionen $in_A$, $in_B$ und $in_C$ schalten, aber alle anderen Transitionen nicht mehr.

Diese Analyse lässt vermuten, dass die Straßenkreuzung von uns noch besser modelliert werden kann.



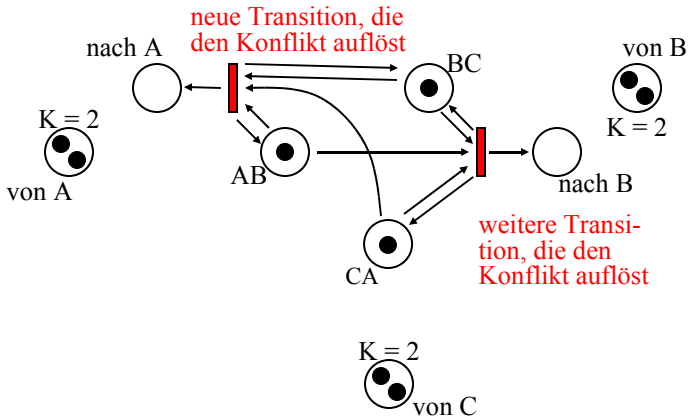
Wir sollten die Eingangs-Stellen "von A", "von B" und "von C" mit einer Kapazität, z.B. mit 2 belegen.

In diesem Fall gibt es eine Verklemmung, nämlich (wir zeigen nur den relevanten Ausschnitt aus dem Netz, also die Stellen der Verklemmung, die noch Marken tragen):






Diese Verklemmung können wir beseitigen, indem wir eine Transition einfügen, die in diesem Fall z.B. das von C nach A fahrende Fahrzeug als erstes über die Kreuzung lässt:




Wir können zusätzlich eine Transition hinzufügen, die das von A nach B fahrende Fahrzeug bevorzugt.

Als drittes können wir eine Transition hinzufügen, die das von B nach C fahrende Fahrzeug als erstes über die Kreuzung lässt.

Dies führt zu einem Netz, das relativ realistisch die Situationen an einer Kreuzung widerspiegelt. Es enthält alle Möglichkeiten, eine Verklemmung aufzulösen und unsinnige Überlastungen in gewissen Teilen des Netzes zu vermeiden.

Die Leser(innen) mögen diese Einzelheiten in das bereits vorhandene Netz eintragen und das neue Netz untersuchen.

*Ende Beispiel 1.1.14*



1.1.15: Wirkung des Schaltens einer Transition $t_j$:

$$\Delta t_j = \begin{pmatrix} W'((t_j,s_1)) - W'((s_1,t_j)) \\ W'((t_j,s_2)) - W'((s_2,t_j)) \\ W'((t_j,s_3)) - W'((s_3,t_j)) \\ \dots \\ W'((t_j,s_n)) - W'((s_n,t_j)) \end{pmatrix} \in \mathbb{N}_0^n \quad \text{mit } n = |S|$$

Wenn $t_j$ schaltet, wird die Markierung genau um diesen Vektor verändert, d.h.: Aus $M[t_j>M'$ folgt $M' = M + \Delta t_j$.

Wir übertragen diese Aussage nun auf eine Folge von Transitionen. Hierzu sei $T = \{t_1, t_2, ..., t_m\}$.



Hierzu definieren wir für $w \in T^*$:
$\#_j w$ = Anzahl der $t_j$, die in w vorkommen.

*Formal:* $\#_j \varepsilon = 0$ und $\#_j vt = \underline{\text{if}}\ t_j = t\ \underline{\text{then}}\ \#_j v + 1\ \underline{\text{else}}\ \#_j v\ \underline{\text{fi}}$ $(\forall v \in T^*, \forall t \in T)$.

Dann gilt für jede Folge w von Transitionen mit $M[w>M'$:

$M' = M + \#_1 w \cdot \Delta t_1 + \#_2 w \cdot \Delta t_2 + \#_3 w \cdot \Delta t_3 \dots + \#_m w \cdot \Delta t_m$ .

Dies formulieren wir nun in Matrizenschreibweise. Hierzu sei #w der (Spalten-) Vektor, dessen Komponenten $\#_1 w$, $\#_2 w$, ..., $\#_m w$ sind:

$$\#w = \begin{pmatrix} \#_1 w \\ \#_2 w \\ \#_3 w \\ \dots \\ \#_m w \end{pmatrix}$$



1.1.16 Definition:
Gegeben sei ein Netz $N = (S, T, F, K, W, M_0)$. Es seien $S = \{s_1, s_2, ..., s_n\}$ und $T = \{t_1, t_2, ..., t_m\}$. Die (n,m)-Matrix

$$C = (\Delta t_1, \Delta t_2, \Delta t_3, ..., \Delta t_m)$$

heißt Inzidenzmatrix des Netzes N.

1.1.17 Folgerung:

Für alle Markierungen M und M' und für alle Schaltfolgen $w \in T^*$ gilt:

Aus $M[w>M'$ folgt $M' = M + C \cdot \#w$.

1.1.18 Definition:

(1) Jeder (Zeilen-) Vektor $y \in \mathbb{Z}^n$ mit $y \cdot C = 0$ heißt S-Invariante.

(2) y heißt echte S-Invariante $\Leftrightarrow$ y ist eine S-Invariante mit nichtnegativen Komponenten und $y \neq 0$.

(3) Jeder (Spalten-) Vektor $x \in \mathbb{Z}^m$ mit $C \cdot x = 0$ heißt T-Invariante.

Beachte 1.1.17, so sieht man: Eine T-Invariante gibt an, wie oft die einzelnen Transitionen schalten müssen, damit die ursprüngliche Markierung wieder hergestellt wird.

Eine S-Invariante y besagt, dass der Wert $y \cdot M = y \cdot M'$ für alle von M aus erreichbaren Markierungen M' gleich ist.



1.1.19 Satz:

Sei $N = (S, T, F, K, W, M_0)$ ein Netz.

(1) Sei y eine S-Invariante, dann gilt für alle $M' \in ERR(M)$: $y \cdot M = y \cdot M'$.

(2) Sei y eine echte S-Invariante. dann gilt für jede Stelle s von N, deren zugehörige Komponente in y positiv ist: s ist k-beschränkt für $k = y \cdot M_0$.

(3) Wenn es eine Markierung M und eine Schaltfolge $w \in T^*$ mit $M[w>M$ gibt, dann ist #w eine T-Invariante von N.

Dieser Satz folgt unmittelbar aus den bisherigen Ausführungen.

Bei (2) beachte man für die Komponente M(s):

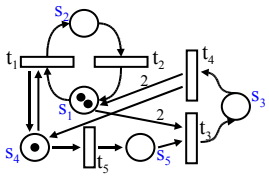
$M(s) \leq y(s) \cdot M(s) \leq y_1 \cdot M_1 + y_2 \cdot M_2 + \ldots + y_n \cdot M_n = y \cdot M = y \cdot M_0 = k$

Aus der linearen Algebra wissen wir, dass die Menge der S- bzw. T-Invarianten einen Untervektorraum bildet.



Anwenden auf das Beispiel aus 1.1.4:

$$C = \begin{pmatrix} -1 & 1 & -2 & 2 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix} \quad M_0 = \begin{pmatrix} 2 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

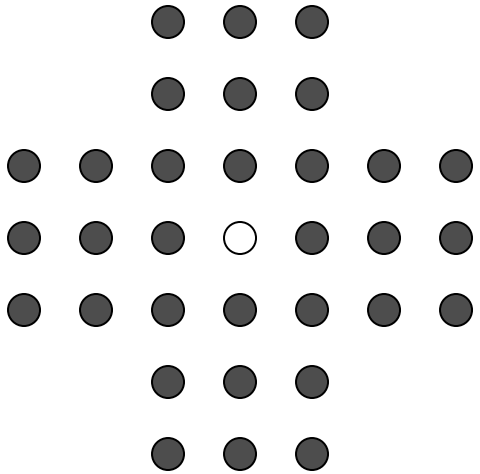



Eine echte S-Invariante lautet: y = (1, 1, 3, 1, 1). Nach Satz 1.1.19 (2) sind daher alle Stellen 3-beschränkt und somit ist auch das Netz beschränkt, siehe auch Beispiel 1.1.7.

(1,1,0,0,0) und (0,0,1,1,1) sind T-Invarianten, daher verändern die Schaltfolgen $t_1t_2$ oder $t_2t_1$ bzw. $t_3t_4t_5$ oder $t_3t_5t_4$ oder $t_4t_3t_5$ oder $t_4t_5t_3$ oder $t_5t_3t_4$ oder $t_5t_4t_3$ (sofern sie schalten können) die Markierung nicht.

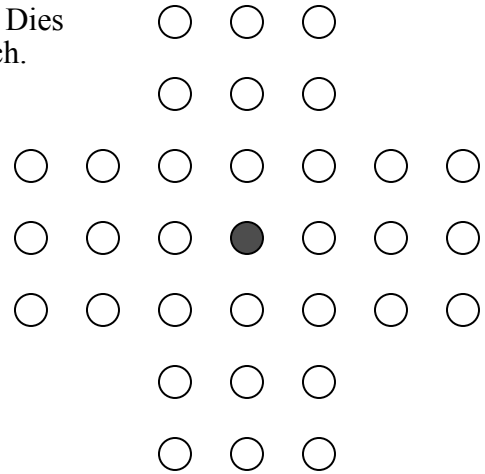


Beispiel 1.1.20: Solitaire-Spiel



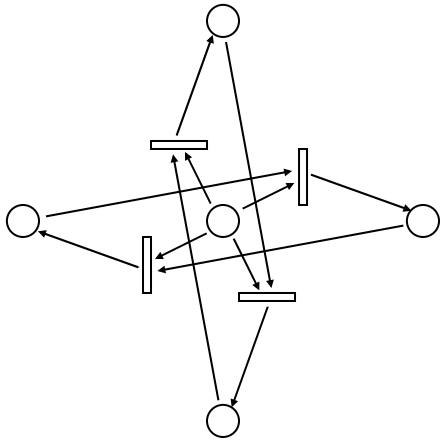
33 Plätze, 32 Steine, d.h., ein Platz bleibt frei. Ein Zug = Springe in gerader Richtung (nicht diagonal) über einen Nachbarstein auf einen freien Platz und entferne den übersprungenen Stein. Ziel: Am Ende soll nur noch ein Stein (möglichst auf einem vorgegebenen Platz) übrig sein.

Meist versucht man, diese Endsituation zu erreichen. Dies geht tatächlich.



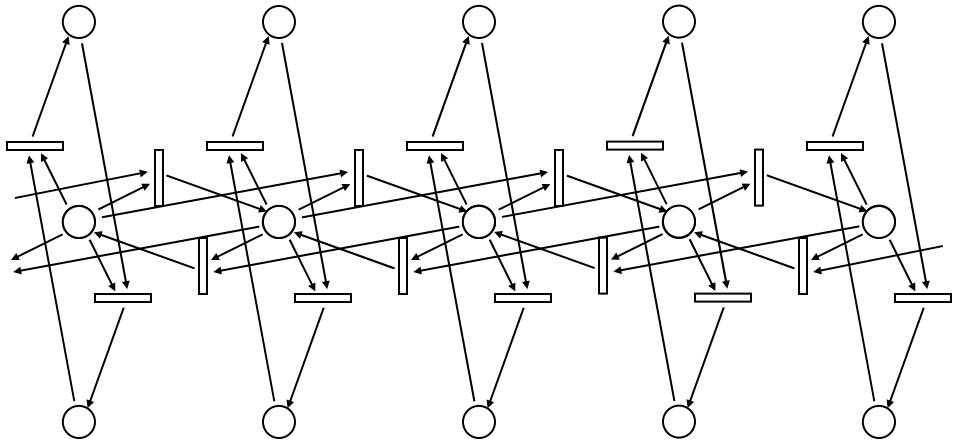


Solitaire als S/T-Netz formulieren. Betrachte einen Platz:

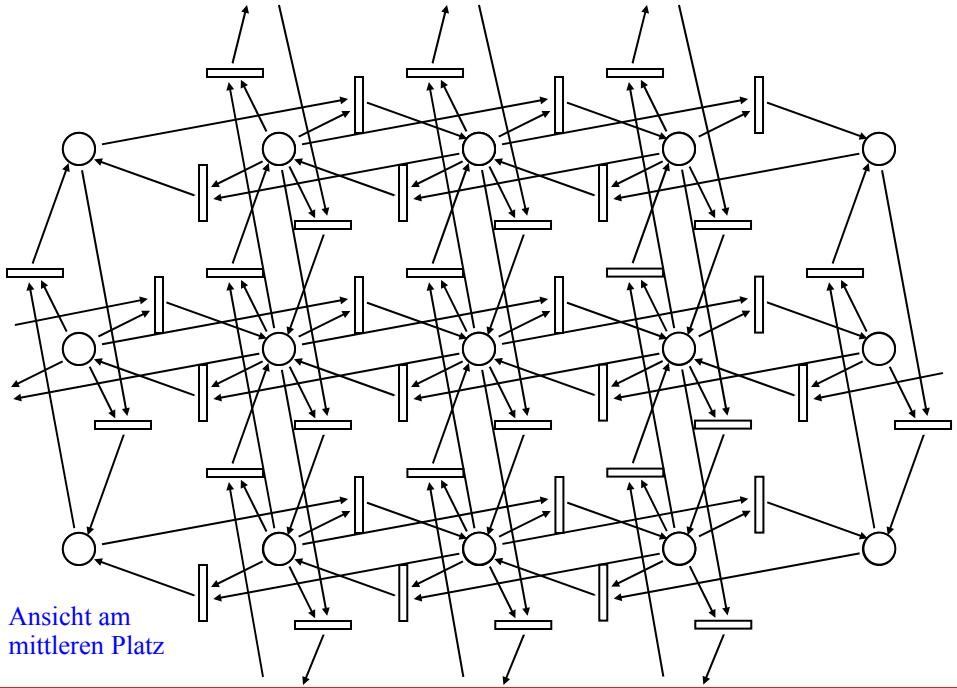




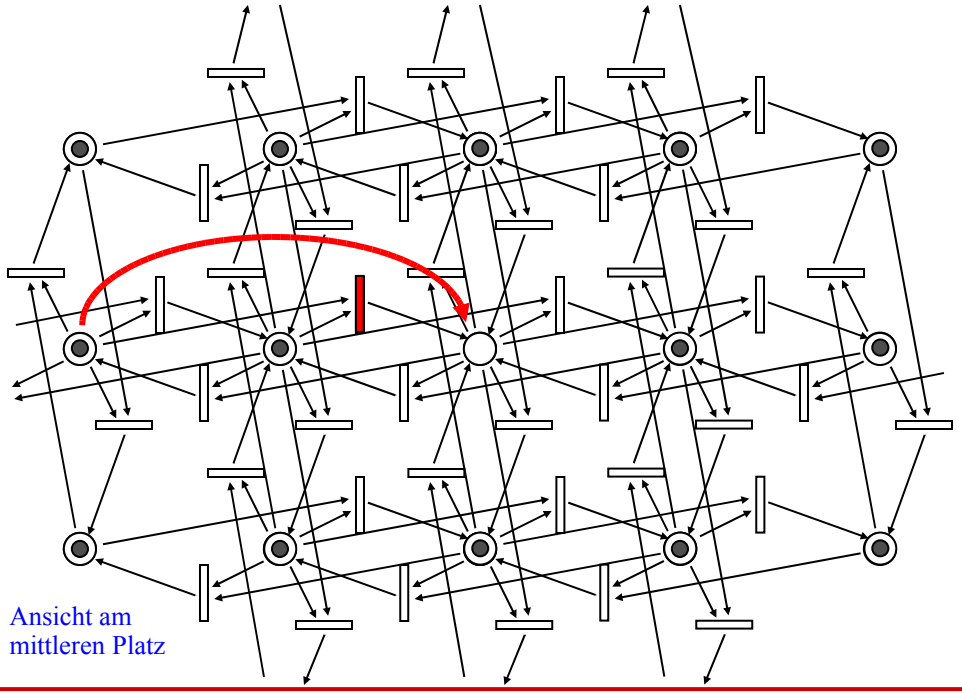
Plätze werden also Stellen, Transitionen sind das Überspringen. Dieses Muster muss nun an jeder Stelle eingefügt werden.



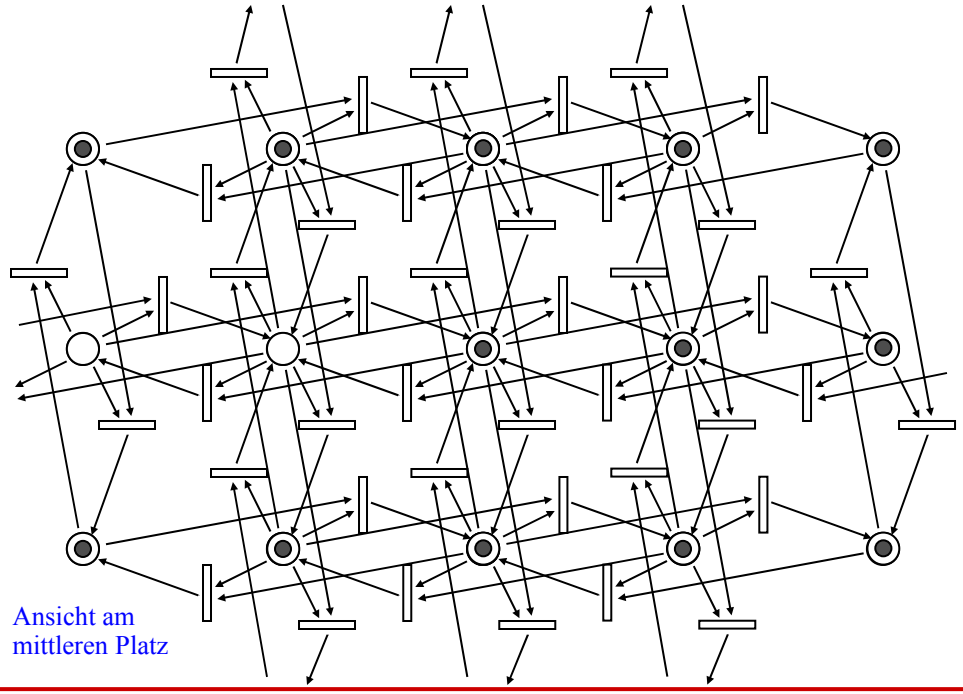




Ansicht am mittleren Platz








Das Ziel des Spiels besteht nun darin nachzuweisen, dass von der Anfangsmarkierung
(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
eine vorgegebene Markierung, zum Beispiel
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
erreicht werden kann und zugleich eine Transitionenfolge hierfür zu finden.



1.1.21: Abschließende Hinweise:

Die Erreichbarkeit ist entscheidbar, allerdings i. A. nur mit einem gewaltigen Zeitaufwand.
Das gleiche gilt für die Beschränktheit. (Beachten Sie, dass Satz 1.1.19 nur ein hinreichendes Kriterium enthält. Der Nachweis der Beschränktheit erfolgt mit sog. Überdeckungsgraphen. Diese kann man auch für die Untersuchung der Lebendigkeit einsetzen.)
Eine unendlich lange Schaltfolge heißt (anschaulich gesprochen) fair, wenn wenn sie jede Transition, die unendlich oft aktiviert ist, auch irgendwann schaltet.
Fairness definieren wir hier nicht. Das Problem, ob ein beliebiges S/T-Netz nur faire Schaltfolgen besitzt, ist nicht entscheidbar.

Hinweise zum umfangreichen Üben/Durchdenken (dies führt aber weit über diese Vorlesung hinaus):

(1) Schreiben/skizzieren Sie ein Paket "ST_Netz" in Ada, das die Stellen-Transitionsnetze darstellt.

(2) Formulieren Sie hierzu eine Prozedur, die zu einem gegebenen S/T-Netz den Erreichbarkeitsgraphen (bis zu einer vorzugebenden Größe) aufbaut.

(3) Implementieren Sie die Aufstellung der Inzidenzmatrix und die Berechnung von S- und T-Invarianten (sofern sie existieren).

(4) Schlagen Sie in der Literatur nach und implementieren Sie weitere Analyseverfahren (z.B. den Aufbau eines Überdeckungsgraphen, den Farkas-Algorithmus, den Nachweis der Erreichbarkeit mittels Backtracking usw.).



## 1.2 Nachrichtenaustausch

Wie können verschiedene Prozesse Daten austauschen? Bei den S/T-Netzen müssen sich die Daten im Vorbereich der Transition befinden; sie werden "lokal" über den Nachbereich anderen Prozessen zur Verfügung gestellt.

Wir betrachten nun zwei andere Mechanismen:
- gemeinsamer Speicherbereich (shared variables),
- Aufbau von Kanälen.

Der Datenaustausch kann synchron und asynchron erfolgen.

Hierbei gehen wir sofort von einer konkreten Programmiersprache aus. Diese ist eine Erweiterung der Sprachelemente, die wir zu Beginn der Grunhdvorlesung eingeführt hatten. Wir beginnen mit dem gemeinsamen Speicherbereich.



1.2.1 Elementare Anweisungen (diese übernehmen wir aus Abschnitt 2.1.5 der Grundvorlesung und fügen await hinzu):

| | |
|---|---|
| skip | Nichtstun. |
| $X := \alpha$ | Wertzuweisung. $\alpha$ ist ein Ausdruck. (Rechne den Ausdruck $\alpha$ aus und lege den erhaltenen Wert in der Variablen X ab.) |
| read (X) | Leseanweisung. (Lies den nächsten Wert ein und lege ihn in der Variablen X ab.) |
| write ($\alpha$) | Schreibanweisung. (Drucke den Wert, den der Ausdruck $\alpha$ besitzt, aus.) |
| await $\beta$ | Warten. *Bedeutung:* Warte, bis $\beta$ wahr ist. ($\beta$ ist ein Boolescher Ausdruck.) |
| $F(X_1, ..., X_n)$ | Aufruf. (Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.) |



1.2.2 Zusammengesetzte Anweisungen (siehe wiederum Abschnitt 2.1.5 der Grundvorlesung sowie Abschnitt 0.2, der zu Beginn an der Tafel vorgetragen wurde):

Hintereinanderausführung oder Sequenz: Wenn $\gamma_1$ und $\gamma_2$ Anweisungen sind, dann ist auch $\gamma_1; \gamma_2$ eine Anweisung.

Alternative: Wenn $\gamma_1$ und $\gamma_2$ Anweisungen und $\beta$ ein Boolescher Ausdruck sind, dann ist auch

if $\beta$ then $\gamma_1$ else $\gamma_2$ fi .

eine Anweisung (else skip darf man weglassen.)

Schleife: Wenn $\gamma$ eine Anweisung und $\beta$ ein Boolescher Ausdruck sind, dann ist auch while $\beta$ do $\gamma$ od eine Anweisung.

Hinzu kommen folgende zusammengesetzte Anweisungen (d. h., wenn $\gamma$, $\gamma_1$, $\gamma_2$, .., $\gamma_n$ Anweisungen sind, dann sind auch die folgenden Konstrukte ... Anweisungen):

Nichtdeterministische Auswahl für n≥2:

($\gamma_1$ or $\gamma_2$ or $\gamma_3$ or ... or $\gamma_n$)

Nebenläufige Abarbeitung für n≥2:

($\gamma_1$ | $\gamma_2$ | $\gamma_3$ | ... | $\gamma_n$)

(Die nebenläufigen Anweisungen $\gamma_i$ nennt man auch "Prozesse").

Blöcke mit gemeinsamen Variablen:

[ local <lokale Deklarationen>; $\gamma$ ]

Wie in Ada lassen wir in den lokalen Deklarationen Konstanten und Initialisierungen zu. Weiterhin darf man jeder Anweisung eine Marke mittels <Name>:: voranstellen.

*Hinweis:* Unsere Beispiele haben meist die Form:

P:: [ local <lokale Deklarationen>; ($\gamma_1$ | $\gamma_2$ | $\gamma_3$ | ... | $\gamma_n$) ]



Beispiel 1.2.3:

```
[ local  L: Integer := 0;
         max: constant Integer := 2;
  ( P1:: while true do                 |  P2:: while true do
         await L < max;                |         await L > 0;
         (L := L+1  or  skip)          |         (L := L-1 or skip)
         od;                           |         od;
  )
]
```

Dieses Programm beschreibt den Erzeuger-Verbraucher-Kreislauf, siehe 1.1.3, mit der Anzahl L der Elemente im Lager, die zwischen 0 und max (=Kapazität) liegen darf.



Dieses Programms besitzt zwei Prozesse. Was ist seine Bedeutung? Ist es genau die gleiche wie die des S/T-Netzes?

Wir führen den Begrif des Zustands für Programme mit mehreren Prozessen ein. Ein Zustand gibt zu jedem Zeitpunkt an, welche Werte die Variablen besitzen und an welchen Stellen im Programm sich die Prozesse befinden.

Hierfür müssen wir die "Stelle im Programm" definieren. Grob gesprochen ist dies eine Stelle zwischen zwei elementaren Anweisungen oder Berechnungen von Bedingungen. Wir nummerieren diese Stellen einfach durch, z.B.:

```
( P1::(1) while (2) true do          |  P2:: (1) while (2) true do
    (3) await  L < max;              |        (3) await L > 0;
        ((4) L := L+1  or  (5) skip) |            ((4) L := L-1 or (5) skip)
    (6) od;                          |        (6) od;
)
```



Hier besitzt man eine gewisse Willkür. Man kann beispielsweise auch die Stelle vor der inneren nebenläufigen Anweisung markieren, also

(4)((5)L := L+1 or (5')skip)

Man kann auch die Berechnungen der Ausdrücke feiner unterteilen, also

(4)((5) L (7) := (6)L+1 or (5')skip)

Dies hängt davon ab, ob die Programme in Zeittakten bearbeitet werden und ob es Beerechnungsteile gibt, die von außen nicht unterbrochen werden können.

Wir formulieren nun die Menge der möglichen Zustände zu der Unterteilung, die auf der vorigen Folie angegeben wurde.

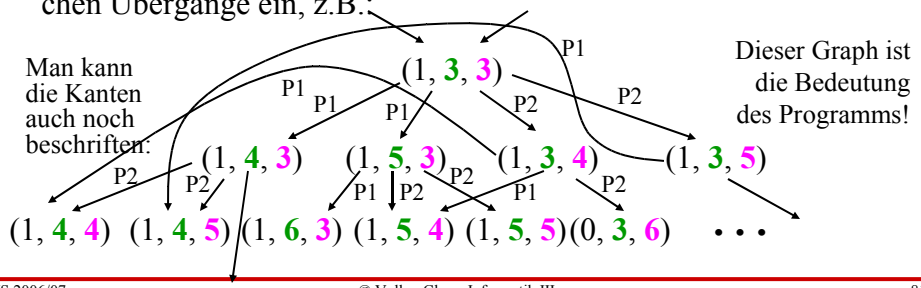
P1:: ① while ② true do
③ await L < max;
(④ L := L+1 or ⑤ skip)
⑥ od;

P2:: ① while ② true do
③ await L > 0;
(④ L := L-1 or ⑤ skip)
⑥ od;

Zustandsmenge

Z = { (a, **i**, **j**) | a ist der Wert von L, **i** ist die Stelle im Programm P1 und **j** ist die Stelle im Programm P2 }

Nun tragen wir wie beim Erreichbarkeitsgraphen 1.1.6 die möglichen Übergänge ein, z.B.:

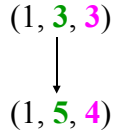




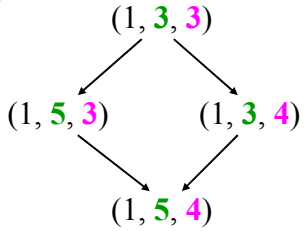

P1:: ① while ② true do
③ await L < max;
(④ L := L+1 or ⑤ skip)
⑥ od;

P2:: ① while ② true do
③ await L > 0;
(④ L := L-1 or ⑤ skip)
⑥ od;

In der Regel betrachtet man hierbei keine "Gleichzeitigkeit". Zum Beispiel ist der Übergang




hier nicht zulässig, man muss vielmehr zwei Schritte in irgendeiner Reihenfolge durchführen:






1.2.4: Legt man die Bedeutung (Semantik) nebenläufiger Programme so fest, dass zwei Prozesse niemals gleichzeitig einen Schritt (= Wechsel des Zustands) ausführen können, sondern stets eine Reihenfolge erzwungen wird, so spricht von einer "Interleaving"-Semantik.

Die Interleaving-Semantik lässt sich aus theoretischer Sicht leichter behandeln als eine Semantik, in der auch Gleichzeitigkeit zugelassen ist. Weiterhin beschreibt die Interleaving-Semantik genau die Verhältnisse, die bei einem *Monoprozessor* vorliegen, der die verschiedenen nebenläufigen Programme alleine ausführen muss (der also die Nebenläufigkeit nur vortäuscht und in Wahrheit die Prozesse verzahnt sequenziell ausführt).



1.2.5: Zur "Granularität" (feinkörnig / grobkörnig): Um die Zustände exakt definieren zu können, muss man festlegen, welche Anweisungsteile "nicht unterbrechbar" sind. Solche Teile werden als in sich geschlossene Einheiten angesehen, deren Ausführung von anderen Ereignissen nicht gestört wird.

Sind in unserem obigen Beispiel "L:=L+1" und "L:=L-1" nicht unterbrechbar auf, so hat nach der nebenläufigen Abarbeitung von (Q1:: L:=L+1 | Q2:: L:=L-1) die Variable L ihren Wert nicht verändert. Sind aber nur die arithmetischen Operationen und die Wertzuweisungen jede für sich nicht unterbrechbar, so kann folgende Möglichkeit bei dieser Abarbeitung auftreten:

Hole den Wert a von L bzgl. Q1; hole den Wert a von L bzgl. Q2; bilde a+1 bzgl. Q1; bilde a-1 bzgl. Q2; weise a+1 der Variablen L zu bzgl. Q1; weise a-1 der Variablen L zu bzgl. Q2.

Am Ende hat L also den Wert a-1 (an Stelle des erwarteten Wertes a).

Sind überhaupt keine Anweisungsteile "nicht unterbrechbar", dann sind alle zeitlich verschachtelten Reihenfolgen möglich.

Beispiel: ( R1:: X:=Y; X := X+1 | R2:: Y:=X; X:=X+1 )

Man kann diese Anweisungen nun beliebig in der zeitlichen Reihenfolge ineinander stecken. Vier Beispiele sind:

*Anfangs seien X=a, Y=b*

| R1: | R2: |
|---|---|
| Y holen | |
| | X holen |
| | Y zuweisen |
| | X holen |
| | um 1 erhöhen |
| | X zuweisen |
| X zuweisen | |
| X holen | |
| um 1 erhöhen | |
| X zuweisen | |

*Am Ende: X=b+1, Y=b*

Zeit

*Anfangs seien X=a, Y=b*

| R1: | R2: |
|---|---|
| | X holen |
| | Y zuweisen |
| Y holen | |
| | X holen |
| | um 1 erhöhen |
| | X zuweisen |
| X zuweisen | |
| X holen | |
| um 1 erhöhen | |
| X zuweisen | |

*Am Ende: X=a+1, Y=a*



( R1:: X:=Y; X := X+1 | R2:: Y:=X; X:=X+1 )

*Anfangs seien X=a, Y=b*

| R1: | R2: |
|---|---|
| Y holen | |
| X zuweisen | |
| X holen | |
| | X holen |
| um 1 erhöhen | |
| | Y zuweisen |
| X zuweisen | |
| | X holen |
| | um 1 erhöhen |
| | X zuweisen |

*Am Ende: X=b+2, Y=b*

Zeit

*Anfangs seien X=a, Y=b*

| R1: | R2: |
|---|---|
| | X holen |
| Y holen | |
| | Y zuweisen |
| X zuweisen | |
| | X holen |
| X holen | |
| | um 1 erhöhen |
| um 1 erhöhen | |
| | X zuweisen |
| X zuweisen | |

*Am Ende: X=b+1, Y=a*

*Prüfen Sie:* Lassen sich auch *Am Ende: X=b+2, Y=b+1* oder *Am Ende: X=a+2, Y=b+1* erreichen? Wie viele verschiedene Möglichkeiten gibt es bei diesem Beispiel?



Hierbei können Konflikte auftreten. Beispielsweise muss man vermeiden, dass ein Prozess Werte in eine Variable (= in einen Speicherbereich) schreibt, während ein anderer Prozess diese Variable ebenfalls verändert. Das Gleiche gilt, wenn irgendein anderes Betriebsmittel (Eingabegerät, Drucker, Übertragungmedium, Beamer usw.) exklusiv genutzt werden muss.

Die Anweisungsfolge, die ungestört von nebenläufigen Prozessen ausgeführt werden muss, nennt man einen *kritischen Abschnitt*. Kann man während ihrer Ausführung den exklusiven Zugriff garantieren, so spricht man vom wechselseitigen oder gegenseitigen Ausschluss.

Gewisse exklusive Zugriffe lassen sich hardwaremäßig sicherstellen, z.B. der Zugriff auf eine Speicherzelle. Für Anweisungsfolgen muss man in der Praxis softwaremäßige Lösungen finden.



1.2.6 Definition:

Ein sequentiell abzuarbeitender Teil eines Programms heißt kritischer Abschnitt, wenn es ein Betriebsmittel gibt, das während der Ausführung dieses Programmteils von keinem anderen (hierzu nebenläufigen) Prozess genutzt werden darf. Prozesse, die auf das gleiche Betriebsmittel zugreifen wollen, *konkurrieren* um dieses Betriebsmittel und *bilden einen Konflikt*.

In einem kritischen Abschnitt für ein Betriebsmittel darf sich zu jedem Zeitpunkt höchstens einer der konkurrierenden Prozesse befinden. Kann man sicherstellen, dass sich bei einem Konflikt zu jedem Zeitpunkt höchstens einer der konkurrierenden Prozesse in seinem kritischen Abschnitt befindet, so spricht man vom wechselseitigen Ausschluss (mutual exclusion).

## 1.2.7 Beispiel

Ein kritischer Abschnitt ist in einem Programm das Ausdrucken von Daten, wenn nur ein Drucker vorhanden ist. Im einfachsten Fall konkurrieren nur zwei Prozesse um den Drucker.

Wir beschreiben im Folgenden den wechselseitigen Ausschluss zuerst mit einem S/T-Netz: "Stellen" sind die elementaren Anweisungen der Prozesse, "Transitionen" sind die Übergänge zu den jeweils nächsten Berechnungen.

Anschließend realisieren wir den wechselseitigen Ausschluss softwaremäßig durch geeignete Kontrollvariable in den beiden Prozessen.

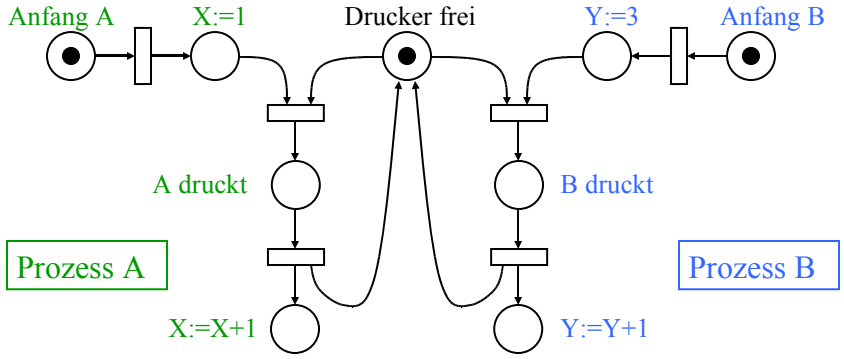
Beispielprogramm: Zwei Prozesse wollen X bzw. Y drucken.

[ local X, Y : Integer;
(A:: X:=1; write(X); X:=X+1 | B:: Y:=3; write(Y); Y:=Y+1 ) ]

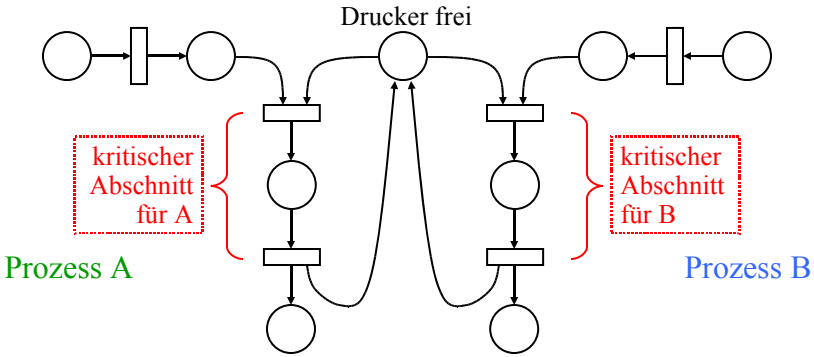


[ local X, Y : Integer;
(A:: X:=1; write(X); X:=X+1 | B:: Y:=3; write(Y); Y:=Y+1 ) ]

Die Programmteile write(X) und write(Y) bilden für jeden der beiden Programmteile kritische Abschnitte. Für dieses Beispiel lässt sich der wechselseitige Ausschluss mit S/T-Netzen leicht modellieren, indem man dem Betriebsmittel "Drucker" eine Stelle "frei" zuordnet :






Beschreibung des wechselseitigen Ausschlusses mit einem S/T-Netz.

Für die Realisierung in der Programmierung wird die Stelle "Drucker frei" durch eine Boolesche Variable dargestellt:



*Obiges Beispiel als Programm mit wechselseitigem Ausschluss:*

```
[ local X, Y: Integer; frei: Boolean := true;
  (A:: X:=1;                    |  B:: Y:=3;
      await frei;               |      await frei;
      frei:=false;              |      frei:=false;
      write(X);                 |      write(Y);
      frei := true;             |      frei := true;
      X:=X+1                    |      Y:=Y+1 )      ]
```

*Diese Realisierung ist nur dann korrekt*, wenn die Anweisungsfolge " await frei; frei := false " nicht unterbrechbar ist! Anderenfalls könnten beide Prozesse (fast) gleichzeitig auf die Variable "frei" zugreifen und sie beide als true erkennen. Beide Prozesse betreten dann gleichzeitig ihre kritischen Abschnitte.

*Hinweis:* Wenn k Drucker für mehrere Prozesse vorliegen, so würde man im S/T-Netz die Stelle "Drucker frei" anfangs mit k Marken belegen. Bei der Übertragung in ein Programm muss man dann eine Variable

Drucker_frei: Natural := k;

deklarieren. Will einer der Prozesse in seinen kritischen Abschnitt eintreten, so würde man schreiben:

await Drucker_frei > 0;
Drucker_frei := Drucker_frei - 1;
< kritischer Abschnitt >;
Drucker_frei := Drucker_frei + 1;

Im allgemeinsten Fall würde man bei der Programmierung noch prüfen, ob die Variable Drucker_frei einen maximalen Wert MAX nicht überschreiten kann, d.h., man würde vor dem Erhöhen von Drucker_frei noch await Drucker_frei<MAX einfügen.



1.2.8 a Definition (das Semaphor, eingeschränkte Form)

Eine Variable vom Typ Natural, die eine Menge von maximal MAX Ressourcen "bewacht", zusammen mit den nicht unterbrechbaren Operationen "Warten und Erniedrigen" und "Warten und Erhöhen" bezeichnet man als Semaphor. (Im Falle MAX=1 kann man auch eine Variable vom Typ Boolean verwenden, siehe oben.)

*Erläuterung des Namens:* Unter einem Semaphor versteht man einen Signalmast, auch "Flügeltelegraph" genant. Solche Masten wurden ab 1790 für die optische Übermittlung von Nachrichten benutzt. Ab 1840 (in Europa ab 1850) wurden sie rasch durch die elektrische Nachrichtenübertragung ("Telegraph") verdrängt. Es gibt sie noch als "Windtelegraphen" in der Schifffahrt. Vorgänger waren bereits bei den alten Griechen um 500 v. Chr. in Gebrauch, vor allem als Feuerzeichen-Übertragung nachts.



*Hinweis:* Das allgemeine Semaphorkonzept wurde 1968 von dem niederländischen Informatiker E. W. Dijkstra eingeführt. Neben der Kontrollvariablen S besitzt jedes solche Semaphor eine Warteschlange, in die alle Prozesse nacheinander eingetragen werden, die zur Zeit noch nicht auf das Betriebsmittel zugreifen können. Das Semaphor aktiviert die Prozesse in der Warteschlange, sobald ein angefordertes Betriebmittel frei ist.

In der Literatur bezeichnet man die Operation "Warten und Erniedrigen", also await S > 0; S := S - 1 auch als P-Operation der Semaphorvariablen S (nach dem niederländischen Wort Passeer = Betreten) oder als Warteoperation. Die andere Operation heißt V-Operation (vom niederländischen Verlaat = Verlassen) oder Signaloperation.



1.2.8 b Definition (das Semaphor, ausführliche Form)

Ein Semaphor besteht aus einer Variablen S des Typs Natural (oder 0..MAX), einer Warteschlange W(S) für Prozesse und folgenden beiden nicht-unterbrechbaren Operationen P und V:

```
procedure P(S: in out Natural);
begin  if S > 0 then S := S-1;
       else  <Stoppe diesen Prozess>;
             <trage ihn in die Warteschlange W(S) ein>; end if;
end P;

procedure V(S: in out Natural);
begin  S := S+1;
if not isempty(W(S)) then <Wähle einen Prozess A aus W(S) aus>;
       <aktiviere dessen Ausführung ab der P-Operation in A,
         durch die A gestoppt wurde>;  end if;
end V;
```

*Obiges Beispiel als Programm mit allgemeinem Semaphor:*

```
[ local X, Y : Integer; S: semaphor;
   (A:: X:=1;                |  B:: Y:=3;
        P(S);                |       P(S);
        write(X);            |       write(Y);
        V(S);                |       V(S);
        X:=X+1               |       Y:=Y+1 )      ]
```

Semaphore sind eine Standardtechnik, um den wechselseitigen Ausschluss zu realisieren, bzw. allgemein, um eine gegebene Menge von Betriebsmitteln mehreren auf sie zugreifenden Prozessen zur Verfügung zu stellen, ohne dass eine Verklemmung eintreten kann.
(Vgl. "Scheduler" in Vorlesungen über Betriebssysteme.)



Problem: Kann man durch ein Programm, das nur unsere Anweisungen benutzt (also keine allgemeinen Semaphore kennt), den wechselseitigen Ausschluss sicherstellen, falls keine unterbrechbaren Operationen vorliegen (nur das Schreiben in eine Variable sei nicht-unterbrechbar)?

*Wie muss man also das bisherige Programm*

```
[ local X, Y : Integer; frei: Boolean := true;
   (A:: X:=1;                |  B:: Y:=3;
        await frei;          |       await frei;
        frei:=false;         |       frei:=false;
        write(X);            |       write(Y);
        frei := true;        |       frei := true;
        X:=X+1               |       Y:=Y+1 )      ]
```

*abändern? Oder kann man dies gar nicht garantieren??*



Vorschlag: Führe eine Variable ein, die nur die Werte PA und PB annehmen kann.

```
[ local  type prozess is (PA, PB);
         Nr: prozess := PA;  X, Y: Integer;

  (A:: X:=1;                |  B:: Y:=3;
       Nr := PB;            |       Nr := PA;
       await Nr = PA;       |       await Nr = PB;
       write(X);            |       write(Y);
       Nr := PB;            |       Nr := PA;
       X:=X+1               |       Y:=Y+1    )   ]
```

Jeder Prozess gibt dem anderen Prozess die Berechtigung, als erster in den kritischen Abschnitt einsteigen zu können. Ist dies ein Konzept, das Fehler vermeidet?



Dieses Programm verhindert zwar, dass sich beide Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden können, aber wenn die Anweisungsfolgen (wie oft üblich) in einer unendlichen Schleife stehen, dann können die beiden Prozesse nur abwechselnd ihren kritischen Abschnitt betreten, und beide Prozesse müssen dauernd aktiv bleiben, sonst wartet der andere Prozess ewig:

⇒ Unbrauchbare Lösung

```
[ local  type prozess is (PA, PB);
         Nr: prozess := PA;  X, Y: Integer;

  (A:: while true do          |  B:: while true do
         X:=1; Nr := PB;      |         Y:=3; Nr := PA;
         await Nr = PA;       |         await Nr = PB;
         write(X);            |         write(Y);
         Nr := PB; X:=X+1     |         Nr := PA; Y:=Y+1
       od                     |       od    )          ]
```

1.2.9 Problemformulierung: Gesucht wird also eine Softwarelösung, bei der jeder Prozess unabhängig vom anderen ist, außer in dem Fall, dass beide zur gleichen Zeit in ihren kritischen Abschnitt eintreten wollen. Hierzu gibt es diverse Lösungen in der Literatur (z.B.: T. Dekker 1965, G. L. Peterson 1981, S. Owicki und L. Lamport 1982).

Versuchen Sie zunächst selbst, für "Vorbereitung A" bzw. B und "Nachbereitung A" bzw. B eine Lösung des allgemeinen Problems zu finden:

```
[ local <Deklarationen>;
  (A:: while true do                 |  B:: while true do
         Vorbereitung A;             |         Vorbereitung B;
         kritischer Abschnitt A;     |         kritischer Abschnitt B;
         Nachbereitung A             |         Nachbereitung B
       od                            |       od   )                ]
```



Wir geben hier nur die Lösung von Peterson an. Jeder Prozess hat hierbei eine eigene Boolesche Variable PrA bzw. PrB, die den Wunsch, in den kritischen Abschnitt einzutreten, signalisiert. Zusätzlich gibt es eine Variable "dran", die dem anderen Prozess den Vortritt lässt, sofern dieser auch gerade in den kritischen Abschnitt will.

Diese Lösung erfüllt die geforderten Eigenschaften:

- Es kann sich zu jedem Zeitpunkt nur ein Prozess in seinem kritischen Abschnitt befinden.
- Jeder Prozess kann in seinen kritischen Abschnitt gelangen unabhängig davon, wo sich der andere Prozess befindet oder ob er noch aktiv ist.
- Es tritt keine Verklemmung auf.



*1.2.10 Die Lösung von Peterson (1981):*
*(or ist hier das Oder in einem Booleschen Ausdruck)*

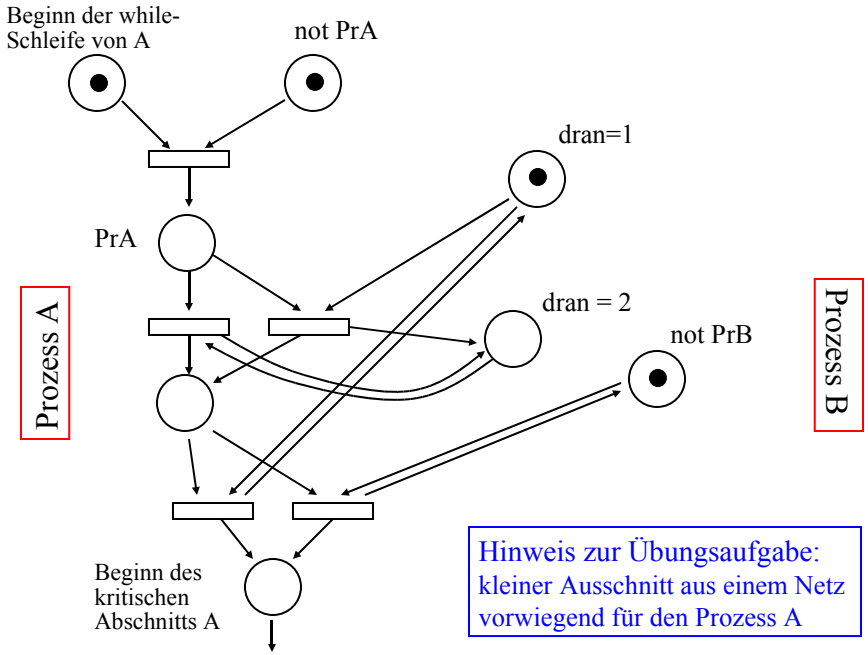
```
[ local  dran: Integer:=1;  PrA, PrB: Boolean:= false;

(A:: while true do                 |  B:: while true do
       PrA := true;                |        PrB := true;
       dran := 2;                  |        dran := 1;
       await (not PrB) or (dran=1);|        await (not PrA) or (dran=2);
       < kritischer Abschnitt A >; |        < kritischer Abschnitt B >;
       PrA := false;               |        PrB := false;
       ...                         |        ...
     od                            |      od              )       ]
```



In der Vorlesung wird an der Tafel die Arbeitsweise dieses Vorgehens genauer erläutert. Machen Sie sich diese Arbeitsweise an einem Ablaufdiagramm klar!

Zu diesem Programm kann man ein Stellen-Transitions-Netz zeichnen, das die Arbeitsweise genau widerspiegelt. Einen Beweis für die Korrektheit der Peterson-Lösung kann man dann über dieses S/T-Netz führen.

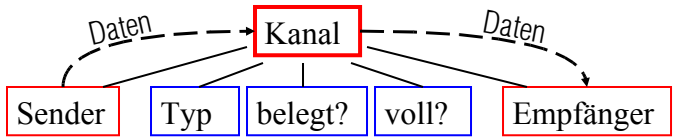
*Übungsaufgabe:* Konstruieren Sie dieses S/T-Netz zu der oben angegebenen Peterson-Lösung.






1.2.11 Kanäle: Wir haben einige Aspekte des Nachrichtenaustausches vorgestellt, der über einen gemeinsamen Speicherbereich erfolgt. Anders funktioniert das Telefonieren: Dort wird jedem solchen Nachrichtenaustausch ein eigener Kanal zur Verfügung gestellt, der nach dessen Beendigung einer anderen Kommunkation zugeordnet werden kann.

Anstelle des Ablegens von Informationen in einem gemeinsamen Speicherbereich betrachten wir nun also die Nutzung von Kanälen, über die eine Verbindung zwischen zwei Partnern hergestellt werden kann.




uni-/ bidirektional, synchron / asynchron



Für Kanäle erlauben wir übergreifend den Datentyp

"channel [1..max] of T"

in den Daten d vom Typ T mittels CH ← d vom Sender hineingelegt und aus dem Daten dieses Typs vom Empfänger mittels CH → X seiner Variablen X zugewiesen werden können. (CH ist eine Variable des eingeführten Datentyps.). Benutzen zwei Prozesse den gleichen Kanal, so muss einer Sender und einer Empfänger sein und es können Daten nur vom Sender an den Empfänger geschickt werden.

Ein Kanal ist wie eine Warteschlange organisiert und er besitzt in der Regel eine Kapazität. Die Daten, die zuerst hineingesteckt werden, kommen auch als erste wieder heraus (FIFO-Prinzip), und die Warteschlange kann meist nur die begrenzte Zahl "max" von Daten aufnehmen.

Mit einem Kanal muss weiterhin eine Boolesche Variable "belegt" verbunden sein, die einen Kanal nicht frei gibt, sofern er derzeit benutzt wird.



Die Anweisung CH ← X in einem Prozess P besagt also:

Wenn der Kanal CH nicht belegt ist, so wird er als belegt gekennzeichnet und P ist der Sender für CH;
wenn CH belegt ist und bisher nur der Empfänger dem Kanal zugeordnet ist, so wird P der Sender von CH;
wenn CH belegt ist und schon zwei Partner besitzt, so muss P unter diesen Partnern der Sender sein.

Trifft eine dieser drei Bedingungen zu, so wird geprüft, ob die Daten (hier: der Wert von X) vom Typ T sind und ob CH noch Platz für die Aufnahme eines weiteren Datums besitzt.

Trifft auch dies zu, so wird der Wert von X in den Kanal gelegt; anderenfalls erfolgt eine geeignete Fehlermeldung.

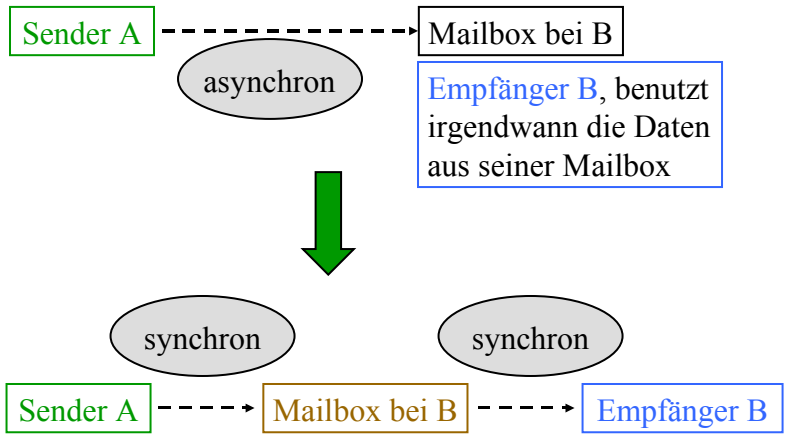
Analog ist die Anweisung CH → X in einem Prozess P zu interpretieren.

Will man einen Datenaustausch zwischen zwei Prozessen installieren, so muss man zwei Kanäle verwenden (wie beim Telefonieren). Will man den Zustand der Kanäle noch überwachen oder unabhängig von den Daten Kontrollinformationen übertragen, so muss man einen oder zwei weitere Kanäle hinzunehmen (wie beim Telefon).

Man muss noch festlegen, ob eine synchrone Verbindung besteht (wenn A sendet, so muss B zeitgleich empfangen; A kann erst weiterarbeiten, wenn B alle Daten empfangen hat) oder ob die Daten aynchron überliefert werden (z.B. in eine Mailbox gelegt werden; allerdings muss dann die Mailbox mit A oder mit dem Kanal synchron zusammenarbeiten).



1.2.12: Hieraus folgt: Eine asynchrone Kommunikation zwischen zwei Prozessen kann man durch zwei synchrone Kommunikationen zwischen drei Prozessen simulieren:






Ein solcher Fall liegt beim Erzeuger-Verbraucher-Kreislauf vor (siehe 1.1.3): Der Erzeuger schickt asynchron seine Produkte an den Verbraucher. Fasst man das Lager als zusätzlichen Prozess auf, so lässt sich dieser Vorgang synchron darstellen (siehe nächste Folie).

Wir wollen diese Ausführungen nun mit einem Beispiel beenden, an dem die prinzipielle Arbeitsweise von Kanälen abgelesen werden kann. Das Thema des Nachrichtenaustausches wird in Vorlesungen über Betriebssysteme, Verteilte Systeme und Sichere Systeme weiter vertieft.

Als Beispiel wählen wir den Erzeuger-Verbraucher-Kreislauf mit der Kapazität 10 des Lagers (hier bedeutet "or" die nichtdeterministische Auswahl; die erzeugten und verbrauchten Elemente sind von Datentyp T).



*1.2.13 Beispiel* EL, LV: channel [1..1] of T;

```
local X: T;                 local Z: T; k: 0..10 :=0;        local Y: T;
                            puffer: array [1..10] of T;
while true do                                                while true do
"erzeuge X";                while true do                    LV → Y;
EL ← X                        ( if k<10 then                 "verbrauche Y"
od                                  EL → Z; k:=k+1;          od
                                    puffer[k] := Z  fi
                              or
                              if k>0 then
                                    LV ← puffer[k];
                                    k := k-1  fi )
                            od
```

Erzeuger | Lager | Verbraucher

Noch einige Begriffe:

Geblockte Übertragung: Daten werden meist nicht einzelnen, sondern in größeren Einheiten (Blöcken) übertragen. Dies erhöht vor allem die Effizienz der Übertragung.

Gepackte Daten: Daten werden oft noch komprimiert, damit sie weniger Platz benötigen. Beim Empfänger müssen sie dann wieder "entpackt" werden (Beispiel: zip-Files).

Synchron: Der Empfänger übernimmt die Daten, während der Sender sendet.

Asynchron: Die Daten werden irgendwo gepuffert, bis der Empfänger sie abholt. Das Puffern kann auch im Kanal integriert sein.

Blockierendes Senden: Der Sender kann erst weiterarbeiten, wenn alle gesendeten Daten entweder beim Empfänger angekommen sind oder vom Puffer des Kanals aufgenommen wurden.

Blockierendes Empfangen: Der Empfänger kann erst weiterarbeiten, wenn alle gesendeten Daten bei ihm abgespeichert sind.

Den Datenaustausch mittels synchronem blockierendem Senden und blockierendem Empfangen bezeichnet man als Rendezvous. Dies ist in Ada realisiert, siehe folgenden Abschnitt 1.3.



# 1.3 Nebenläufigkeit in Ada

## 1.3.1 Einführende Begriffe

*Prozess* = Abarbeitung eines Algorithmus, wobei nur immer höchstens eine Stelle im Algorithmus aktiv ist.

Einen Prozess kann man mit einem einzelnen Prozessor (Monoprozessor) abarbeiten. Er läuft sequenziell ab. Es können viele Prozesse gleichzeitig aktiv sein (Nebenläufigkeit).

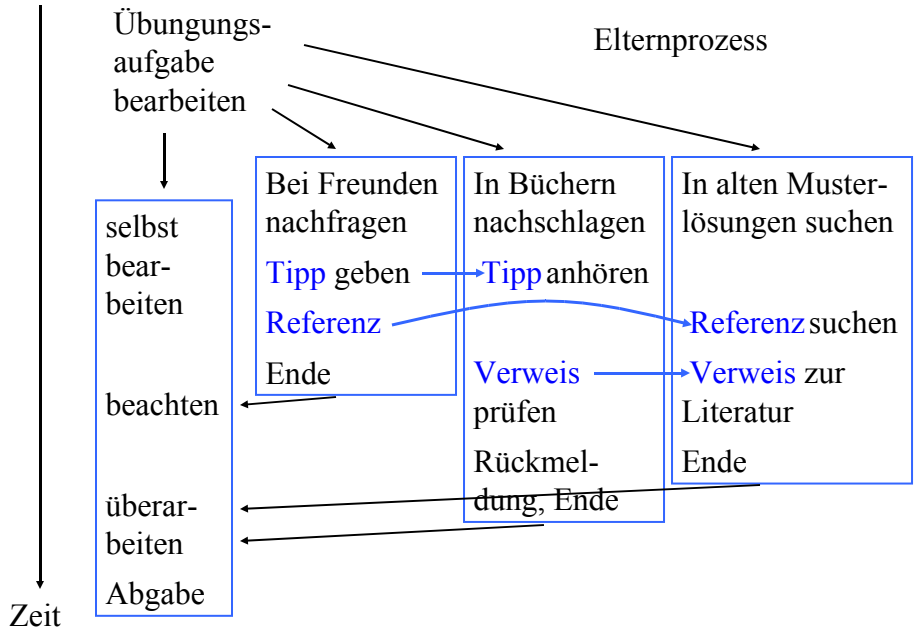
Das Programmstück, das einen Prozess beschreibt, nennen wir *Prozesseinheit*. Schlüsselwort in Ada hierfür: task. Sie kann in einem Deklarationsteil vereinbart werden. Sie ist in die Spezifikation und die Implementierung (task body) geteilt. Der Nachrichtenaustausch zwischen Prozessen erfolgt implizit durch Kanäle (zwischen einem Entry-Aufruf und einem accept).

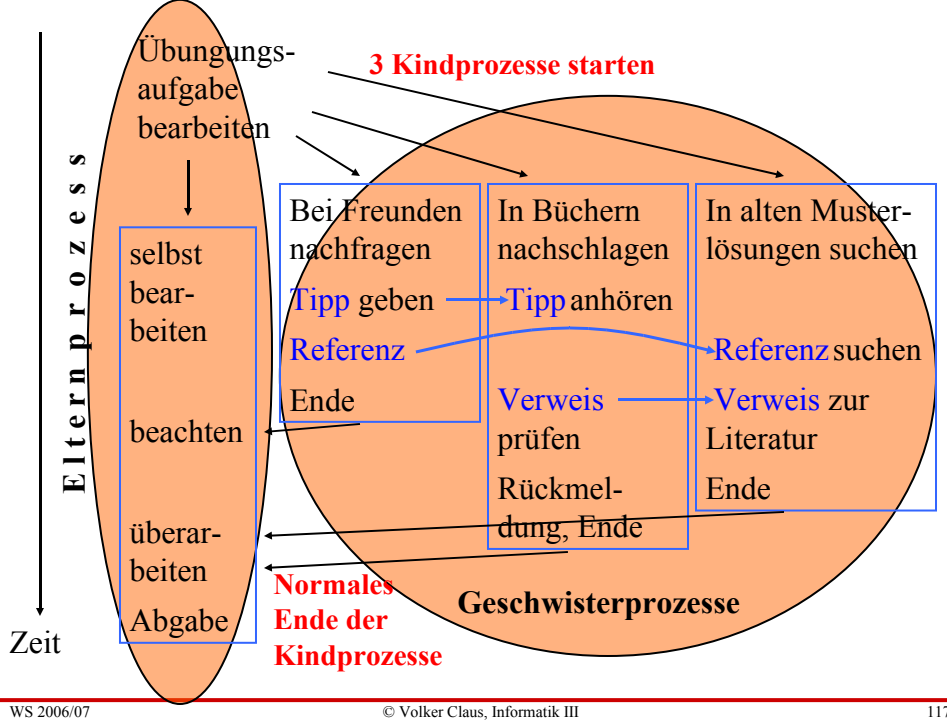


Es gibt in Ada keine Anweisung der Form "starte Prozess X". Vielmehr wird eine Prozesseinheit in Ada gestartet, sobald ihre Deklaration im Programmablauf erreicht wird (impliziter Start). Die Einheit, in der eine Prozesseinheit deklariert wird, heißt *Elternprozess*; andere im gleichen Deklarationsteil vereinbarte Prozesseinheiten heißen *Geschwisterprozesse*.

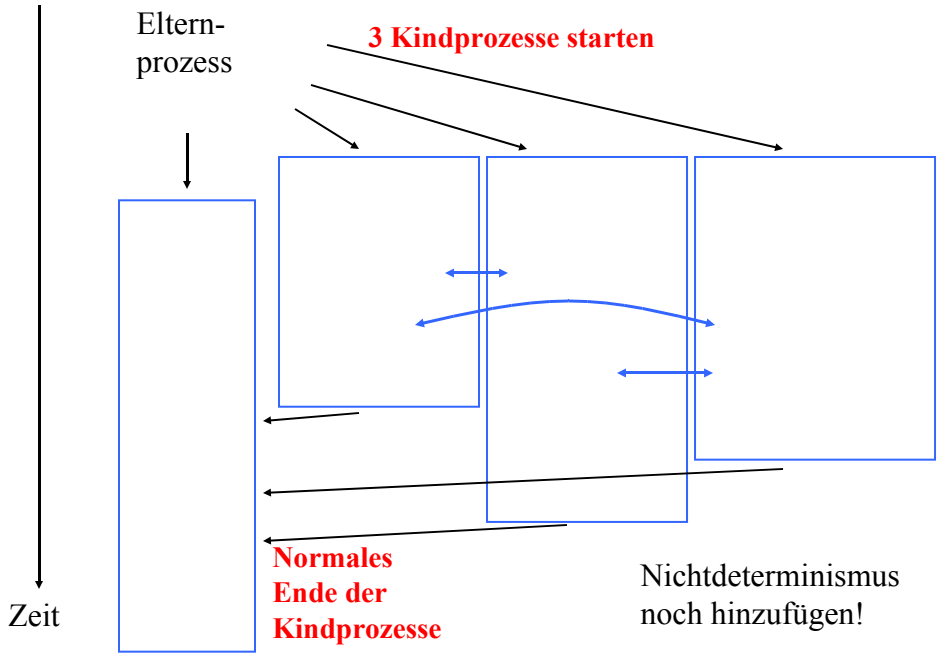
In einer Prozesseinheit kann es mehrere Stellen geben, an denen eine Synchronisation oder ein Datenaustausch erfolgen sollen. Diese Stellen bezeichnet man als Entry-Schnittstellen; sie erhalten einen (Entry-) Namen und der Ablauf, der bei der Synchronisation erfolgen soll, wird in einer accept-Anweisung genau ausformuliert. Die accept-Anweisung trägt den Entry-Namen; für eine Synchronisation wird sie wie eine Prozedur aufgerufen, erhält evtl. aktuelle Parameter und kann Werte über out-Variablen zurückgeben.












Wir benötigen also Sprachelemente für

Deklaration eines Prozesses: task (Spezifikation und Rumpf).

Starten eines Prozesses: Erfolgt implizit mit der Deklaration.

Normales Ende eines Prozesses: Erreichen von end
(ein Elternprozess endet aber frühestens, wenn alle seine Kindprozesse beendet sind) oder eigene Beendigung mittels terminate (in select-Anweisung).

Datenaustausch zwischen den Prozessen:
entry für die Spezifikation des Austausches
entry-Aufruf und accept für die Programmstellen

Gewaltsames Abbrechen eines Prozesses: abort

Nichtdeterministische Auswahl: select ... or ... or ... end select

Warten in nichtdeterministischen Alternativen: delay [until]



## 1.3.2 a Zugehörige Syntax in Ada

```
single_task_declaration ::=
        task defining_identifier [is task_definition];
task_definition ::= {task_item} [private {task_item}]
        end [task_identifier]
task_item ::= entry_declaration | representation_clause
task_body ::= task body defining_identifier is
        declarative_part
        begin
        handled_sequence_of_statements
        end [task_identifier];
```

1.3.2 b Syntax (*Zu "statement" siehe Grundvorlesung 2.8.5 ff)*

entry_declaration ::= entry defining_identifier
[(discrete_subtype_definition)] parameter_profile;

*(parameter_profile beschreibt den Parameterteil in einer Programmeinheit, vgl. Grundvorlesung 3.1.1)*

entry_call_statement ::= *entry*_name [actual_parameter_part];

*(actual_parameter_part beschreibt die aktuellen Parameter)*

accept_statement ::=
accept *entry*_direct_name [(entry_index)]
parameter_profile
[do handled_sequence_of_statements
end [*entry*_identifier]];

*(zu handled_sequence_of_statements siehe ebenfalls 3.1.1, insbesondere ist diese Folge von Anweisungen nie leer)*

entry_index ::= expression



1.3.2 c Syntax

select_statement ::= selective_accept | timed_entry_call |
conditional_entry_call | asynchronous_select

selective_accept ::= select [guard] select_alternative
{or [guard] select_alternative}
[else sequence_of_statements] end select;

guard ::= when condition =>

select_alternative ::= accept_alternative | delay_alternative |
terminate_alternative

accept_alternative ::= accept_statement
[sequence_of_statements]

delay_alternative ::= delay_statement [sequence_of_statements]

terminate_alternative ::= terminate;

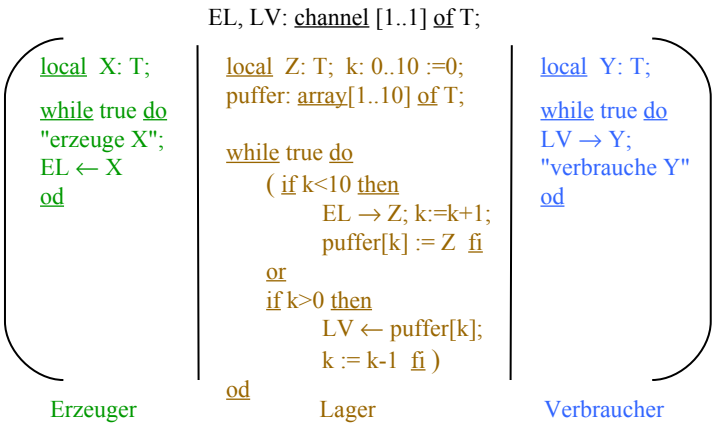
delay_statement ::= delay_until_statement |
delay_relative_statement

delay_until_statement ::= delay until *delay*_expression;

delay_relative_statement ::= delay *delay*_expression;



1.3.3 Beispiel: Der Nachrichtenaustausch erfolgt in Ada über Kanäle. Wir übertragen daher Beispiel 1.2.13:




DasProgramm wird zu procedure ... und die drei Prozesse werden zu task Erzeuger, task Lager und task Verbraucher.



Die Kanäle spielen keine direkte Rolle. Ihre Namen können als Aufrufstellen im jeweiligen Prozess verwendet werden, sofern kein Namenskonflikt entsteht.

Die Sendeoperation "←" wird durch einen Entryaufruf zu der entsprechenden Stelle ersetzt, die Empfangsoperation "→" wird zu einer accept-Anweisung.

Verwendet man die gleichen Bezeichner, so wird
aus EL ← X; der Aufruf Lager.EL(X); und
aus EL → Z; wird accept EL(U: in Float) do Z := U; end;
(hier wurde willkürlich U als Name für den formalen Parameter der Aufrufstelle gewählt; dieser muss nun natürlich auch in der Spezifikation von task Lager für diesen entry benutzt werden).

Wir fügen für den Abbruch noch eine Boolesche Variable Ende und für die Interaktion eine Character-Variable C hinzu.

```
procedure Erzeuger_Verbraucher is
        Ende: Boolean := false; C: Character;
        task Erzeuger;
        task Lager is
                entry EL(U: in Float);
        end;
        task Verbraucher;
                entry LV(W: in Float);
        end;
        < Die drei Taskrümpfe, siehe unten, hier einfügen >
begin   Put("Erzeuger, Lager und Verbraucher sind aktiv.");
        while not Ende loop get(C); Ende := C='0'; end loop;
        Put("Erzeuger, Lager und Verbraucher enden nun.");
end Erzeuger_Verbraucher;
```



Anstelle der Endlosschleife verwenden wir die Schleife mit der Bedingung "not Ende". Als "Produkt" nehmen wir reelle Zahlen.

```
task body Erzeuger is
X: Float := 1.0;
begin
while not Ende loop          -- Schleife, um ständig Zahlen mit der
    X := Sin(X+1.0);         -- Variablen X neu zu erzeugen.
    Lager.EL(X);             -- Entry-Aufruf: X wird ans Lager gesandt.
end loop;
end Erzeuger;                -- Schluss, falls "Ende" den Wert true
erhält.
```



```
task body Lager is                -- fast wörtliche Übertragung nach Ada
Z: Float; K: Integer range 0..10 := 0;
Puffer: array(1..10) of Float;
begin
while not Ende loop               -- statt der Endlosschleife
  select                          -- "(" wird zu select
    when K < 10 =>                -- das if wird in ein when umgewandelt
      accept EL(U: in Float) do Z := U; end EL;
      K := K+1; Puffer(K) := Z;
  or
    when K > 0 =>                 -- das if wird in ein when umgewandelt
      Verbraucher.LV(Puffer(K));
      K := K-1;
  end select;                     -- ")" wird zu end select
end loop;
end Lager;                        -- Schluss, falls "Ende" den Wert true erhält.
```



```
task body Verbraucher is

Y: Float;
begin
while not Ende loop
      accept LV(W: in Float) do Y := W; end LV;
      < hier können Anweisungen zur Verarbeitung von Y folgen>
end loop;
end Verbraucher;
```

Wenn Ende auf false gesetzt wird, kann es vorkommen, dass der task Lager syeine while-Schleife beendet, der task Verbraucher aber gerade noch einmal in die Schleife hineingelangt ist. Dann würde "Verbraucher" beim entry LV beliebig lange warten. Lösung hierfür:

```
task body Verbraucher is

Y: Float;
begin
while not Ende loop
      select
          accept LV(W: in Float) do Y := W; end LV;
          < hier können Anweisungen zur Verarbeitung von Y folgen>
      or
          when not Ende => delay 0.5
      end select
end loop;
end Verbraucher;
```



## 1.3.4: Synchronisation und Kommunikation zweier Prozesse in Ada (1):

Die Interaktion zweier Prozesse erfolgt in Ada durch "Entry-Schnittstellen". Ein Entry ist eine durch accept bezeichnete Stelle in einer Prozesseinheit. Diese kann wie ein Unterprogrammruf von einem anderen Prozess aufgerufen werden, allerdings erfolgt keine Verzweigung des Programmablaufs zu dieser Stelle, sondern es wird eine Synchronisation vorbereitet: Der aufrufende Prozess wartet an der Entry-Aufrufstelle solange, bis der gerufene Prozess die Entry-Schnittstelle erreicht hat (= Synchronisation). Dann wird der hinter accept zwischen do und end stehende Programmteil, der wie ein Unterprogrammrumpf aufgebaut sein kann, ausgeführt. Erst danach trennen sich die beiden Prozesse wieder, d.h., sie fahren unabhängig von einander mit ihrer jeweils nächsten Anweisung fort. Diesen Vorgang bezeichnet man als ***Rendezvous***.

Ein Entry kann von einem beliebigen anderen Prozess aufgerufen werden. Rufen mehrere Prozesse gleichzeitig die gleiche Entry-Schnittstelle, so werden diese Aufrufe nacheinander abgearbeitet, wobei der wechselseitige Ausschluss für den do-end-Programmteil garantiert wird, d.h., dieses Programmstück kann nicht durch weitere Entry-Aufrufe unterbrochen werden.



## Synchronisation und Kommunikation zweier Prozesse in Ada (2):

Trifft umgekehrt ein Prozess auf eine seiner Entry-Schnittstellen (also auf ein accept), so wartet er dort, bis ein anderer Prozess dieses Entry aufruft. Erfolgt dieser Aufruf, so wird der zwischen do und end stehende Programmteil ausgeführt (in dieser Zeit wartet der aufrufende Prozess nichtstuend) und anschließend trennen sich die beiden Prozesse wieder.

Man beachte, dass das Rendezvous mit dem Ende der accept-Anweisung endet. Danach folgende Anweisungen wirken sich nur auf die Wartezeit weiterer aufrufender Prozesse aus. In unserem Beispiel 1.3.3 endet das Rendezvous an der Entry-Schnittstelle EL in der Prozesseinheit Lager mit dem end nach dem accept. Die anschließend folgenden Anweisungen

K := K+1; Puffer(K) := Z;  gehören nicht mehr zum Rendezvous.

Hierdurch wird offensichtlich eine *Synchronisation* erreicht. Zugleich können durch den Entry-Aufruf, der aktuelle Parameter enthalten kann, Daten an den gerufenen Prozess übergeben werden und umgekehrt können am Ende des do-end-Programmstücks Daten zum rufenden Prozess zurückfließen, falls out-Variablen übergeben wurden. Dadurch wird eine *Kommunikation* zwischen den Prozessen realisiert.



## Synchronisation und Kommunikation zweier Prozesse in Ada (3):

Durch diesen Mechanismus kann es vorkommen, dass ein rufender Prozess oder ein Prozess, der für einen Aufruf bereit ist, ewig wartet. (Dann ist der Programmierer "selbst schuld", siehe aber unten (5).)

Man beachte die *Asymmetrie des Rendezvous-Mechanismus:* Während der rufende Prozess genau weiß, mit wem er eine Kommunikation durchführt (der Entry-Aufruf besteht ja aus dem Namen des Tasks, gefolgt von einem Punkt und dem Namen der Entry-Schnittstelle; fehlt der Name des Tasks, so ist stets der Elternprozess gemeint), kennt der gerufene Prozess den Namen seines Partners nicht. Um die rufenden Prozesse korrekt bedienen zu können, muss mit jeder Entry-Schnittstelle eine Warteschlange für rufende Prozesse verbunden sein: Ein Prozess, der auf ein accept trifft, führt ein Rendezvous mit dem ersten in der Warteschlange stehenden Prozess durch (oder wartet, bis ein Aufruf eintrifft).

In der Syntax gibt es die Möglichkeit, Entries als geheim einzustufen: task_definition ::= ... [ private {task_item}] ... . Dies wird man dann tun, wenn diese Entry-Schnittstellen nur von Unterprozessen genutzt werden sollen.

Synchronisation und Kommunikation zweier Prozesse in Ada (4):

Weiterhin lässt die Syntax bei der entry-Deklaration einen diskreten Untertyp [(discrete_subtype_definition)] und beim accept-statement einen [(entry_index)] zu. Diese beiden Varianten werden nur wichtig, wenn man "Entry-Familien" benutzt (dies sind viele Entries, die alle eine ähnliche Spezifikation besitzen).

Hinweise zur accept-Anweisung: Diese enthält keinen Deklarationsteil; man kann aber einen hinzufügen, indem man zwischen do und end Blöcke verwendet. Der do-end-Teil kann fehlen; in diesem Fall dient der Entry-Aufruf nur zur Synchronisation. Weiterhin kann es sinnvoll sein, die gleiche Entry-Schnittstelle an mehreren Stellen im Prozess zu platzieren; es sind daher zu einem Entry-Namen mehrere accept-Anweisungen erlaubt.
Wie üblich sind goto- und exit-Anweisungen, die von außen in eine accept-Anweisung gelangen oder eine accept-Aweisung verlassen, verboten. Allerdings ist ein return erlaubt, wodurch die umfassende Prozedur und zugleich das laufende Rendezvous beendet wird. Es gibt viele weitere Einschränkungen, siehe hierzu die Ada-Literatur.



Synchronisation und Kommunikation zweier Prozesse in Ada (5):

Natürlich muss es Mechanismen geben, um ein "unverschuldetes" unendliches Warten zu beenden. In Ada sind dies für einen rufenden Prozess:

*timed_entry_call* der Form
select <Entry-Aufruf> <und evtl. weitere Anweisungen>
or <delay-Alternative> <und evtl. weitere Anweisungen> end select

Bedeutung: In der delay-Alternative kann man eine Zeit vorgeben; hat bis dahin das Rendezvous des "Entry-Aufruf" nicht begonnen, so wird der Aufruf abgebrochen und die hinter der delay-Alternative aufgeführten Anweisungen werden durchgeführt.

*conditional_entry_call* der Form
select <Entry-Aufruf> <und evtl. weitere Anweisungen>
or <Folge von Anweisungen, evtl. leer> end select

Bedeutung: Ist der gerufene Prozess nicht zum sofortigen Rendezvous bereit, so wird der or-Teil ausgeführt.

Ebenso gibt es Abbruchmöglichkeiten für den Prozess, der an einer Entry-Schnittstelle auf ein Rendezvous wartet.



1.3.5: (Eingeschränkter) Nichtdeterminismus in Ada (1)

Es gibt die select-Anweisung mit den or-Alternativen. Diese Anweisung ist *vorwiegend für die nichtdeterministische Behandlung von Entry-Aufrufen bzw. accept-Anweisungen* vorgesehen, siehe Syntax 1.3.2 c.

In der Praxis ist Nichtdeterminsimus (also willkürliches Verhalten von Prozessen an gewissen Stellen des Kontrollflusses) schwer kalkulierbar. Insbesondere können nicht alle Folgemöglichkeiten durchprobiert oder vorhergesehen werden. Entscheidet sich ein Prozess falsch, so kann er in Verklemmungen geraten.

Die select-Anweisung besitzt daher Varianten, um eine Entscheidung bzgl. der Kommunikation hinauszuzögern, bis bessere Informationen vorliegen, um eine Entscheidung, die nicht erfolgreich war, rückgängig zu machen oder durch eine andere zu ersetzen oder um eine eingeschlagene Alternative gewaltsam abzubrechen. Hierfür kann man einschränkende Bedingungen (sog. Wächter, "guards") vor die Alternativen setzen (when ... => ...), man kann die Auswahl davon abhängig machen, ob innerhalb einer Zeitspanne (delay <Zeit in Sekunden>) eine Alternative nicht erfolgreich war, man kann den eigenen Prozess zum Abbruch anbieten (terminate) oder man kann einen Prozess gewaltsam abbrechen (abort).



(Eingeschränkter) Nichtdeterminismus in Ada (2)

Betrachte ein einfaches Beispiel:

```
select
    accept Bildschirmausgabe (...) do ... end; ...
or
    when <Bildschirm an> => delay 300.0;
    <rufe Bildschirmschoner> ...
end select;
```

Liegt für 300 Sekunden kein Rendezvous mit dieser Entry-Schnittstelle "Bildschirmausgabe" vor, so wird die zweite Alternative ausgewählt.

Steht in einer Alternative das Sprachsymbol terminate, so wird dies einem "übergeordneten" Prozess, der zum Ende kommen möchte, als Möglichkeit angeboten, den laufenden Prozess an dieser Stelle zu beenden.

Bei den vielen Möglichkeiten muss man zwischen der rufenden (aktiver Prozess) und der gerufenen Seite (passiver Prozess) unterscheiden, für die ein Sprachelement unterschiedliche Bedeutung haben kann. Einzelheiten siehe Ada-Literatur und zum Teil im Programmierkurs, Teil 2.

# 2. Maschinennahe (abstrakte) Programme





## 2.1 Registermaschinen (= random access maschine = RAM)

Die folgenden 10 Folien sind eine Wiederholung aus der Grundvorlesung, siehe in "Informatik I" den Abschnitt 6.6.

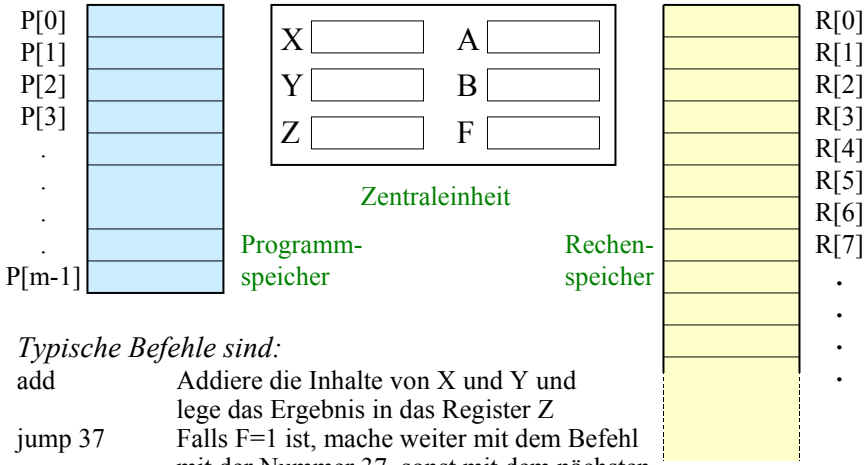
2.1.1 Veranschaulichung von Registermaschinen

Eine Registermaschine ist wie ein kleiner Mikroprozessor aufgebaut. Sie besteht aus

1. einer *Zentraleinheit* mit 6 Registern (Speicherzellen): 3 Register X, Y und Z für arithmetische und logische Operationen, ein Adressregister A für den Zugriff auf Rechenspeicherzellen, ein Befehlszählregister B, in dem die Adresse des auszuführenden Befehls steht, und ein "Flag"-Register F, in dem das Ergebnis von Operationen abgelegt wird;
2. einem *Programmspeicher* P, in dem nacheinander die Befehle des abzuarbeitenden endlichen Programms stehen;
3. einem (unendlich langen) *Rechenspeicher* R mit durchnummerierten Speicherzellen, die jeweils eine (beliebig große) ganze Zahl aufnehmen können.



**Struktur der RAM**




*Typische Befehle sind:*

| | |
|---|---|
| add | Addiere die Inhalte von X und Y und lege das Ergebnis in das Register Z |
| jump 37 | Falls F=1 ist, mache weiter mit dem Befehl mit der Nummer 37, sonst mit dem nächsten |
| LeftShift X | Schiebe den Inhalt von X eine Stelle nach links und füge eine 0 an |
| comp (Y > Z) | if der Inhalt von Y ist größer als der von Z then F:=1 else F:=0 fi |
| read X | X := $R_{<A>}$, d.h., sei a der Inhalt von A, so weise zu X:=$R_a$ |
| load A,4 | A := 4 (Wertzuweisung einer Konstanten an ein Register) |



2.1.2 *Der übliche Befehlssatz einer Registermaschine* lautet (hierbei bezeichnen V und V' beliebige Register, c ist eine ganze Zahl, $b \in \mathbb{N}_0$, $R_k$ ist die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ ist eine Vergleichsoperation; außer beim Jump-Befehl wird nach jedem Befehl B um 1 erhöht):

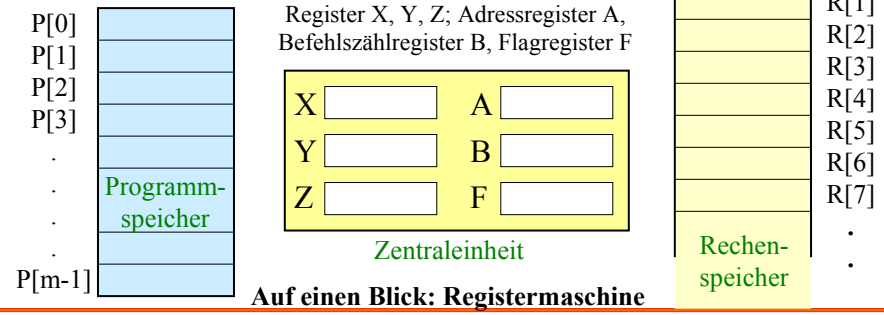
| Befehl | Bedeutung | Befehl | Bedeutung |
|---|---|---|---|
| load V,c | V := c | copy V,V' | V := V' |
| read V | V := $R_{<A>}$ | write V | $R_{<A>}$ := V |
| succ | X := X+1 | add | X := Y+Z |
| sub | X := Y-Z | shift | X := X div 2 |

| | |
|---|---|
| comp (σ) | if XσY then F:=1 else F:=0 fi |
| jump b | if F=1 then B:=b else B:=B+1 fi |
| stop | Anhalten, Ende der Berechnung |

Diese Befehle finden sich in allen Mikroprozessoren; diese haben in der Regel aber noch viel mehr Befehle, insbesondere bzgl. der Adressierung, der Verschiebungen von Daten, der eingebauten Kellermechanismen und der Unterbrechungsbefehle. Die Ausformulierung von Algorithmen als Registermaschine ähnelt daher der Maschinenprogrammierung.

Der übliche Befehlssatz einer Registermaschine (V und V' seien Register, c eine ganze Zahl, $b \in \mathbb{N}_0$, $R_k$ die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ eine Vergleichsoperation; außer bei jump wird nach jedem Befehl B um 1 erhöht):

| Befehl | Bedeutung | Befehl | Bedeutung |
|---|---|---|---|
| load V,c | V := c | copy V,V' | V := V' |
| read V | V := $R_{<A>}$ | write V | $R_{<A>}$ := V |
| succ | X := X+1 | add | X := Y+Z |
| sub | X := Y-Z | shift | X := X div 2 |
| comp (σ) | if XσY then F:=1 else F:=0 fi | | |
| jump b | if F=1 then B:=b else B:=B+1 fi | | |
| stop | Anhalten, Ende der Berechnung | | |




**Auf einen Blick: Registermaschine**



*Beispiel 2.1.3:* Test auf Teilbarkeit durch 2 (die Zahl $n \geq 0$ stehe anfangs in $R_0$). Naheliegende Lösung: Subtrahiere von n ständig die Zahl 2. Die Zahl n sei hier der Variablen I zugewiesen. Dann lautet das Programmstück: **while I>1 do I:=I-2 od;** Anschließend steht in I der Rest der Division von n durch 2.
Übertrage dieses Programmstück in den Befehlssatz der Registermaschine (I ⇔ Register Y, Zahl 2 ⇔ Z; anfangs steht n in der Rechenspeicherzelle $R_0$, am Ende stehe das Ergebnis in $R_0$; sei a der Inhalt von A, so bezeichnet $R_{<A>}$ den Inhalt der Rechenspeicherzelle $R_a$). Man erhält folgendes Registermaschinenprogramm:

| Nummer | Befehl | Erläuterung |
|---|---|---|
| 0: | load A,0 | A := 0 |
| 1: | read Y | Y := n (= $R_0 = R_{<A>}$) |
| 2: | load Z,2 | Z := 2 |
| 3: | load X,1 | X := 1 |
| 4: | comp ( ≥ ) | teste, ob X ≥ Y ist (beachte: in X steht eine 1) |
| 5: | jump 10 | if 1 ≥ Y then weiter bei Befehl mit Nummer 10 fi |
| 6: | sub | X := Y-Z (also X:=Y-2) |
| 7: | copy Y,X | Y := X |
| 8: | load F,1 | F:=1 (um einen Sprung nach 3 zu erzwingen) |
| 9: | jump 3 | weitermachen beim Befehl mit der Nummer 3 |
| 10: | write Y | in A steht noch 0, also: $R_0$ := Y |
| 11: | stop | |



Betrachtet man nun Registermaschinen, die nur *auf ganzen Zahlen* arbeiten, so muss man zuvor prüfen, ob n < 0 ist oder nicht. Wir fügen daher vor das oben angegebene Programmstück noch die Anweisung if I < 0 then I := 0 - I fi und übertragen dieses Ada-Programm wiederum in ein Registermaschinenprogramm und erhalten das rechts stehende Programm:

Man kann sich überzeugen, dass jedes Ada-Programm in ein solches Registermaschinenprogramm übersetzt werden kann und dass sich dieser Prozess automatisieren lässt (siehe 2.2). Solche einfachen Sprachen heißen in der Praxis "Maschinensprachen" oder "Maschinencode"; sie werden meist in Form von Assemblern ein wenig lesbarer und komfortabler gemacht.

```
 0: load A,0
 1: read Y
 2: load X,0
 3: comp(≤)
 4: jump 9
 5: copy Z,Y
 6: load Y,0
 7: sub
 8: copy Y,X
 9: load Z,2
10: load X,1
11: comp (≥)
12: jump 17
13: sub
14: copy Y,X
15: load F,1
16: jump 10
17: write Y
18: stop
```



Die verschiedenen Typen von Registermaschinen unterscheiden sich in ihren (elementaren) Datentypen und in ihren Befehlssätzen, siehe Übungen. Meist fügt man noch weitere Speicher (z. B. ein Eingabeband, das nur gelesen werden darf, und ein Ausgabeband, das nur beschrieben werden darf) hinzu.

Die englische Bezeichnung "random access maschine" stammt daher, dass die Maschine über das Register A einen "wahlfreien Zugriff" auf die Rechenspeicherzellen erlaubt.

Registermaschinen geben in der Regel die Komplexität von Programmen recht gut wieder. Sie werden daher vielen theoretischen Untersuchungen zugrunde gelegt.

## 2.1.4 *Hinweis zur Übersetzung von Ada-Programmen*

In den Programmen von Registermaschinen gibt es nur zwei Kontrollstrukturen:

- Übergang zum nächsten Befehl (dies entspricht dem ";" in Ada). Dies wird dadurch realisiert, dass nach jeder Ausführung eines Befehls B um 1 erhöht wird.
- Bedingter Sprung (falls F=1) zum Befehl mit der Nummer b (jump b), sofern die Nummer b im Programm vorkommt (anderenfalls Fehlerabbruch).

Man kann alle strukturierten Anweisungen in höheren Programmiersprachen durch diese beiden Kontrollstrukturen simulieren. In Ada sind Sprünge mit dem Schlüsselwort goto zugelassen. Anstelle der Nummern verwendet Ada *Marken*, dies sind Bezeichner, die in << ... >> eingeschlossen und vor eine Anweisung gesetzt werden. Zwei Beispiele folgen.



if X ≥ Y then Z := 1; else X := Y; end if; X := X+1; ...

wird im Registerprogramm zu (willkürlich wurde 20 als Nummer des ersten Befehls gewählt)

```
20: comp (<)      vergleiche X mit Y bzgl. "kleiner"
21: jump 25       springe zum else-Zweig
22: load Z,1      Z := 1 (then-Zweig ausführen)
23: load F,1      bereite einen "unbedingten" Sprung vor
24: jump 26       überspringe den else-Zweig
25: copy X,Y      X := Y (else-Zweig ausführen)
26: succ          X := X+1
```

In Ada kann man dieses Programmstück direkt nachbilden:

```
F := X < Y; if F then goto ELSE_ZWEIG; end if;
Z := 1; goto DANACH;
<<ELSE_ZWEIG>>  X := Y;
<<DANACH>>  X := X+1; ...
```



while X < Y loop X := X+1; end loop; Z:=X;...

wird im Registerprogramm zu

```
30: comp (≥)      vergleiche X mit Y bzgl. nicht-"kleiner"
31: jump 25       überspringe die Schleife
32: succ          X := X+1  (Schleifenrumpf)
33: load F,1      bereite einen "unbedingten" Sprung vor
34: jump 30       zurück zur Schleifen-Bedingung
35: copy Z,X      Z := X  ...
```

In Ada lautet dieses Programmstück:

```
<<Schleife>> F := X ≥ Y; if F then goto danach; end if;
X := X+1; goto Schleife;
<<danach>>  Z := X; ...
```



Ähnliche Übersetzungen von Ada-Konstrukten in einfache Programme, die nur aus Wertzuweisungen, Sprüngen und Hintereinanderausführungen (einschließlich Marken) bestehen, lassen sich leicht angeben.

Zumindest für die Kontrollstrukturen (=Anweisungsteil) sollte daher klar sein, dass man sie in eine Registermaschine übertragen kann. Bei der Übertragung der Datenstrukturen muss man sich den erforderlichen Speicherbedarf aller Teilstrukturen merken und beim Zugriff auf Variablen oder deren Komponenten entsprechende Werte in das Adressregister laden.

Somit ist im Prinzip klar, dass man jedes Ada-Programm in eine Registermaschine umwandeln kann, die die gleiche Resultatsfunktion berechnet. Diesen Prozess kann man teilweise automatisieren.

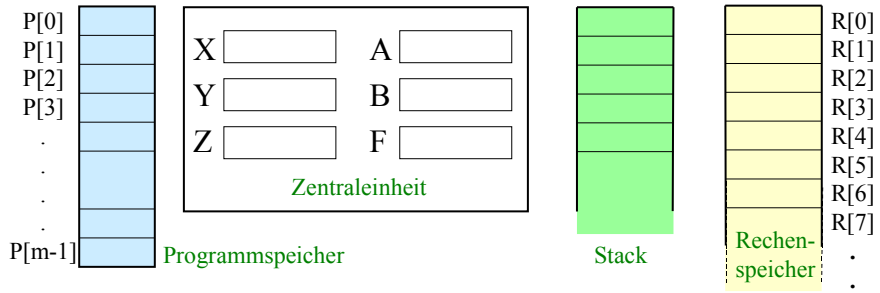
## 2.2 Stackmaschine

Wir fügen der Registermaschinen nun einen weiteren Speicher zur einfacheren Abarbeitung von Klammerstrukturen (Ausdrücke, Rekursion) hinzu. Solche Strukturen werden mit Hilfe eines Stacks (Keller, Pushdown, siehe Grundvorlesung 3.5.4, 4.3.3.a und 6.6.3) bearbeitet. Bei der Übersetzung von Programmiersprachen wird als Zwischensprache oft die Sprache einer Stackmaschine verwendet.

Im Folgenden beschreiben wir die Stackmaschine mit Hilfe der Registermaschine. Manche Befehle der RAM werden dann überflüssig, wenn man alle Berechnungen nur noch über den Stack laufen lässt und die zugrunde liegende Maschine darüber hinaus nur die Abarbeitungsreihenfolge überwacht.



**Struktur der Stackmaschine** (Luxusausführung)



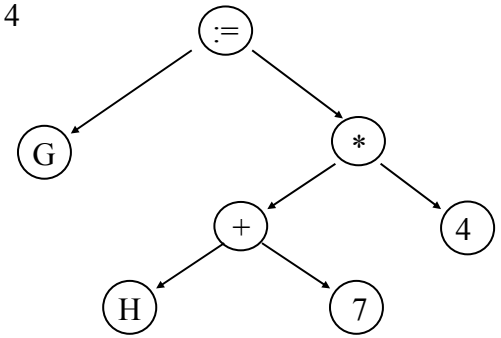

*Befehle der Stackmaschine*

| | |
|---|---|
| empty | Stack löschen |
| isempty | F:=1, falls Stack leer ist, sonst F:=0 |
| pop | Lösche oberstes Stackelement |
| push(V) | Lege Inhalt von V auf den Stack (fehlt V, so ist das Register X gemeint) |
| top(V) | Kopiere oberstes Stackelement nach V (fehlt V, so ist das Register X gemeint) |
| addStack | ≡ top(Z); pop; top(Y); pop; X:=Y+Z; push(X); |
| subStack, multStack, divStack, modStack | *analog* ("+" durch die jeweilige Operation ersetzen) |
| shiftStack | ≡ top; pop; shift; push; (*analog* succStack ≡ top; pop; succ; push; ) |
| LoadStack c | ≡ Load X,c; push; |
| ReadStack | read X; push; (*analog* WriteStack ≡ top; write X; pop; *beachte:* hier wird gelöscht) |
| compStack(σ) | ≡ top(Y); pop; top(X); comp(σ); push(Y); |

Alle weiteren Befehle wie bei der Registermaschine.



*Beispiel:*
G := (H + 7) * 4




A := "Adresse von H im Rechenspeicher"; ReadStack;

LoadStack 7; addStack; LoadStack 4; multStack;

A := "Adresse von G im Rechenspeicher"; WriteStack;



Der Ableitungsbaum wird *postorder* ausgeführt und legt hierdurch die Abarbeitungsreihenfolge fest.
Wertzuweisungen können also durch Stack-Befehle vollständig ersetzt werden. Somit können die auf Registern basierenden Befehle add, sub, succ, shift usw. entfallen.
Streicht man die Register bezogenen Befehle und fügt weitere Speicherstrukturen hinzu (für das Programm, den Kellerspeicher und die Halde, damit sie durch die Maschine verwaltet werden können), so erhält man eine Stackmaschine, wie sie bereits Anfang der 1970er Jahre für Pascal (sog. P-Code) und später für andere Sprachen (z.B.: JVM = Java Virtual Machine) verwendet werden.

Die Java Virtual Machine (JVM), in die Java-Programme vor (bzw. während) der Ausführung übersetzt werden, besteht aus Stackbefehlen, z. B. (die Variable H möge in der Rechenspeicherzelle 18 stehen; Java findet die Variable in der Symboltabelle, siehe 2.4):

| Beschreibung | unsere Darstellung | JVM-Befehl |
|---|---|---|
| Lade die Variable H auf den Stack | Load A,18; ReadStack | iload H |
| Addiere die obersten Stackelemente | addStack | iadd |
| Bedingte Verzweigung | comp (σ); ... | if σ ... |
| Sprung zum Befehl, der 5 Stellen im Programm weiter steht | <wir hatten nur absolute Adressen benutzt; gemeint ist B := B+5 > | goto 5 |



Viele Programmiersprachen nutzen bei der Übersetzung mathematische (oder abstrakte) Maschinen aus. Hierdurch wird die Übersetzung in der Regel durchsichtiger und somit prinzipiell auch leichter verifizierbar. Beispiele:

Pascal benutzt den P-Code,

Java die JVM (Java Virtual Machine),

Smalltalk den Bytecode,

Haskell die Spineless Tagless G-Machine (STGM),

Prolog die Warren Abstract Machine (WAM).

Details: siehe Vorlesung über Compilerbau.



# 2.3 Attributierung von Grammatiken

2.3.1 Erinnerung EBNF (Grundvorlesung 1.10). Man kann diese stets in eine kontextfreie Darstellung übertragen.

| EBNF Darstellung | kontextfreie Darstellung |
|---|---|
| $X ::= u_1 \mid u_2 \mid ... \mid u_r$ | $X \to u_1$, $X \to u_2$ ..., $X \to u_r$ |
| $X ::= uv$ | $X \to uv$ |
| $X ::= u_1 (v_1 \mid v_2) u_2$ | $X \to u_1 v_1 u_2$, $X \to u_1 v_2 u_2$ |
| $X ::= u[v]w$ | $X \to uw$, $X \to uvw$ |
| $X ::= \{u\}$ | $X \to H$, $H \to \varepsilon$, $H \to uH$ |



2.3.2 Wiederholung (vergleiche Definition 2.7.1 der Grundvorlesung):
Eine kontextfreie Grammatik ist ein Viertupel $G = (N, T, P, S)$ mit

(1) N ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen oder Variablen),
(2) T ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $N \cap T = \emptyset$,
(3) $S \in N$ ist ein Nichtterminalzeichen (das Startsymbol),
(4) $P \subset N \times (N \cup T)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen; statt (X,u) schreibt man $X \to u$).

Auf $(N \cup T)^*$ führt man die "Ableitungsrelationen" $\Rightarrow$ und $\Rightarrow^*$ ein:
(a) Es gilt $u \Rightarrow v$ genau dann, wenn man die Wörter u und v in der Form $u = xAy$, $v = xwy$ mit $x, y \in (N \cup T)^*$ und $A \to w \in P$ schreiben kann. Man sagt: v ist aus u in einem Schritt ableitbar.
(b) Es gilt $u \Rightarrow^* v$ genau dann, wenn entweder $u = v$ ist oder wenn es Wörter $z_0, z_1, ..., z_k \in (N \cup T)^*$ für ein $k \geq 1$ gibt mit $u = z_0$, $v = z_k$, $z_i \Rightarrow z_{i+1}$ für alle $i = 0, 1, ..., k-1$. Man sagt, v ist aus u ableitbar.
Die von G erzeugte Sprache ist die Menge
$L(G) = \{ w \in T^* \mid S \Rightarrow^* w \} \subseteq T^*$.

## 2.3.3 Beispielgrammatik $G_1$ mit N={S,R,Z}, T={.,0,1}:

| ausführlich wie in BNF | knappe Darstellung |
|---|---|
| <Binärzahl> → <Ziffernfolge> | S → R |
| <Binärzahl> → <Ziffernfolge>.<Ziffernfolge> | S → R.R |
| <Ziffernfolge> → <Ziffer> <Ziffernfolge> | R → ZR |
| <Ziffernfolge> → <Ziffer> | R → Z |
| <Ziffer> → 0 | Z → 0 |
| <Ziffer> → 1 | Z → 1 |

*Erzeugte Sprache:*

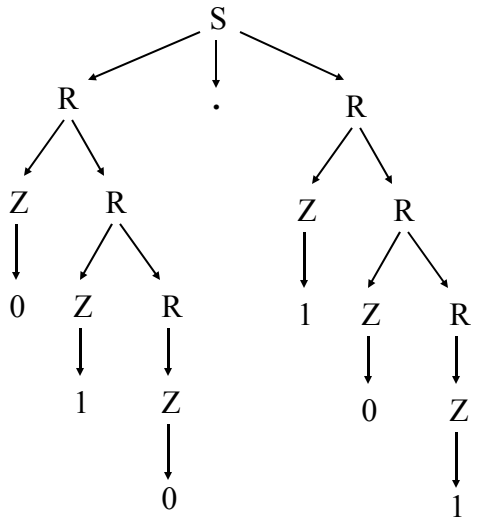
$L(G_1) = TT^* \cup TT^* \{.\} TT^*$

$= \{u \in T^* \mid u \neq \varepsilon\} \cup \{u.v \mid u,v \in T^*, u \neq \varepsilon \text{ und } v \neq \varepsilon\}$.

Man kann dies als die Menge aller binär dargestellten Zahlen auffassen (mit führenden Nullen). Zum Wort w = 010.101 gehört also die Zahl
$0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 2{,}625$ (dezimal)



## Ableitungsbaum für w = 010.101




Ziel: Wir wollen nun die Bedeutung dieses abgeleiteten Wortes w, also die Zahl 2,625, aus dem Baum berechnen.



## 2.3.4 Hinzunahme von Attributen

Wir ordnen jedem Zeichen aus N∪T Attribute zu, die mit Gleichungen, welche den Produktionen der Grammatik angefügt werden, berechnet werden.

Wenn a ein Attribut für X ist, so schreiben wir X.a. Gleiche Zeichen müssen durch einen Index 1, 2, ... unterschieden werden.

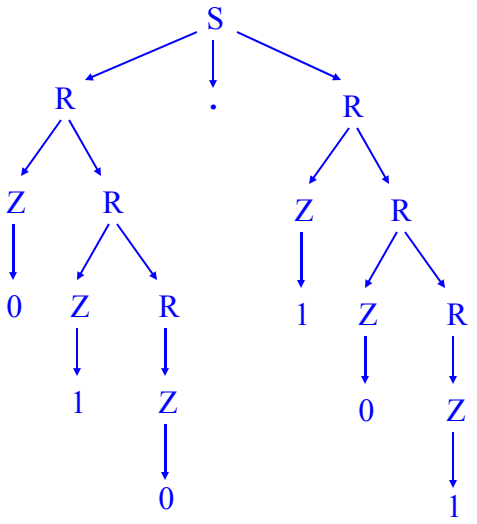
Attribute in unserem Beispiel sollen sein:
S.Wert, R.Wert, Z.Wert, 0.Wert, 1.Wert, R.Länge.

Die Gleichungen zu den Produktionen lauten dann (die Zahlen werden hier fett und grün dargestellt):

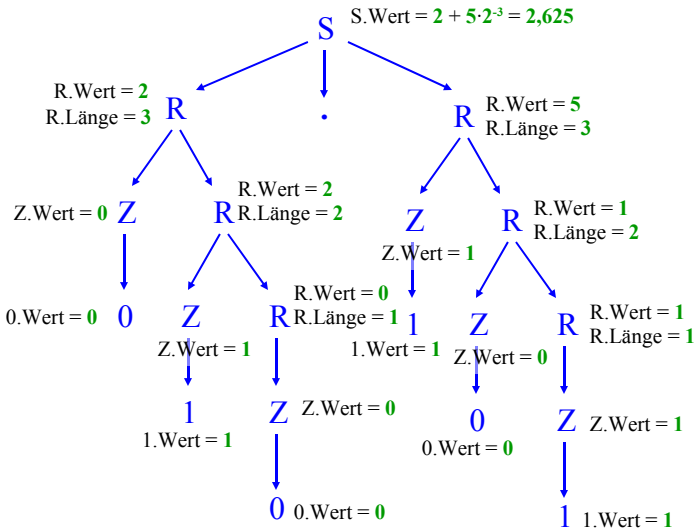
| Produktion | Gleichungen |
|---|---|
| S → R | S.Wert = R.Wert |
| S → $R_1.R_2$ | S.Wert = $R_1$.Wert + $R_2$.Wert·$\mathbf{2}^{-R_2.\text{Länge}}$ |
| $R_1$ → Z$R_2$ | $R_1$.Wert = Z.Wert·$\mathbf{2}^{R_2.\text{Länge}}$ + $R_2$.Wert<br>$R_1$.Länge = $R_2$.Länge + **1** |
| R → Z | R.Wert = Z.Wert , R.Länge = **1** |
| Z → 0 | Z.Wert = **0** |
| Z → 1 | Z.Wert = **1** |



## Diese Gleichungen fügen wir nun an den Ableitungsbaum an



Diese Gleichungen fügen wir nun an den Ableitungsbaum an






Oft ist ein spezielles Attribut ausgezeichnet, welches die Bedeutung des abgeleiteten Wortes ist. Im Beispiel ist dies S.Wert .

Im Beispiel wurden die Attribute stets von den Blättern aufwärts zur Wurzel hin berechnet (bottom up). Dies muss aber nicht sein. Die Attribute können auch von oben nach unten (top down) oder sowohl aufwärts wie auch abwärts ermittelt werden.
Ein Beispiel für eine nicht unmittelbar durchschaubare Auswertungsreihenfolge zeigt die nächste Folie mit der gleichen Beispielgrammatik $G_1$, aber anderen Attributen.



*2.3.5 Hinweis:* Die attributierte Grammatik in 2.3.4 enthält einen Fehler, den Herr D. Lippold entdeckte. Versuchen Sie, ihn durch Inspektion der attributierten Grammatik zu finden.

Falls dies nicht gelingt, dann berechnen Sie S.Wert für das Wort 011.100 . Nun müssten Sie den Fehler entdeckt haben.

*Aufgabe:* Überlegen Sie sich, wie die attributierte Grammatik geändert werden kann, um eine fehlerfreie Bedeutung zu erreichen.

(Wir korrigieren den Fehler hier bewusst nicht, weil das Lernen in der Informatik am nachhaltigsten durch das Begehen und Aufspüren von Fehlern geschieht. Im folgenden Beispiel wurde der Fehler vermieden. Klären Sie, wie dies erreicht wurde, und lösen Sie dann ggf. die Aufgabe.)

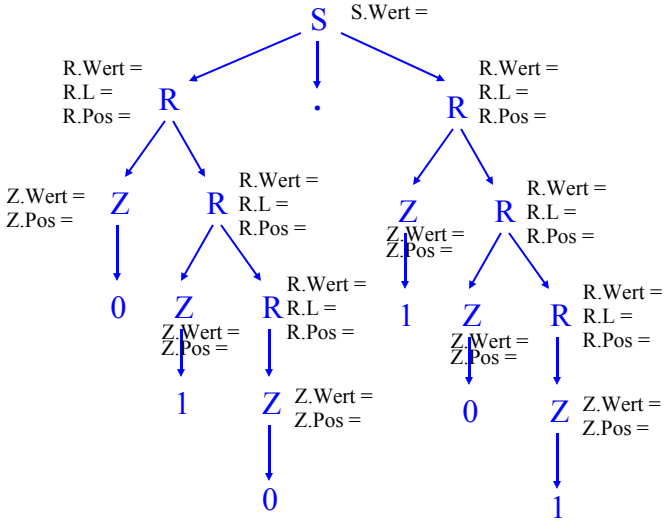


## 2.3.6 Andere Attribute mit anderer Auswertungsreihenfolge

Wir verwenden die Attribute Wert, Pos (für "Position") und L (für "Länge"), und zwar S.Wert, R.Wert, R.L, R.Pos, Z.Wert und Z.Pos.

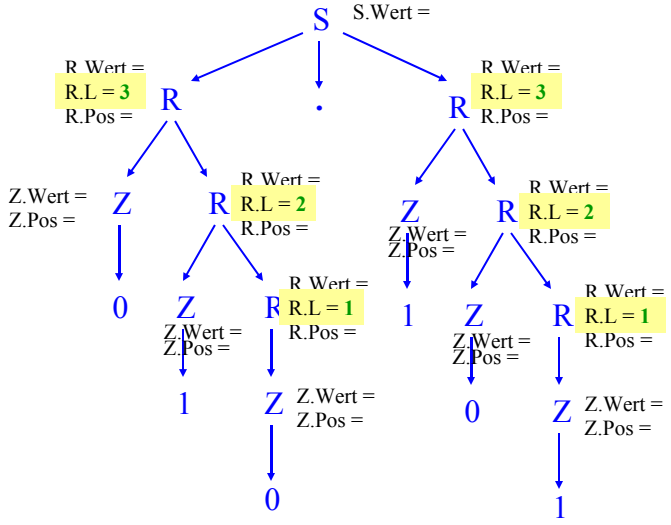
| Produktion | Gleichungen |
|---|---|
| $S \to R$ | $S.Wert = R.Wert$<br>$R.Pos = R.L - 1$ |
| $S \to R_1.R_2$ | $S.Wert = R_1.Wert + R_2.Wert$<br>$R_1.Pos = R_1.L - 1$<br>$R_2.Pos = -1$ |
| $R_1 \to ZR_2$ | $R_1.Wert = Z.Wert + R_2.Wert$<br>$Z.Pos = R_1.Pos$<br>$R_2.Pos = R_1.Pos - 1$<br>$R_1.L = R_2.L + 1$ |
| $R \to Z$ | $R.Wert = Z.Wert$<br>$Z.Pos = R.Pos$<br>$R.L = 1$ |
| $Z \to 0$ | $Z.Wert = 0$ |
| $Z \to 1$ | $Z.Wert = 2^{Z.Pos}$ |

Diese Gleichungen fügen wir erneut an den Ableitungsbaum an (hier nur die linken Seiten)



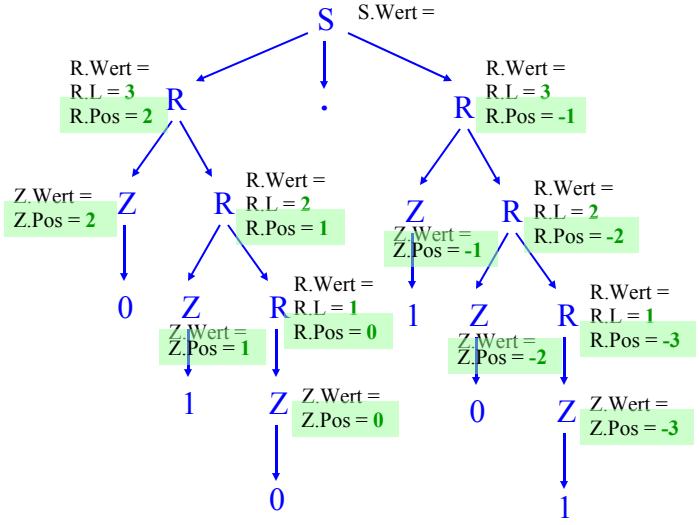



Nun berechnen wir die Attribute. Man sieht: Zunächst lässt sich die Länge L von unten nach oben berechnen:



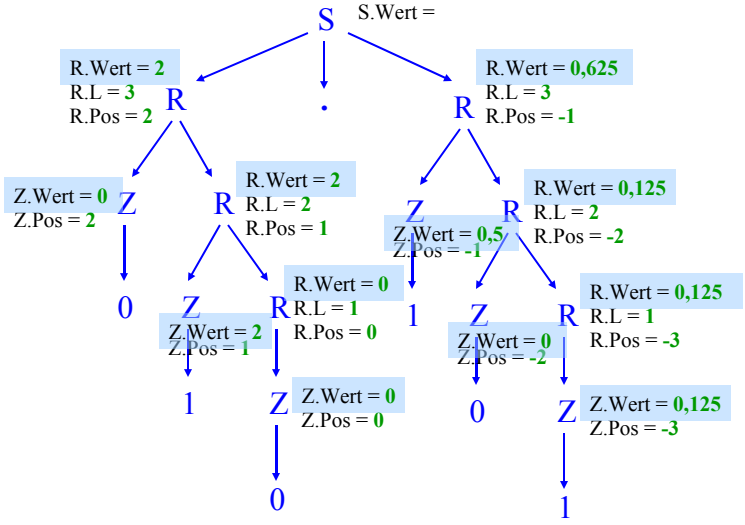



Nun kann man die Position Pos von oben nach unten berechnen:

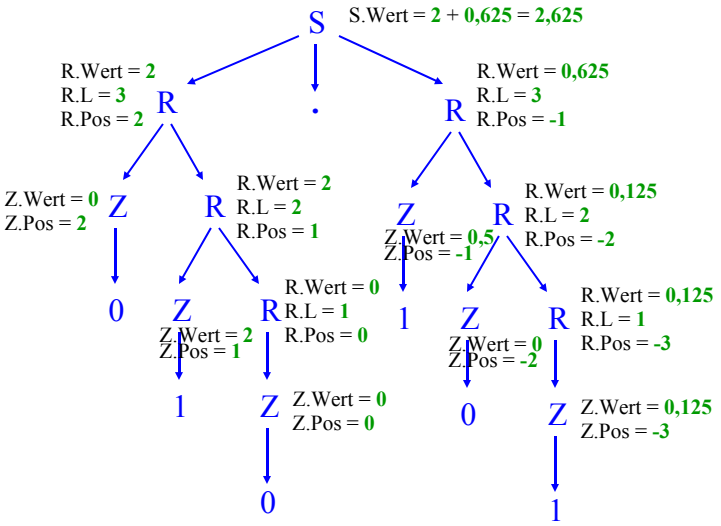





Schließlich kann man den "Wert" von unten nach oben berechnen:



Ergebnis: S.Wert = 2,625






## Definition 2.3.7: Attributierte Grammatik (Knuth, 1968)

Eine kontextfreie Grammatik G = (N; T; P; S) heißt attributierte Grammatik, wenn jedem Zeichen $Z \in N \cup T$ eine Menge $\hat{\mathbf{A}}(Z)$ (= Menge der Attribute von X), jedem Attribut $a \in \hat{\mathbf{A}}(Z)$ eine (Werte-) Menge $W_a$ und jeder Produktion $X \to Y_1 Y_2 ... Y_m \in P$ eine Menge von Abbildungen mit folgenden Eigenschaften zugeordnet ist.

(1) Jedes $\hat{\mathbf{A}}(Z)$ lässt sich in zwei disjunkte Teilmengen zerlegen:
$\hat{\mathbf{A}}(Z) = \hat{\mathbf{A}}_0(Z) \cup \hat{\mathbf{A}}_1(Z)$ mit $\hat{\mathbf{A}}_0(Z) \cap \hat{\mathbf{A}}_1(Z) = \emptyset$.

(2) Für jedes $a \in \hat{\mathbf{A}}_0(X)$ existiert genau eine Abbildung
$f_a^X: W_{a_1} \times W_{a_2} \times ... \times W_{a_r} \to W_a \quad (r \geq 0)$ und

für jedes i = 1, 2, ..., m und jedes $a \in \hat{\mathbf{A}}_1(Y_i)$ existiert genau eine Abbildung $f_a^{Y_i}: W_{a_1} \times W_{a_2} \times ... \times W_{a_r} \to W_a \quad (r \geq 0)$

mit $a_1, ..., a_r \in \hat{\mathbf{A}}(X) \cup \hat{\mathbf{A}}(Y_1) \cup \hat{\mathbf{A}}(Y_2) \cup ... \cup \hat{\mathbf{A}}(Y_m)$, d. h. dies sind nur Attribute von Zeichen, die in der Produktion vorkommen.
($f_a^X$ und $f_a^{Y_i}$ werden meist als Gleichungen geschrieben; diese werden von rechts nach links ausgewertet.)



Die Attribute aus $\hat{\mathbf{A}}_0(Z)$ bezeichnet man als zusammengesetztes oder synthetisches oder synthetisiertes Attribut (englisch: synthesized).

Die Attribute aus $\hat{\mathbf{A}}_1(Z)$ bezeichnet man als ererbtes oder vererbtes oder inherites Attribut (englisch: inherited).

Zu einem Wort w berechnet man den Syntaxbaum $S \Rightarrow^* w$, ordnet den Zeichen im Baum ihre Attribute zu und rechnet diese mit Hilfe der Regeln aus. Um genau eine Bedeutung zu erhalten, sollte die Grammatik "eindeutig" sein (siehe Grundvorlesung Definition 2.7.11), d.h., zu jedem ableitbaren Wort darf es nur genau einen Syntaxbaum geben.

Im Allgemeinen zeichnet man ein Attribut des Startsymbols S aus, das die Bedeutung des abgeleiteten Wortes angibt.



## 2.3.8 Problem: Gibt es stets eine eindeutige Auswertung der Attribute? (Genaueres: Vorlesung über Compilerbau)

Im Allgemeinen nein. Es können zyklische Definitionen auftreten. Zum Beispiel schon bei einer Produktion:

$S \to XY$ $\quad$ S.L = Y.Pos + X.L
Y.Pos = S.L

Eine attributierte Grammatik heißt "zulässig" (oder "wohldefiniert"), wenn sich alle Attribute für jeden Syntaxbaum $S \Rightarrow^* w$ stets auswerten lassen.

Zwar ist das Problem, ob eine kontextfreie Grammatik eindeutig ist, algorithmisch nicht entscheidbar, jedoch kann man die Eigenschaft der Zulässigkeit entscheiden, wie man sich folgendermaßen klar machen kann.

*Vorgehensmodell* hierfür: Gegeben ein Wort w der Sprache. Konstruiere hierzu einen Syntaxbaum S $\Rightarrow^*$ w. An jedes Zeichen Z im Baum füge man die Attribute $a \in \hat{A}(Z)$ an. Sodann geben die jeweiligen Produktionen an, wie die Attribute zu berechnen sind. Nun konstruiere man einen Graphen, der die Paare

(Knoten des Syntaxbaums, zugehöriges Attribut)

als Knoten besitzt. Man ziehe eine Kante von (X,a) nach (Y,b) genau dann, wenn (auf Grund der hier benutzten Produktion) für die Berechnung des Wertes des Attributs $a \in \hat{A}(X)$ das Attribut $b \in \hat{A}(Y)$ benötigt wird.

Die Attributberechnung ist in diesem Syntaxbaum genau dann möglich, wenn dieser Graph azyklisch ist und jeder Knoten, der keine einlaufende Kante besitzt, auf Grund der Regeln eine Konstante als Wert zugewiesen bekommt.

Nun muss man sich noch überlegen, dass man mit endlich vielen Wörtern w für jede Grammatik auskommt (nämlich Wörter, deren Syntaxbäume keine Teil-Wiederholungen enthalten, vgl. Theorie-Vorlesungen). Folglich kann man das Problem prinzipiell entscheiden.



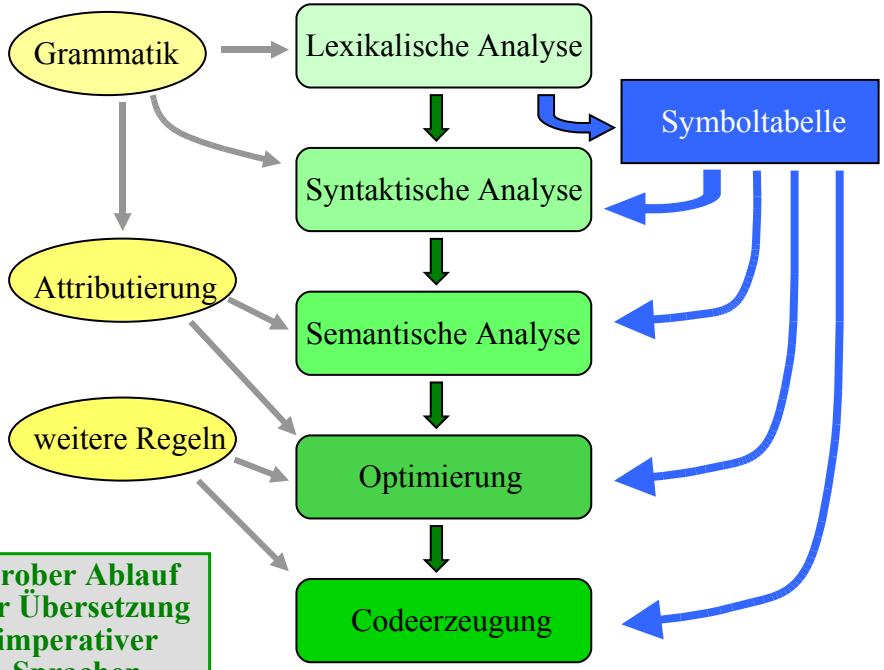
## 2.4 Prinzip der Übersetzung

2.4.1 Allgemeines zur Übersetzung von Sprachen

Wir haben in Abschnitt 2.1 gesehen, dass man gewisse Teile von Ada in den Machinencode einer Register- oder einer Stackmaschine übertragen kann. Man überlegt sich, dass dies prinzipiell für die gesamte Sprache Ada, aber auch für jede andere imperative Programmiersprache gilt.

Die Syntax einer Programmiersprache enthält gewisse Teile, die man vorab leicht identifizieren kann (sog. reguläre Anteile, die man mit einem "endlichen Automaten" erkennen kann). Dies sind vor allem die "lexikalischen Einheiten", siehe Abschnitt 3.2 der Grundvorlesung. Dabei kann man zugleich die verwendeten Identifikatoren (Namen, Konstanten, strings, Wortsymbole) in eine besondere Tabelle, die sog. Symboltabelle, schreiben. Der grobe Ablauf bei der Übersetzung lautet:






**Grober Ablauf der Übersetzung imperativer Sprachen**



2.4.2 Was schreibt man in die Symboltabelle?

Grundsätzlich werden hier alle Symbole, die der Programmierer selbst eingeführt hat, notiert, insbesondere die Typen und die Variablen; zugleich legt man fest, wie diese Symbole auf die "Zielmaschine", also in deren Maschinencode, abgebildet werden. Bei einer Variablen gibt man z. B. an, in oder ab welcher Rechenspeicherzelle sie stehen und wie viele Byte (oder Speicherwörter) sie belegen soll. Bei einem Datentyp gibt man an, wie viele Speicherplätze man insgesamt benötigt, an welchen Stellen hierbei die einzelnen Selektoren stehen und welche elementaren Datentypen jeweils vorliegen (dies braucht man zum Beispiel, um bei der Übersetzung zu prüfen, ob Operatoren in Ausdrücken richtig verwendet werden). An einem Beispiel skizzieren wir kurz das Konzept, das in jedem Compiler in irgendeiner Variante benutzt wird.

## 2.4.3 Beispielprogramm

```
procedure BSP is
type RAT is record Z: Integer; N: Natural; end record;
type XX is record N: array (1..20) of Character;
     L: Float; G: Boolean; R: RAT; end record;
A,B,C: RAT;  Y: XX;
begin
     GET(A.Z); GET(A.N); B:=A;
     C.Z:=A.Z*(A.N+B.Z)-4; C.N:=B.N;
     if A.N > 3 then Y.N(2) := 'E'; else Y.G:= true; end if;
     PUT (A.N);
end;
```



## 2.4.4 Die lexikalische Analyse ermittelt die lexikalischen Einheiten und legt die Bedeutung von Bezeichnern und deren Zuordnung zum Speicher in der Symboltabelle ab.

Laut 3.2.2 sind dies in unserem Beispiel:

*Bezeichner:* BSP, RAT, RAT.Z, RAT.N, XX, XX.N, XX.L, XX.G, XX.R, A, B, C, Y, GET, PUT

*Literale:* 4, 3, 'E', true

*Begrenzer:* : ; .. ( ) := * + - >

*Reservierte Wörter:* procedure is type record end array of begin if then else end

Hinzu kommen Trennzeichen und Kommentare, die beim ersten Lesen des Programmtextes unmittelbar durch interne Darstellungen ersetzt oder überlesen werden. Das Ergebnis der lexikalischen Analyse könnte sein:



| Nr. | Identifikator | Art | Typ | Beginn R[..] | Länge in Byte | Bezug (Nr) | ... ... |
|---|---|---|---|---|---|---|---|
| 1 | BSP | PROC | - | 0 | ? | - | |
| 2 | RAT | Typ | - | - | 8 | - | |
| 3 | RAT.Z | Selektor | Integer | 0 | 4 | 2 | |
| 4 | RAT.N | Selektor | Natural | 4 | 4 | 2 | |
| 5 | XX | Typ | - | - | 34 | - | |
| 6 | XX.N | Selektor | array | 0 | 20 | 5 | |
| 7 | XX.N( ) | Typ | Char | - | 1 | 6 | |
| 8 | XX.L | Typ | Float | - | 4 | 5 | |
| 9 | XX.G | Typ | Bool | - | 1 | 5 | |
| 10 | XX.R | Typ | RAT | - | 8 | 5 | |
| 11 | A | Var | RAT | 40 | 8 | 2 | |
| 12 | B | Var | RAT | 48 | 8 | 2 | |
| 13 | C | Var | RAT | 56 | 8 | 2 | |
| 14 | Y | Var | XX | 64 | 34 | 5 | |
| 15 | GET | FKT | - | - | - | - | |
| 16 | 4 | Konst | Natural | - | 4 | - | |
| 17 | 3 | Konst | Natural | - | 4 | - | |
| 18 | E | Konst | Char | - | 1 | - | |
| 19 | true | Konst | Bool | - | 1 | - | |
| 20 | PUT | FKT | - | - | - | - | |



"-" ist als "nicht zutreffend" zu lesen.
"?" bedeutet, dass die Eintragung erst später erfolgt.
Wir nehmen hier an, dass die Variablen ab Rechenspeicherzelle 40 stehen und in der Reihenfolge, wie sie im Programm aufgeführt sind, abgespeichert werden. Die Konstanten c fassen wir wie Variablen auf, die aber keinen eigenen Rechenspeicherplatz erhalten, sondern direkt aus der Symboltabelle in die jeweiligen Anweisungen LOAD V,c später eingesetzt werden.

Machen Sie sich klar. dass man diese Tabelle mit nur einem Durchlauf durch das Programm aufstellen kann. Natürlich muss in einer "richtigen Symboltabelle" noch mehr stehen. Z. B. ist anzugeben, in welchem Block man sich befindet, wie der Indextyp bei Feldern lautet, an welchen Stellen in der Bibliothek man die global definierten Funktionen (GET, PUT, ...) findet, usw. Dies hängt von der zu übersetzenden Programmiersprache ab.

## 2.4.5 Die weiteren Phasen

Die syntaktische Analyse ermittelt aus dem Programm den Syntaxbaum. Hierzu gibt es Standardalgorithmen für beliebige kontextfreie Grammatiken, die allerdings eine Laufzeit von $O(n^3)$ besitzen, wobei n die Länge des Programmtextes ist. Dies dauert aber für die Praxis zu lange.

Man schränkt daher die kontextfreien Grammatiken auf solche eindeutige Grammatiken ein, deren Syntaxanalyse nur lineare Zeit erfordert (Stichwörter: LL, LR(1), LALR(k)).

Der Syntaxbaum wird anschließend mit Methoden, wie sie in Abschnitt 2.3 besprochen wurden, in ein "Zielprogramm" übersetzt. Während dieser Phase und/oder anschließend wird der übersetzte Code optimiert (z.B. Entfernen überflüssiger Befehle, bessere Nutzung der Register, Ersetzen durch eine kürzere gleichwertige Befehlsfolge).



Das übersetzte Programm in den Code der Stackmaschine aus Abschnitt 2.2 würde (ohne Optimierung) also lauten:

```
 0: IN (A.Z);          1: IN (A.N);
 2: ReadStack A.Z;     3: WriteStack B.Z;
 4: ReadStack A.N;     5: WriteStack B.N;
 6: ReadStack A.Z;     7: ReadStack A.N;
 8: ReadStack B.Z;     9: addStack;
10: multStack;        11: LoadStack 4;
12: subStack;         13: WriteStack C.Z;
14: ReadStack B.N;    15: WriteStack C.N;
16: ReadStack A(N);   17: LoadStack 3;
18: compStack (<=);   19: jump 24;
20: LoadStack 'E';    21: WriteStack Y.N(2);
22: Load F,1;         23: jump 26;
24: LoadStack true;   25: WriteStack Y.G;
26: OUT (A.N);        27: stop
```

IN und OUT ist die maschinenabhängige Ein-/Ausgabe. Die Adressen der Variablen entnimmt man der Symboltabelle und setzt sie hier ein.



## 2.4.6 Compiler-Compiler

Die Ziele dieser theoretischen Überlegungen sind zum einen Techniken, um einen Compiler für eine konkrete Programmiersprache zu schreiben, und zum anderen Methoden, um einen solchen Compiler automatisch aus einer attributierten Grammatik erzeugen zu lassen. Ein Programm, das dieses leistet, nennt man Compiler-Compiler.

Hierzu muss man eine eindeutige kontextfreie Grammatik für die z uübersetzende Programmiersprache zugrunde legen, die eine rasche Syntaxanalyse erlaubt. Diese muss man mit Attributen versehen, deren Auswertung den übersetzten Code eines Programms liefert. An folgendem Beispiel für arithmetische Ausdrücke wird dies klar. Es gibt hier nur ein Attribut "Code".



| Produktion | Gleichungen |
|---|---|
| $W \rightarrow V := E$ | W.Code = **WriteStack** V.Code **;** |
| $E_1 \rightarrow T + E_2$ | $E_1$.Code = T.Code $E_2$.Code **addStack ;** |
| $E \rightarrow T$ | E.Code = T.Code |
| $T_1 \rightarrow F * T_2$ | $T_1$.Code = F.Code $T_2$.Code **multStack ;** |
| $T \rightarrow F$ | T.Code = F.Code |
| $F \rightarrow (E)$ | F.Code = E.Code |
| $F \rightarrow B$ | F.Code = **LoadStack** B.Code **;** |
| $F \rightarrow V$ | F.Code = **ReadStack** V.Code **;** |

Die Terminalzeichen sind fett grün dargestellt. In den Produktionen steht V für eine Variable und B für eine binär dargestellte natürliche Zahl. Negative Zahlen kann man durch Hinzunahme eines "Komplementbefehls Negiere oberstes Stackelement" oder durch "0 minus Stackelement" behandeln. Das Attribut W.Code liefert dann das übersetzte Programm auf einer Stackmaschine.

Man muss die Befehle noch durchnummerieren, über die Symboltabelle die korrekte Verwednung der Operatoren prüfen usw. - aber im Prinzip müsste klar sein, dass man solche Compiler-Compiler bauen kann.

2.4.7 Schlussbemerkung: Sie erkennen an diesem Beispiel auch, wozu man Theorie braucht.

Die Theorie liefert den Formalismus für Grammatiken und ihre Attribute, sie beschreibt unzweideutig die Auswertungen und die Bedeutungen, und sie beweist die Korrektheit und Terminierung.

Dabei entsteht eine Fülle von Erkenntnissen und Spezialisierungen. Zum Beispiel die genannte eindeutige Grammatik, vor allem LALR(1). Aber auch die Attributierungen müssen aus Effizienzgründen beschränkt werden. Eine Attributierung, die nur synthetisierte Attribute besitzt, heißt *S-Attributierung*. In der Praxis genügt die *L-Attributierung*, bei der sich jedes Attribut eines Nichtterminals X durch die Attribute der links davon stehenden Nichtterminale berechnen lässt, usw. usw.



# 3. Funktionales Programmieren

*37 Folien über Softwareentwicklung von D. Lippold*
*43 Folien über Einf. in Scheme von D. Lippold*





## 3.1 Einführung in Scheme

3.1.1 Einführendes Beispiel Wir wollen feststellen, ob der Wert von 27*37+91*11-83*55 durch 3 teilbar ist.

Zunächst müssen wir klären, was "durch 3 teilbar" bedeutet. Hierfür ziehen wir den Rest, der bei der Division bleibt, heran. Im Falle "3" bedeutet dies: Für x = 0, 1, 2 ist x der Rest, der bei der Division durch 3 bleibt. Weiterhin gilt:
x mod 3 = 1 ⇔ (x-1) mod 3 = 0 ⇔ (x-2) mod 3 = 2 ,
x mod 3 = 2 ⇔ (x-1) mod 3 = 1 ⇔ (x-2) mod 3 = 0.

Dies ergibt eine Rekursionsformel für die drei Funktionen Rest0, Rest1 und Rest2, die den Wahrheitswert liefern, ob der Rest bei der Division durch 3 gleich 0, 1 oder 2 ist. Also:

Rest0(x) = if x = 0 then true else Rest2(x-1) fi;
Rest1(x) = if x = 0 then false else Rest0(x-1) fi;
Rest2(x) = if x = 0 then false else Rest1(x-1) fi;



Rest0(x) = if x = 0 then true else Rest2(x-1) fi;
Rest1(x) = if x = 0 then false else Rest0(x-1) fi;
Rest2(x) = if x = 0 then false else Rest10(x-1) fi;

Dies liefert bereits das Programm in der Sprache Scheme:

```
(define (Rest0 x) (if (= x 0) "ja" (Rest2 (- x 1))))
(define (Rest1 x) (if (= x 0) "nein" (Rest0 (- x 1))))
(define (Rest2 x) (if (= x 0) "nein" (Rest1 (- x 1))))
(Rest0 ( - (+ (* 27 37) (* 91 11)) (* 13 55)))
```

Was fällt auf?

Preorder Darstellung der Operatoren; viele Klammern; rekursive Definitionen erlaubt; Liste von Objekten, wobei das erste Objekt als Operator aufgefasst wird, der auf die restlichen Objekte angewendet wird.

*Hinweis:* Die Sprache Scheme wurde als ein einfacher LISP-Dialekt ("properly tail-recursive") von G.L.Steele jr. und G.J.Sussmann am MIT entwickelt und implementiert.

Die Sprache Scheme ist 1998 standardisiert worden. Den

Revised[5] Report on the Algorithmic Language Scheme

finden Sie im Netz zum Beispiel über die Adresse

http://www-swiss.ai.mit.edu/projects/scheme

Alle folgenden Ausführungen können somit bzgl. Syntax und deren Bedeutung nachvollzogen werden. Zugleich finden Sie dort weitere Sprachelemente, die wir in unserer Vorlesung nicht behandeln.

Der 6. Revised Report on Scheme soll 2007 erscheinen.

Eine Kurzübersicht zu Scheme findet sich unter Wikipedia. (Besser ist natürlich ein Lehrbuch.)



## 3.1.2 (Elementare) Datentypen

[Erinnerung: Elementare Datentypen in Ada sind Boolean, Character, Integer, Float und Aufzählungstypen, wobei zum Wertebereich stets auch die zulässigen Operationen gehören. Der Typ einer (Informatik-) Variablen wird bei der Deklaration festgelegt, er braucht daher später in Ada nicht abgefragt zu werden.]

In der Sprache Scheme werden die Identifikatoren/Bezeichner/Variablen zunächst im mathematischen Sinne (also als "formale Parameter") aufgefasst. Sie stehen für einen Wert. Der Typ dieses Wertes liegt nicht von vornherein fest. Daher muss man im Programm abfragen können, von welchem Typ der Wert eines Bezeichners ist. Dies erfolgt durch Typ-Prädikate.

In Scheme werden vor allem Listen und die folgenden Typen verwendet.



Höchstens genau einer der folgenden Typen sind für ein Objekt zutreffend.

| *Typprädikat* | *Datentyp* |
|---|---|
| number? | eine Zahl (zunächst wird nicht zwischen reellen und ganzen Zahlen unterschieden) |
| boolean? | Boolescher Wert ( #f für "false", #t für "true") |
| char? | alphanumerisches Zeichen (mit #\ beginnend) |
| symbol? | Objekt mit einem Namen/Bezeichner zur Identifikation |
| pair? | Paar (zwei Objekte, wie man sie mittels cons bildet, auch eine Liste ist ein Paar) |
| string? | Zeichenkette (auch leer) |
| vector? | Vektor (= eindimensionales Feld, mit #( beginnend) |
| procedure? | Prozedur |
| port? | Ein-/Ausgabe-Einheiten |



Wie sehen die Konstanten in Scheme aus?

Zahlen (number): Man gibt eine Folge von Ziffern an, wenn man eine natürliche Zahl meint; diese kann eine Vorzeichen besitzen, wenn man eine ganze Zahl meint; diese kann einen gebrochenen Anteil besitzen (abgetrennt durch einen Punkt), wenn man eine reelle Zahl meint.
Beispiele: 34, -651, 54.1204, -23.65. In Scheme kann man auch rationale und komplexe Zahlen verwenden.

Wahrheitswerte (boolean): Für false schreibt man #f und für true #t.

Zeichen (char) werden in der Form #\a (Kreuz,Backslash,Zeichen) dargestellt. (Eine Folge von Zeichen ist ein string, s. u.)

Symbole: Dies sind Objekte, die keine Konstanten sind, im Programm verwendet werden und durch einen Bezeichner identifiziert werden können.
Als Bezeichner sind alle nichtleeren Folgen von Buchstaben, Ziffern und folgenden Zeichen

! $ & % + - * / . : < = > ? @ ^ _ ~

zugelassen, deren erstes Zeichen nicht mit dem Anfang einer Zahl verwechselt werden kann. Beispiele für Bezeichner:
x ?a2 =<_als%er v123-aber-nicht-negativ x-->y
Mittels (define x <Ausdruck>) wird dem Bezeichner x ein Wert zugeordnet.

Prozeduren (= Funktionen) bestehen aus einem Namen, einer Folge formaler Parameter und einem definierenden Ausdruck (dem Rumpf). Wir führen Prozeduren in einem Programm in der Regel ein mittels (s. u.)
(define <Name und Parameter> <Prozedurrumpf>)



Allgemein erhält man Funktionen aus Ausdrücken durch den Lambda-Operator. Meint man z.B. nicht den Ausdruck (+ (* x x) x), sondern die Funktion $f(x) = x^2+x$, so schreibt man

(lambda (x) (+ (* x x) x))

Allgemein:

(lambda <Formalteil> <Rumpf>)

Im einfachsten Fall ist der Formalteil die Liste der formalen Parameter (eventuell leer) und der Rumpf ein Ausdruck. Will man der Funktion einen Namen geben, so muss man dies mittels "define" tun:

(define h (lambda (x) (+ (* x x) x)))

Der Prozeduraufruf (procedure call) liefert dann z.B.:

(h 5) ==> 30



Anstelle von
(define <Bezeichner> (lambda <Formalteil> <Rumpf>))

schreibt man kürzer

(define <Bezeichner und Formalteil> <Rumpf>)

Beispiel: Statt

(define h (lambda (x) (+ (* x x) x)))

kann man daher auch schreiben

(define (h x) (+ (* x x) x))

(h 5) liefert erneut den Wert 30.



## 3.1.3 Ausdrücke

Ausdrücke werden aus Konstanten, Variablen (Bezeichner), Operatoren und Funktionsaufrufen (procedure calls) in preorder Darstellung gebildet. Durch Klammern wird die für die Auswertung wichtige Baumstruktur festgelegt. Dies klingt zunächst unsinnig, weil die Preorder-Darstellung ja die eindeutige Reihenfolge festlegt, jedoch ist in Scheme die Anzahl der Argumente einer Funktion oft nicht fest, und zugleich dient die Klammerung der Zusammenfassung zusammengehöriger Teile des Ausdrucks. Weiterhin ist der Begriff "Ausdruck" nicht auf Zahlbereiche beschränkt (s. u.).

Ausdrücke sind einzelne Variablen oder Konstante oder sie besitzen eine Listenstruktur.

*Beispiel:* Ganze Zahlen mit den Operationen succ, pred, +, -, *, /, sign, =, <, > , /=, <=, >=. Hiermit kann man unmittelbar Ausdrücke (Terme) bilden: (23+18)*(4-2), jedoch muss man in Scheme die preorder-Darstellung verwenden, also

(* (+ 23 18) (- 4 2))

Dieser Term lässt sich als eine vierstellige Funktion fk auffassen: (define (fk a b c d) (* (+ a b) (- c d)) ).
Auch diese Liste ist ein Ausdruck, nämlich ein Ausdruck, der den Bezeichner fk definiert und der seine Stelligkeit und die Berechnungsvorschrift festlegt.

Der Begriff "Ausdruck" (expression) wird also nicht nur für arithmetische und boolesche Ausdrücke benutzt, sondern für alle Darstellungen, also auch für Kontrollelemente, Funktionen und Programme. Letztlich ist ein Scheme-Programm eine Menge von Definitions-Ausdrücken mit einem oder mehreren Ausdrücken, die zum Ergebnis des Programms ausgewertet werden können.



## 3.1.4 Auswertung und die Funktion quote

Ausdrücke werden ausgewertet, indem man sie ausrechnet. Das Ergebnis der Auswertung benennen wir durch "==>".
Beispiel: (+ 4 5) ==> 9
Der Pfeil ==> ist also zu lesen als *"wird ausgewertet zu"*.

Konstanten werden immer zu sich selbst ausgewertet:

```
27      ==> 27
#f      ==> #f
```

Ein Bezeichner wird zu dem ihm zugeordneten Wert ausgewertet:

```
(define x 6)  x    ==> 6
(define (f x) (* x x))  f  ==> #<procedure:f>
(define (q x) (if (#t + -))  q  ==> +
```

Hat ein Bezeichner keinen Wert, so liegt ein Fehler vor.



Ein Lambda-Ausdruck kann ausgewertet werden, indem er auf so viele Argumente, wie die Liste seiner formalen Parameter angibt, angewendet wird:

( **(**lambda (x) (+ (* x x) x)**)** 5) ==> 30

Allgemein bildet man also eine Liste aus dem Lambda-Ausdruck (oder dem Namen einer Funktion) und k aktuellen Parametern, wobei die Funktion k formale Parameter besitzen muss:

(<Lambda-Ausdruck> <$param_1$> ... <$param_k$>)

Beispiel für eine Funktion zum Prüfen, ob $x^2+y^2=z^2$ ist:

```
(define (Pyth x y z) (= (+ (* x x) (* y y)) (* z z)))
(Pyth 3 4 5)    ==> #t
```



Generell muss man stets zwischen den Objekten und dem, was sie bedeuten (also dem Ergebnis einer Auswertung), unterscheiden. Das Objekt selbst erhält man mittels quote.

Meint man zum Beispiel den Ausdruck (+ 4 5) selbst und nicht das Ergebnis 9, so schreibt man

(quote (+ 4 5)) ==> (+ 4 5)

Statt (quote <Objekt>) schreibt man kurz '<Objekt>

```
'(+ 4 5)    ==>   (+ 4 5)
' '(+ 4 5)  ==>   '(+ 4 5)
```

3.1.5 Datenstrukturen erhalten wir, indem wir auf Datentypen Konstruktoren anwenden. In Ada haben wir array, record, access und die Unterbereichsbildung a..b betrachtet.

In Scheme gibt es im Wesentlichen nur einen Konstruktor: die Paarbildung. Hieraus werden Listen aufgebaut. In ihnen werden Elemente sequentiell aneinander gereiht und mit der leeren Liste abgeschlossen. Da die Elemente einer Liste selbst wieder Listen sein können, ergibt sich eine große Vielfalt.

Paar: Zusammenfassung zweier Objekte mit Hilfe des Operators cons: (cons <objekt1> <objekt2>). Paare bezeichnet man auch als "dotted pair"; im Ausdruck erscheint zwischen den beiden Objekten dann auch ein Punkt. Auf das erste Objekt eines Paars kann mittels (car ...) und auf das zweite mittels (cdr ...) zugegriffen werden.



Listen sind Folgen von Objekten, in runde Klammern eingeschlossen und mit der leeren Liste beendet. Man kann sie mittels (list <Folge von Objekten>) erzeugen, z.B.:

```
(list '2 ' 3 '4)                ==>  (2 3 4)
(list '2 (- 5 2) '4)            ==>  (2 3 4)
(list (list '5 'a) '7 (list 'b))  ==>  ((5 a) 7 (b))
```

Man kann Listen auch explizit hinschreiben mittels ' :

```
'(2 3 4)      ==>  (2 3 4)
```

Es gibt diverse Operationen für Listen, z.B. (reverse L) dreht die Reihenfolge der Objekte in der Liste L um.

Ein Spezialfall ist die leere Liste ( ).

*Formale Definition:* Eine Liste ist entweder die leere Liste oder (list $e_1$ $e_2$ $e_3$ ... $e_n$) ist gleichbedeutend mit der (n-1)-maligen wiederholten Anwendung des cons-Operators in der Form:

(cons $e_1$ (cons $e_2$ (cons $e_3$ ...(cons $e_n$ '()) ...))).



Wegen dieser Definition gehört eine nichtleere Liste auch zum Typ "Paar", d. h.

```
(pair? (list 'a))  ==>  #t
```

Dagegen ist die leere Liste kein Paar: (pair? '( )) ===> #f.
Daher führt man einen eigene Abfrage auf die leere Liste ein:

```
null? <Objekt>        Ist <Objekt> die leere Liste?
```

Die leere Liste ist in Scheme eine spezielle Konstante, mit der jede Liste abgeschlossen wird (siehe formale Definition).

Man kann auch abfragen, ob eine Liste vorliegt:

```
list? <Objekt>        Ist <Objekt> eine Liste?
```

Ein Paar ist in der Regel keine Liste, weil es nicht mit der leeren Liste abschließt.



Auf Listen sind die Operatoren car und cdr ebenfalls erlaubt:
(car L) liefert das erste Objekt der Liste L,
(cdr L) liefert die Liste L ohne das erste Element.

In Scheme werden Ausdrücke (außer einzelne Konstanten und Bezeichner) als Listen beschrieben. Eine Liste L

($a_1$ $a_2$ $a_3$ $a_4$)

wird als Ausdruck ausgewertet, indem man das erste Objekt $a_1$ als eine Funktion auffasst, die auf die weiteren Objekte $a_2$ $a_3$ $a_4$ angewendet wird; $a_2$, $a_3$, $a_4$ sind also die aktuellen Parameter für den Operator $a_1$.

Eine Liste L wird als Ausdruck folglich ausgewertet als
(car L) anwenden auf (cdr L).
Listen werden in Abschnitt 3.4 genauer behandelt.

Zeichenketten (string): Eine Folge von beliebigen Tastatur-Zeichen, die in Anführungsstriche eingeschlossen ist, also "es", "Heute ist Dienstag", "" (für den leeren String). Der Backslash \ hat hierbei die Bedeutung einer Unterbrechung. Dies benutzt man, um \ und " in eine Zeichenkette einzufügen, z.B.: den Text

"Mengendifferenz" wird mit "M\N" bezeichnet.

stellt man dar durch

"\"Mengendifferenz\" wird mit \"M\\N\" bezeichnet."

Vektoren: Eine Folge von Elementen, die fortlaufend mit den Indizes 0, 1, 2, ... nummeriert sind. Darstellung in der Form #( <Folge von Elementen> )



## 3.1.6 Die Funktionen apply und eval

(apply <proc> <Ausdruck$_1$> ... <Ausdruck$_m$> <Liste>)
wendet die Funktion <proc> auf die Liste an, die entsteht, indem man <Ausdruck$_1$>, ..., <Ausdruck$_m$> an den Anfang der Liste <Liste> setzt (m = 0 ist erlaubt).

(apply - '(9 4 3)) ==> 2

(apply + 4 (- 6 1) '(3 1))
Dies wird in (apply + '(4 5 3 1)) umgewandelt und dann als (+ 4 5 3 1) ausgewertet, also ergibt sich ==> 13

(apply (if (= 7 8) + *) 4 (- 6 1) '(3 1)) ==> 60



(eval <Ausdruck>) wertet den <Ausdruck> aus.

```
(+ 2 3)                  ==> 5
'(+ 2 3)                 ==> (+ 2 3)
(eval (+ 2 3))           ==> 5
(eval '(+ 2 3))          ==> 5
(eval (eval '(+ 2 3)))   ==> 5
(eval ' '(+ 2 3))        ==> (+ 2 3)
(eval (eval ' '(+ 2 3))) ==> 5
(eval ' ' '(+ 2 3))      ==> '(+ 2 3)
(eval (eval ' ' '(+ 2 3))) ==> (+ 2 3)
(eval ' ' ' '(+ 2 3))    ==> ' '(+ 2 3)
```



```
(define (von-1-bis-x  x)
     (if (= 0 x) ( ) (cons x (von-1-bis-x (- x 1)))))

(define (summenformel a) (cons + (von-1-bis-x a)))

(summenformel 9)              ==> (+ 9 8 7 6 5 4 3 2 1)
```

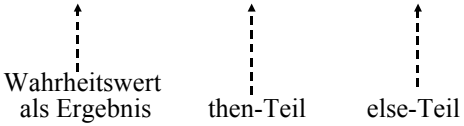
[Hinweis: DrScheme liefert hier die externe Darstellung: (#<primitive:+> 9 8 7 6 5 4 3 2 1), d.h., "+" ist ein Symbol, kein Zeichen]

Dies ergibt nur die Liste. Nun möchte man die Summe auch wirklich ausrechnen, z.B. mittels apply oder eval:

```
(apply + (von-1-bis-x  0))      ==> 0
(apply + (von-1-bis-x  10))     ==> 55
(eval (summenformel 2))         ==> 3
(eval (summenformel 9999))      ==> 49995000
```

## 3.1.7 Alternativen

*Syntax:* (if <Ausdruck> <Ausdruck> <Ausdruck>)




*Beispiele:*

```
(if (< 3 7) 'x 'y)              ===>  x
((if (= 3 5) + -) '6 '2))       ===>  4
((if (= 3 3) + -) '6 '2))       ===>  8
```

Man kann also auch Funktionen auf diese Weise auswählen.

```
(define (fak n) (if (<= n 0) 1 (* n (fak (- n 1)))))
(fak 8)                                  ===>  40320
```



## 3.1.8 Konditionen (Conditionals)

*Syntax:*

(cond <Klausel$_1$> ... <Klausel$_k$>)

Jede Klausel ist von der Form:

(<Bedingung> <Ausdruck$_1$> ...)

oder von der Form

(<Bedingung> => <Ausdruck>).

Die letzte Klausel darf von folgender Form sein:

(else <Ausdruck$_1$> ... >Ausdruck$_m$>)

Bedeutung: Die Klauseln werden der Reihe nach abgearbeitet. Sobald die Bedingung in einer Klausel den Wert true ergibt, wird die Folge der zugehörigen Ausdrücke ausgewertet; danach ist die Kondition beendet. Trifft man auf else, so werden die auf das else folgenden Ausdrücke ausgewertet und danach die Kondition beendet.



## 3.1.9 Hinweis: Probieren Sie alles aus

Manches im Revised Report on Scheme ist beim ersten Lesen nicht recht verständlich. Probieren Sie daher die Bedeutung elementarer Funktionen und insbesondere der Abfragen mit Hilfe des DrScheme-Systems aus. Auf der folgenden Folie sind einige Beispiele abgedruckt einschließlich der Antworten des Systems.

Dabei finden und erlernen Sie zugleich die "Standard-Ein-und Ausgabe", siehe Abschnitt 6.6 des Revised Reports.



| Beispielprogramm | zugehörige Ausgabe |
|---|---|
| (display "list? pair?") (newline)<br>(define q '(7 2 3 6)) q (list? q) (pair? q)<br>(define q (cons 8 4)) q (list? q) (pair? q)<br>(display "boolean?") (newline)<br>'x (boolean? 'x)<br>#t (boolean? #t) | list? pair?<br>(7 2 3 6) #t #t<br>(8 . 4) #f #t<br>boolean?<br>x #f<br>#t #t |
| (display "char?") (newline)<br>4 (char? 4)<br>'a (char? 'a)<br>'4 (char? '4)<br>#\a (char? #\a)<br>#\4 (char? #\4) | char?<br>4 #f<br>a #f<br>4 #f<br>#\a #t<br>#\4 #t |
| #\space (char? #\space)<br>(display "symbol?") (newline)<br>(define x 'S) x (char? x) (symbol? x)<br>(define x #\a) x (char? x) (symbol? x)<br>'z (symbol? 'z)<br>(define z 'y) z (symbol? z) | #\space #t<br>symbol?<br>S #f #t<br>#\a #t #f<br>z #t<br>y #t |
| (display "string? char?") (newline)<br>(define x "Die \"Informatik\" am 30.11.06 (+- eine Woche)")<br>(display x) (newline)<br>(string? x) (char? x)<br>(define y (string-ref x 27)) y<br>(string? y) (char? y) | string? char?<br>Die "Informatik" am 30.11.06<br>(+- eine Woche)<br>#t #f<br>#\6<br>#f #t |

## 3.2 Beispielprogramme

### 3.2.1 Einstiegsprogramme

Ergebnis

```
(define x1 (if (null? '4) 4 (= 4 5)))
x1
```

===> #f

```
(define (hochvier z) (* z (* z z) z))
(hochvier 5)
```

===> 625

In Scheme wird * nacheinander auf beliebig viele Zahlen angewandt, evtl. keinmal (Ergebnis 1) oder nur einmal (Ergebnis z).

```
(define (kubik z) (* z (* z z)))
(define (summe z) (if (null? z)  0
                      (+ (car z) (summe (cdr z)))))
(summe (list (kubik 1)
             (kubik 2) (kubik 3) (kubik 4)))
```

===> 100



### 3.2.2 Zerlegung einer natürlichen Zahl in ihre Primfaktoren

Wie Sie bereits erkannt haben, erfordert es einige Übung, um Scheme-Programme zu lesen. Ohne Erläuterungen ist kaum etwas zu verstehen.

Wir wollen diesen schlechten Stil, Programme einfach ohne Erläuterungen hinzuschreiben, hier weiter pflegen und präsentieren auf der nächsten Folie nur das Ergebnis.

Versuchen Sie, die Korrektheit dieses Programms nachzuvollziehen.

(reverse L) liefert die umgekehrte Reihenfolge einer Liste L.



```
(define  (Primfaktoren a Faktor Faktorliste)
    ( if (> (* Faktor Faktor) a) (cons a Faktorliste)
        ( if (= 0 (remainder a Faktor))
            (Primfaktoren (/ a Faktor) Faktor (cons Faktor
Faktorliste))
            (Primfaktoren a (+ Faktor 1) Faktorliste) ) ) )

(reverse (Primfaktoren 8192  2  '( )))
```

= Funktionskopf

= then-Teil

= if-Bedingungsteil

= else-Teil



### Zerlegung einer Zahl in Primfaktoren

```
(define  (Primfaktoren a Faktor Faktorliste)
    ( if (> (* Faktor Faktor) a) (cons a Faktorliste)
        ( if (= 0 (remainder a Faktor))
            (Primfaktoren (/ a Faktor) Faktor (cons Faktor
Faktorliste))
            (Primfaktoren a (+ Faktor 1) Faktorliste) ) ) )
```

```
(reverse (Primfaktoren 84 2 '( )))          ===>  (2 2 3 7)
(reverse (Primfaktoren 8192 2 '( )))        ===>  (2 2 2 2 2 2 2 2 2 2 2 2 2)
(reverse (Primfaktoren 1443751 2 '( )))     ===>  (103 107 131)
```

Aufgabe 1: Ersetzen Sie in dem obigen Programm zur Primfaktorzerlegung das ">"-Zeichen durch ">=". Ist das Programm dann noch richtig (Beweis?) oder können Sie Beispiele angeben, für das Programm dann fehlerhaft arbeitet?

Aufgabe 2: Hin und wieder benötigt man die Funktion

pr(n) = n-te Primzahl

also pr(1) = 2, pr(2) = 3, pr(3) = 5, ..., pr(10) = 29, ...

Schreiben Sie ein Scheme-Programm, das diese Funktion berechnet. Berechnen Sie hiermit mindestens 5 Werte, davon mindestens zwei mit n > 200.



## 3.2.3 Größter gemeinsamer Teiler (ggT, gcd)

*Definition:* Der größte gemeinsame Teiler d zweier natürlicher Zahlen a und b ist die größte natürliche Zahl, die a und b teilt.

(Hinweis: d teilt a $\Leftrightarrow \exists\, k \in \mathbb{N}: k \cdot d = a$.)

Sei $T(x) = \{y \mid y \text{ teilt } x\}$ die Menge aller Teiler von x, dann erhält man also:

$$ggT(a,b) = \text{Max}\,(T(a) \cap T(b)) \quad \text{für alle } a, b \in \mathbb{N}.$$

Dies ist die Spezifikation, die zum Nachweis der Korrektheit gebraucht wird. Sie wird zunächst implementiert.



Bevor man beginnt, muss man sich überzeugen, dass die Definition sinnvoll und widerspruchsfrei (also "wohldefiniert") ist.

1. Für eine natürliche Zahl a (ohne die Null) ist T(a) eine endliche Menge. Die Zahl 1 liegt stets in T(a).
2. Es folgt: Insbesondere ist $T(a) \cap T(b)$ eine endliche nichtleere Menge natürlicher Zahlen. Diese besitzt stets genau ein Maximum.

Der ggT(a,b) ist somit wohldefiniert.



Wir benötigen die Hilfsfunktion "d teilt a". Diese berechnen wir mittels: d teilt a $\Leftrightarrow$ a mod d = 0.

Hierdurch verlassen wir den bisherigen Zahlbereich $\mathbb{N}$ und arbeiten ab jetzt in $\mathbb{N}_0$, wobei aber die Eingabezahlen a und b nicht Null sein dürfen, da wir den ggT nur auf $\mathbb{N}$ definiert haben. Auch die zwischenzeitlich berechneten Teiler dürfen nicht Null sein.

Die Abfrage, ob a mod = 0 ist, wird in Scheme durch

(= 0 (remainder a d))

dargestellt.

```
;;; Zuerst berechnen wir zu einer Zahl a die Teilermenge.
;;; Hierzu ermitteln wir deren Teiler von einer Zahl i bis a und
;;; setzen die Teilermenge von a auf diese Teiler von 1 bis a.
;;; Diese Menge speichern wir als Liste.

(define (teilermenge-i-bis-x i x)
    (if (> i x) ( )   ; leere Menge, falls i > x
        (if (= 0 (remainder x i))
                    ; falls "i teilt d", dann d in Liste aufnehmen
                    ; in jedem Fall danach mit i+1 weitermachen
            (cons i (teilermenge-i-bis-x (+ i 1) x))
            (teilermenge-i-bis-x (+ i 1) x) ) ) )

(define (teilermenge a) (teilermenge-i-bis-x 1 a))
```



```
;;; Den Schnitt zweier Listen erhält man, indem man für jedes
;;; Element der ersten Liste prüft, ob es in der zweiten enthalten ist.
;;; Wir betrachten daher zunächst die Funktion element?.

(define (element? a L)
    (if (null? L) #f
        (if (= a (car L)) #t (element? a (cdr L)) ) ) )

(define (schnitt L1 L2)
    (if (null? L1) ( )
        (if (element? (car L1) L2)
            (cons (car L1) (schnitt (cdr L1) L2))
            (schnitt (cdr L1) L2) ) ) )
```



```
;;; Nun müssen wir das Maximum einer Liste bestimmen.
;;; Das Maximum einer einelementigen Liste ist das einzige Element
;;; der Liste, anderenfalls vergleiche das erste Element mit dem
;;; Maximum der Restliste.

(define  (eins? L)  (and (not (null? L)) (null? (cdr L))) )

(define  (maximum L)
    (if (eins? L) (car L)
        (if (< (car L) (maximum (cdr L)))
            (maximum (cdr L))  (car L)) ) )
```



```
;;; Nun haben wir alle notwendigen Begriffe eingeführt.
;;; Der ggT lässt sich nun unmittelbar in Scheme "spezifizieren".

(define  (ggT a b)
(maximum (schnitt (teilermenge a) (teilermenge b))) )

:;; Nun kann man Werte berechnen lassen, zum Beispiel:
(ggT 12 66)  (ggT 527 961)  (ggT 782673 969494)
(teilermenge 782673)
(teilermenge 969494)
(schnitt (teilermenge 782673)  (teilermenge 969494))
(cons + (teilermenge 12))          ; Liste aus "+" und teilermenge
(eval (cons + (teilermenge 12)) )  ; Summe aller Teiler von 12
(eval (cons + (teilermenge 720720)))
```

"Gesamte Spezifikation"

```
(define (teilermenge-i-bis-x i x)
    (if (> i x) ( )
        (if (= 0 (remainder x i))
            (cons i (teilermenge-i-bis-x (+ i 1) x))
            (teilermenge-i-bis-x (+ i 1) x) ) ) )
(define (teilermenge a) (teilermenge-i-bis-x 1 a))
(define (element? a L)
    (if (null? L) #f  (if (= a (car L)) #t (element? a (cdr L)) ) ) )
(define (schnitt L1 L2)
    (if (null? L1) ( )
        (if (element? (car L1) L2)
          (cons (car L1) (schnitt (cdr L1) L2))
          (schnitt (cdr L1) L2) ) ) )
(define (eins? L)  (and (not (null? L)) (null? (cdr L))) )
(define (maximum L)
    (if (eins? L) (car L)
        (if (< (car L) (maximum (cdr L))) (maximum (cdr L))  (car L)) ) )
(define  (ggT a b)
    (maximum (schnitt (teilermenge a) (teilermenge b))) )
```

```
(ggT 808038 720720)  (teilermenge 808038)  (teilermenge 720720)
```



Den ggT kennen wir bereits gut:
Der ggT wurde in der Grundvorlesung in Abschnitt 1.6.3 (5) eingeführt, unter 1.7 zweites Beispiel rekursiv definiert und in Beispiel 2.4.3 ausführlich erläutert. In 2.4.4 wurde bewiesen, dass der euklidische Algorithmus den ggT korrekt ermittelt, und in 6.5.2 wurde nachgewiesen, dass die uniforme Zeitkomplexität O(n) und die Zeitkomplexität für die ziffernweise Bearbeitung $O(n^3)$ betragen (letzteres bei Verwendung der "Schulmethode" für die Berechnung des Restes).

Der euklidische Algorithmus euklid in seiner rekursiven Form kann direkt nach Scheme übertragen werden:

```
(define (euklid a b) (if (= b 0) a (euklid b (remainder a b))))
```



*Experimentieren Sie nun:*

Vergleichen Sie die Laufzeiten der beiden Programme ggT und euklid. Stellen Sie fest, für welche Funktionen die Berechnung von ggT die meiste Zeit verbraucht (überrascht?).

Beachten Sie: euklid ist für alle ganzen Zahlen definiert. Geben Sie daher auch negative Zahlen ein. Machen Sie sich die Funktion "remainder" für ganze Zahlen klar.

Geben Sie dann die Funktion euklid als Funktion auf ganzen Zahlen genau an. Zum Beispiel gilt:
(euklid -1 1) ==> 1, (euklid -1 1) ==> -1,
(euklid 4 -4) ==> -4, (euklid 4 -8) ==> 4,
(euklid 4 -5) ==> -1, (euklid 4 -7) ==> 1 usw.



Zeitmessungen mit einem Laptop:

| | |
|---|---|
| (teilermenge 808038) (teilermenge 720720) | (Zeit: < 1 Sekunde) |
| (schnitt (teilermenge 808038) (teilermenge 720720)) | (Zeit: < 1 Sekunde) |
| (ggT 808038 720720) | (Zeit: 16 Sekunden) |

Ergebnis: Um das Maximum der 24-elementigen Schnittmenge zu ermitteln, braucht das Programm wesentlich länger als zur Bildung der beiden Teilermengen und ihres Durchschnitts.

Woran liegt das?

Analysieren Sie die Funktion, die das Maximum berechnet.

## 3.2.4 Die n-te Primzahl

*Aufgabe:* Programmiere die Funktion f(n) = n-te Primzahl.

*Vorgehen:* rekursiv nach der Vorschrift:

a) Die erste Primzahl ist 2.

b) Wenn $p_n$ die n-te Primzahl ist, dann ist $p_{n+1}$ die kleinste Primzahl, die größer als $p_n$ ist.

*Hilfsfunktionen*, die hier offenbar vorkommen:

(prim? x) möge den Wahrheitswert, ob x eine Primzahl ist, liefern.

(nextprim x) möge die kleinste Primzahl, die größer als x ist, liefern.



*Hilfsfunktion* (prim? x)
Beschreibung: x ist genau dann eine Primzahl, wenn x>1 ist und x durch keine der Zahlen 2, 3, ..., x-1 teilbar ist.

*Lösungsansatz 1*: prim1?
x ist genau dann eine Primzahl, wenn die Teilermenge von x zweielementig ist, wobei (teilermenge x) in Beispiel 3.2.3 definiert wurde. Siehe nächste Folie.

*Lösungsansatz 2*: prim2?
Für x>1 muss mindestens eine der Zahlen 2, 3, ..., x die Zahl x teilen. Genau dann, wenn x die kleinste dieser Zahlen ist, ist x eine Primzahl. Siehe übernächste Folie.

*Lösungsansatz 3*: prim3?
Wie Ansatz 2, jedoch nur für 2, 3, ..., Wurzel(x) testen. Details selbst ausführen!



Lösungsvorschlag 1

```
(define (teilermenge-i-bis-x i x)
        (if (> i x)  ( )
                  (if (= 0 (remainder x i))
                      (cons i (teilermenge-i-bis-x (+ i 1) x))
                      (teilermenge-i-bis-x (+ i 1) x) ) ) )
(define (teilermenge a) (teilermenge-i-bis-x 1 a))
(define (zwei? L)
   (cond
      ((null? L) #f)
      ((null? (cdr L)) #f)
      ((null? (cdr (cdr L))) #t)
      (else #f) ) )
(define (prim1? x) (zwei? (teilermenge x)))

(prim1? 1) (prim1? 13) (prim1? 1337) (prim1? 494761)
(teilermenge 362880) (prim1? 362880)
```



Lösungsvorschlag 2

```
(define (kleinster-teiler-i-bis-x  i  x)
        (cond   ((= i x) x)
                ((= 0 (remainder x i)) i)
                (else (kleinster-teiler-i-bis-x  (+ i 1) x))   ) )
(define (prim2? x)
        (and (not (= x 1)) (= x (kleinster-teiler-i-bis-x  2  x))) )
```

;;; Eine Testreihe könnte sein:

```
(prim1? 0)  (prim1? -15)  (prim1? 1)  (prim1? 13)
(prim1? 1337)  (prim1? 494761)  (prim1? 100000001)

(prim2? 0)  (prim2? -15)  (prim2? 1)  (prim2? 13)
(prim2? 1337)  (prim2? 494761)  (prim2? 100000001)
```

;;; Ergebnis: #f #f #f #t #f #t #f #f #f #f #t #f #t #f

*Hilfsfunktion* (nextprim x)
berechnet die kleinste Primzahl, die größer als x ist.

*Lösungsansatz*: nextprim: $\mathbb{N} \to \mathbb{N}$ ist definiert durch
(nextprim x) = x+1, falls x+1 eine Primzahl ist
(nextprim x) = (nextprim (+ x 1)), sonst.

Hieraus folgt:
Die n-te Primzahl $p_n$ erhält man, indem man n-mal nextprim auf die Zahl 1 anwendet.

In der Grundvorlesung (dort 2.5.5) hatten wir solche Funktionen die "Iterierte" genannt. Die Funktion $n \to p_n$ ist also die Iterierte von nextprim.

| x | (nextprim x) |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 5 |
| 4 | 5 |
| 5 | 7 |
| 6 | 7 |
| 7 | 11 |
| 8 | 11 |
| 9 | 11 |



Wie beschreibt man in Scheme die Iterierte einer Funktion?

Formal ist die Iterierte von f die Funktion

(f-x-iter 0 x) = x (die 0-te Iterierte ist die Identität) und

(f-x-iter m x) = $\underbrace{(f\,(f\,(f\,(\,\ldots\,f}_{\text{m-mal}}(x)\,\ldots)\,)\,)\,)$ für m > 0.

Übertragung nach Scheme:

```
(define  (f-x-iter m x)
         (if (= m 0)  x  (f (f-x-iter (- m 1) x)) ) )
```

*Hinweis:* Dies lässt sich aus Sicht der funktionalen Programmierung noch eleganter formulieren, siehe f-iter in Abschnitt 3.2.5.



Hieraus erhält man die gesuchte Funktion (wir verwenden *prim2?*):

```
(define (n-te-Primzahl n) (nextprim-iter n 1))

(define (nextprim-iter m x)
        (if (= m 0)  x  (nextprim (nextprim-iter (- m 1) x)) )  )

(define (nextprim a)
        (if (prim? (+ a 1)) (+ a 1) (nextprim (+ a 1))) )

(define (kleinster-teiler-i-bis-x  i  x)
        (cond   ((= i x) x)
                ((= 0 (remainder x i)) i)
                (else (kleinster-teiler-i-bis-x  (+ i 1) x))   ) )

(define (prim? x)
        (and (not (= x 1)) (= x (kleinster-teiler-i-bis-x  2  x))) )

(n-te-Primzahl 25)     ;;; Ergebnis ist 97
```



Hinweise zu den Beispielen:

Beispiel 3.2.2 ist unkommentiert; man kann daher kaum nachvollziehen, ob es korrekt arbeitet.

Die Spezifikation in Beispiel 3.2.3 ist aufwändig zu programmieren, während der schnelle euklidische Algorithmus sehr einfach ist. Dass beide Verfahren auf den natürlichen Zahlen (ohne 0) die gleiche Funktion berechnen, hatten wir bereits in der Grundvorlesung bewiesen. Was euklid auf den ganzen Zahlen genau berechnet, bleibt undurchsichtig.

Die Spezifikation in 3.2.4 ist einfach (sofern man Primzahlen kennt). Das Programm zur Berechnung ist dagegen deutlich aufwändiger.

Die Zuverlässigkeit ist immer gefährdet, wenn man den vorgegebenen Wertebereich verlässt und dies auch für Eingabewerte erlaubt.

Der Zeitaufwand ist zunächst nicht unmittelbar verständlich. Der Grund liegt in der "Baum-Rekursion" der Funktion maximum. Hier werden bereits berechnete Werte immer wieder neu berechnet. Mit Hilfe einer lokalen Zwischenspeicherung ("let") kann man dies vermeiden.

## 3.2.5 Die Wurzel aus einer natürlichen Zahl a

Ausgangspunkt mag folgender Limerick sein, den man sich mehrfach im Internet ergoogeln kann:

An algebra teacher named Drew
tried to find the square root of two.
He found it between
one fourth and fourteen,
but couldn't get closer. Can you?

Dann wollen wir es einmal versuchen!



3.2.5.1: Betrachte a = 2. Weil Wurzel(2) die Länge der Diagonalen im Einheitsquadrat ist, kennen viele noch die Näherung Wurzel(2) ≈ 1,41421. In mathematischen Tafeln findet man 1,414213562373.

Wie und auf wie viele Ziffern kann man eine solche Zahl per Hand berechnen? Da man das Ergebnis z irgendwann verifizieren muss (sprich: es ist $z \cdot z \approx a$ nachzuprüfen), wofür man die Schulmethode der Multiplikation verwenden wird, gibt es bei etwa 500 Ziffern eine natürliche Grenze, da man hierfür 500·500, also 250.000 Einzelschritte ausführen muss, was mehrere Tage kostet. Mit Computern kommt man jedoch leicht viel weiter.

Die folgenden Ausführungen finden sich im Buch von Nievergelt, Farrar und Reingold, "Computer Approaches to Mathematical Problems", Prentice Hall aus dem Jahre 1974, und später in manchen anderen Büchern.



### Methode 1: Intervallschachtelung

```
function wurz2 (L, R: real) is
lokale Variable M: real := (L+R)/2.0;
if genau genug angenähert then Ergebnis ist M
else if M*M > 2 then wurz2 (L, M) else wurz2 (M, R) fi fi
Man berechne dann wurz2 (1.0, 2.0).
```

### Methode 2: Newtonsche Iteration

Starte mit $x_0 = 2.0$ und bilde (bis zu einem $x_m$) die Folge

$$x_{k+1} = x_k/2.0 + 1.0/x_k$$

Diese Folge konvergiert gegen Wurzel(2), siehe Mathematik.

```
function newton2 (x: real; m: natural) is
if m=0 then Ergebnis ist x else newton2 (x/2.0+1.0/x, k-1) fi
Man berechne dann z.B. newton2 (2.0, 50).
```



### 3.2.5.2 Methode 3: mit Hilfe der Pellschen Gleichung

Es gibt den folgenden zahlentheoretischen Satz:

Wenn die natürliche Zahl a keine Quadratzahl ist, dann hat die Gleichung $p^2 - a \cdot q^2 = 4$ (Pellsche Gleichung) unendlich viele Lösungen in natürlichen Zahlen p und q.

(Der Beweis garantiert, dass man, notfalls durch systematisches Probieren, ein p und ein q algorithmisch finden kann. Wenn es eine Lösung gibt, so gibt es auch unendlich viele Lösungen, siehe Behauptung auf der nächsten Folie).

Der Satz klingt nicht sehr aufregend, jedoch bildet er den Ausgangspunkt für eine schnelle exakte Berechnung der Wurzel aus a mit Hilfe von Computern. "exakt" soll hier bedeuten, dass die Näherung in Form einer rationalen Zahl erfolgt, also durch Angabe eines Zählers und eines Nenners.

Gegeben seien a, $p_0$ und $q_0$, so dass $p_0^2 - a \cdot q_0^2 = 4$ erfüllt ist. Da $a \geq 2$ ist, muss $p_0 \geq 3$ und $p_0 > q_0$ gelten.

Bilde nun die Folge von natürlichen Zahlen $p_k$ und $q_k$:

$$p_{k+1} = p_k^2 - 2 \quad \text{und} \quad q_{k+1} = p_k \cdot q_k$$

Behauptung: a, $p_k$ und $q_k$ erfüllen ebenfalls die Pellsche Gleichung, d. h., es gilt: $p_k^2 - a \cdot q_k^2 = 4$ .

*Beweis durch Nachrechnen:* Für k = 0 ist dies nach Voraussetzung richtig. Sei die Behauptung bis zu einem k bewiesen. Betrachte dann

$$p_{k+1}^2 - a \cdot q_{k+1}^2 = (p_k^2 - 2)^2 - a \cdot (p_k \cdot q_k)^2 = p_k^4 - 4p_k^2 + 4 - a \cdot p_k^2 \cdot q_k^2$$

$$= p_k^2 \underbrace{(p_k^2 - 4 - a \cdot q_k^2)}_{= 0} + 4 = 4,$$

also erfüllen auch a, $p_{k+1}$ und $q_{k+1}$ die Pellsche Gleichung.

= 0, weil a, $p_k$ und $q_k$ die Pellsche Gleichung nach Induktionsannahme erfüllen. ■



Aus den Anfangsbedingungen $p_0 \geq 3$ und $p_0 > q_0 \geq 1$ und aus

$$p_{k+1} = p_k^2 - 2 \quad \text{und} \quad q_{k+1} = p_k \cdot q_k$$

erkennt man: Die Folge der Zahlen $p_k$ und $q_k$ wächst sehr stark, und zwar jedes Mal ungefähr auf die doppelte Länge.

Wir formen $p_k^2 - a \cdot q_k^2 = 4$ um: $\dfrac{p_k^2}{q_k^2} = a + \dfrac{4}{q_k^2}$

Hieraus folgt:

$$\frac{p_k}{q_k} \xrightarrow[k \to \infty]{} \sqrt{a}$$

und die Konvergenzgeschwindigkeit ist hoch, da sich die Zahlen beim Übergang von k nach k+1 in der Länge fast verdoppeln.



### 3.2.5.3 Der Fall a = 2.

Für a = 2, $p_0 = 6$ und $q_0 = 4$ gilt $p_0^2 - a \cdot q_0^2 = 4$. Bilde nun die Zahlen $p_k$ und $q_k$ gemäß $p_{k+1} = p_k^2 - 2$ und $q_{k+1} = p_k \cdot q_k$ :

| k | $p_k$ | $q_k$ |
|---|---|---|
| 0 | 6 | 4 |
| 1 | 34 | 24 |
| 2 | 1154 | 816 |
| 3 | 1331714 | 941664 |
| 4 | 1773462177794 | 1254027132096 |

Für k = 3 stimmt $p_k/q_k \approx 1.41421356237469$ schon auf 11 Stellen nach dem Komma mit Wurzel(2) überein; für k = 4 sind es bereits mehr als 20 Stellen.



Nun konstruieren wir das zugehörige Scheme-Programm. Die Funktion nextzn (= "nächster zähler-nenner") berechnet aus der Zweier-Liste (p, q) die Liste $(p^2 - 2, p \cdot q)$. Dies müssen wir iterieren; wir bilden also **f-iter**, diesmal etwas allgemeiner als in 3.2.4. Am Ende muss man den Zähler p durch den Nenner q teilen, also "/" auf (p, q) anwenden. Der exakten rationalen Zahl sieht man die Ziffernfolge nicht an, daher stellen wir das Ergebnis "inexakt" nochmals durch "wu" dar. Die Anzahl der Iterationen sei k; wir definieren k in einem eigenen define-Ausdruck am Anfang.

```
;;; Betrachte folgende Definition für f-iter
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda  (x)  ((f-iter f (- m 1)) (f x)) )) )
```

```
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda  (x)  ((f-iter f (- m 1)) (f x)) )) )
```

Hier wird direkt die Funktion f-iter definiert - und nicht die Funktion f-x-iter aus 3.2.4 mit ihrem Argument x.

Wenn m = 0 ist, so wendet man die Identität auf den dann gegebenen aktuellen Parameter x an. Die Identität ist gegeben durch (lambda (x) x).

Anderenfalls wird die um eins verringerte Iteration auf die Funktion f angewendet.

f-iter erhält also das Argument x für die Funktion f nicht als Parameter und liefert somit am Ende keinen Wert, sondern:
f-iter ist eine Funktion!



Die Funktion nextzn ist offensichtlich:

```
(define (nextzn x)
        (list (- (* (car x) (car x)) 2) (* (car x) (car (cdr x)) ) ) )
```

Der Ausdruck

```
        (f-iter nextzn k)
```

liefert die k-fach iterierte Funktion von nextzn. Die Wurzel aus 2 erhält man, indem man diese Funktion auf die Anfangsliste '(6 4) ansetzt und auf das Ergebnis die Division anwendet:

```
(define (wurzel-exakt) (apply / ((f-iter nextzn k) '(6 4))))
```

Das Ergebnis ist hierbei eine rationale Zahl. Will man eine Dezimaldarstellung als Ausgabe, so muss man sich auf eine Ziffernfolge als Ausgabe beschränken, die nur eine Näherung ist. In Scheme schreibt man hierfür:

```
(define (wu) (exact->inexact (apply / ((f-iter nextzn k) '(6 4)))))
```



```
;;; Somit erhalten wir ein Scheme-Programm Wurzel(2)
(define k 4)
(define (nextzn x)
        (list (- (* (car x) (car x)) 2) (* (car x) (car (cdr x)) ) ) )
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda (x) ((f-iter f (- m 1)) (f x)) )) )
(define (wurzel-exakt) (apply / ((f-iter nextzn k) '(6 4))))
(define (wu) (exact->inexact (apply / ((f-iter nextzn k) '(6 4)))))

(wurzel-exakt) (wu) ==>      Das Ergebnis lautet für k = 3:
1  195025 / 470832
1.4142135623746899

(wurzel-exakt) (wu) ==>      Das Ergebnis lautet für k = 4:
1  259717522849 / 627013566048
1.4142135623730951
```



3.2.5.4 Der allgemeine Fall (a ist aber keine Quadratzahl).

Für a = 3, $p_0 = 4$ und $q_0 = 2$ gilt $p_0^2 - a \cdot q_0^2 = 4$. Bilde nun die Zahlen $p_k$ und $q_k$ gemäß $p_{k+1} = p_k^2 - 2$ und $q_{k+1} = p_k \cdot q_k$ :

| k | $p_k$ | $q_k$ |
|---|---|---|
| 0 | 4 | 2 |
| 1 | 14 | 8 |
| 2 | 194 | 112 |
| 3 | 37634 | 21728 |
| 4 | 1416317954 | 817711552 |

Für k = 3 stimmt $p_k/q_k \approx 1.7320508100147276$ auf 7 Stellen nach dem Komma mit Wurzel(3) überein; für k = 4 sind es mehr als 13 Stellen.

Man muss also nur die Anfangswerte $p_0$ und $q_0$ verändern, um die Wurzel einer anderen Zahl zu berechnen. Da es für $p \geq 3$ zu jedem q mit $p > q > 0$ ein rationales a mit $p^2 - a \cdot q^2 = 4$ gibt, kann man mit einem (p, q) starten und konvergiert dann gegen die Wurzel aus a. Diese Wurzeln liest man aus der Pellschen Gleichung ab: $a = (p^2-4)/q^2$.

| Anfangswerte p und q | | Quotient konvergiert gegen die Wurzel aus |
|---|---|---|
| 3 | 1 | 5 |
| 3 | 2 | 1,25 |
| 4 | 1 | 12 |
| 4 | 2 | 3 |
| 4 | 3 | 1,333333... |
| 5 | 1 | 21 |
| 5 | 2 | 5,25 |
| 5 | 3 | 2,666666... |
| 5 | 4 | 1,3125 |
| 6 | 1 | 32 |
| 6 | 2 | 8 |
| 6 | 3 | 3,555555.... |

| Anfangswerte p und q | | Quotient konvergiert gegen die Wurzel aus |
|---|---|---|
| 6 | 4 | 2 |
| 6 | 5 | 1,28 |
| 7 | 1 | 45 |
| 7 | 2 | 11,25 |
| 7 | 3 | 9 |
| 7 | 4 | 2,8125 |
| 7 | 5 | 1,8 |
| 7 | 6 | 1,25 |
| 8 | 1 | 60 |
| 8 | 2 | 15 |
| 8 | 3 | 6,666666... |
| 8 | 43,75 | ***usw.*** |



## 3.2.5.5 Anfangswerte durch Ausprobieren finden

Wie findet man Anfangswerte $p_0$ und $q_0$ zu einer natürlichen Zahl a, die nicht Quadratzahl ist?
Der zahlentheoretische Satz besagt, dass es solche Werte stets gibt. Also probieren wir einfach alle Werte für p und q durch. Wir beginnen mit p=3 und q=1. Falls $p^2-a \cdot q^2-4=0$ ist, haben wir ein geeignetes Paar (p, q) gefunden. Ist dieser Wert dagegen kleiner als 0, so muss p erhöht werden, anderenfalls muss q erhöht werden. Dies führt direkt zur Funktion such, welche zu einem Tripel (p q a) das (bzgl. der Zahl $p_0$ kleinste) Tripel ($p_0$ $q_0$ a) ermittelt, das die Pellsche Gleichung erfüllt:

```
(define (such p q a)
  (let ((z (- (- (* p p) (* a q q)) 4)))
    (cond ((= z 0) (list p q a))
          ((> z 0) (such p (+ q 1) a))
          (else (such (+ p 1) q a)))))
```



Die kleinsten Startwerte für eine Zahl a erhalten wir als zweielementige Liste nun durch:

```
(define a ...)
(define (such p q b)
  (let ((z (- (- (* p p) (* b q q)) 4)))
    (cond ((= z 0) (list p q b))
          ((> z 0) (such p (+ q 1) b))
          (else (such (+ p 1) q b)))))
(define (startwerte x)
  (let ((y (such  3 1 x)))
    (list (car y) (car (cdr y)))))
(startwerte a)
```

Mit diesem Programm erhält man (6 4) für a = 2, (4 2) für a = 3, (48 10) für a = 23 und (1860498  83204) für a = 500.

*Aufgabe:* Berechnen Sie Startwerte für a = 13, 19, 46, 73 und 97.



## 3.2.5.6 Scheme-Programm zur Berechnung von Wurzeln

```
;;; Zu definierende Funktionen für die Wurzelberechnung

(define (such p q b)
        (let ((z (- (- (* p p) (* b q q)) 4)))
                (cond ((= z 0) (list p q b))
                ((> z 0) (such p (+ q 1) b))
                (else (such (+ p 1) q b)))))
(define (startwerte x)
        (let ((y (such  3 1 x)))
                (list (car y) (car (cdr y)))))
(define (nextzn x)
        (list (- (* (car x) (car x)) 2) (* (car x) (car (cdr x)) ) ) )
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda (x) ((f-iter f (- m 1)) (f x)) )) )
```

```
;;; Nun kommen die Ein- und Ausgabefunktionen, z.B. für die
;;; Wurzel aus 11, berechnet mit 2 Iterationen. Mehrfach benötigte
;;; Zwischenwerte werden mittels define Variablen zu gewiesen:
;;; we für "wurzel-exakt" und wi für "16-stellige Wurzel".
(define k 2) (define a 11)
(define sw (startwerte a))
(define we (apply / ((f-iter nextzn k) sw)))
(define wi (exact->inexact (apply / ((f-iter nextzn k) sw))))
(define text "Differenz des Quadrats zu a: ")
(display "a = ") (display a) (display ",  Startwerte = ")
    (display (car sw)) (display " und ") (display (car(cdr sw)))
    (display ",  Iterationen = ") (display k) (newline)
(display we) (newline) (display (* we we)) (newline)
    (display text) (display (- a (* we we))) (newline)
(display wi) (newline) (display (* wi wi)) (newline)
    (display text) (display (- a (* wi wi)))
```

```
;;; Ausgabe für a=11 und k=2:
a = 11,  Startwerte = 20 und 6,  Iterationen = 2
79201/23880
6272798401/570254400
Differenz des Quadrats zu a: -1/570254400
3.3166247906197657
11.000000001753605
Differenz des Quadrats zu a: -1.753605261001212e-009
```

```
;;; Ausgabe für a=19 und k=3:
a = 19,  Startwerte = 340 und 78,  Iterationen = 3
892835161607686752012048304338256690696
0
797154625803024119506114035788463239040 1/4195550662121179576347968609412964416 00
Differenz des Quadrats zu a: -1/4195550662121179576347968609412964416 00
4.358898943540673
18.999999999999996
Differenz des Quadrats zu a: 3.552713678800501e-015
```




### 3.2.5.7 Diskussion der Effizienz

Das Programm ist nun relativ einfach, da es im Wesentlichen nur aus einer Suchfunktion für die Startwerte und dann einer iterierten Auswertung eines Ausdrucks besteht.

Ist es für die Praxis tauglich?

Leider nein. Der Grund liegt in der Berechnung der Startwerte. Die Ausprobiertechnik ist nicht effizient, weil die Werte sehr groß sein können. Standardbeispiel hierfür ist die Zahl a = 97, deren kleinste Startwerte in Dezimaldarstellung neun- bzw. achtstellig sind.

Wenn man jedoch die Startwerte kennt, dann ist das Verfahren sehr effizient. Für die dezimale Ziffernfolge müssten am Ende die Zahlen $p_k$ und $q_k$ allerdings noch dividiert werden, doch das ist kein allzu schweres Problem - versuchen Sie es einmal!



Für den Hausgebrauch reicht Lösungsmethode 1 aus, auch wenn die ständige Multiplikation M*M bei wachsender Genauigkeit zum entscheidenden Zeitfaktor wird:

```
(define eps 1.0e-300)
(define (wurzel L R a)
  (let ((M (/ (+ L R) 2)))
     (define diff (- a (* M M)))
     (if (> eps (abs diff))
         M
         (if (> diff 0) (wurzel M R a) (wurzel L M a))  )))
```

Programm für die Lösungsmethode 1

Mit dem Ausdruck (wurzel 1 a a) wird die Wurzel einer Zahl a sehr schnell auf dreihundert Stellen genau berechnet.
*Hinweis:* In DrScheme wird 1.0e-324 gleich 0.0 gesetzt. Für größere Genauigkeit müssen Sie dann die "exakte Darstellung" mit #e anfordern, also z. B. (define eps #e1.0e-800) schreiben.

Schneller als Lösungsmethode 1 arbeitet das Newtonverfahren (Lösungsmethode 2). Für eine Zahl a > 1 beginnt man mit $x_0 = a$ und bildet die Folge: $x_{k+1} = x_k/2 + a/(2x_k)$, die gegen Wurzel(a) konvergiert. Dies liefert das Programm:

Programm für die Lösungsmethode 2

```
(define k 19) (define a 2)
(define (next x) (+ (/ x 2) (/ a (* x 2))))
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda (x) ((f-iter f (- m 1)) (f x)) )) )
(define wurzel ((f-iter next k) a))
wurzel (newline) (- a (* wurzel wurzel))
```

Mit diesem Programm wird Wurzel(2) auf 400.000 Stellen genau berechnet. (Wir haben also den Mathematiklehrer Drew, siehe Anfangsfolie von 3.2.5, bestens helfen können.)



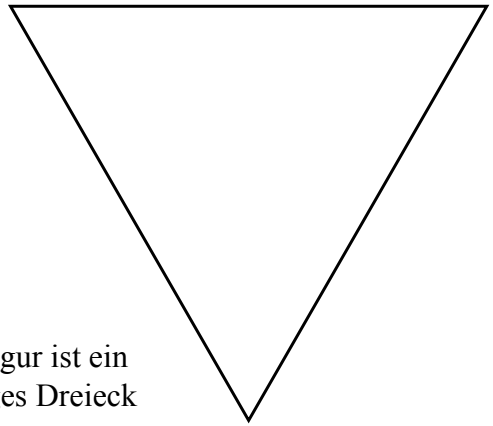
Konkrete Messung: Um die Wurzel von 2 auf 100.000 Stellen genau zu berechnen, benötigte die Lösungsmethode 1 (Intervallschachtelung) rund 5 Minuten, während auf dem selben Rechner die Lösungsmethoden 2 (Newton-Verfahren) und 3 (Pellsche Gleichung) für die gleiche Genauigkeit je 4 bis 6 Sekunden brauchten.

Lösungsmethode 2 brauchte 35 Sekunden für die Genauigkeit von 400.000 Stellen. Lösungsmethode 3 benötigte hierfür nur 16 Sekunden (allerdings ohne Ausgabe, die bei Methode 3 deutlich mehr Zeit benötigt als bei Methode 2, dies liegt wohl an der unterschiedlichen internen Darstellung). Lösungsmethode 1 arbeitet deshalb so langsam, weil sich bei jeder Iteration die Zahl der gültigen Stellen nur um konstant viele Stellen erhöht, während sie sich bei den Methoden 2 und 3 fast verdoppelt.



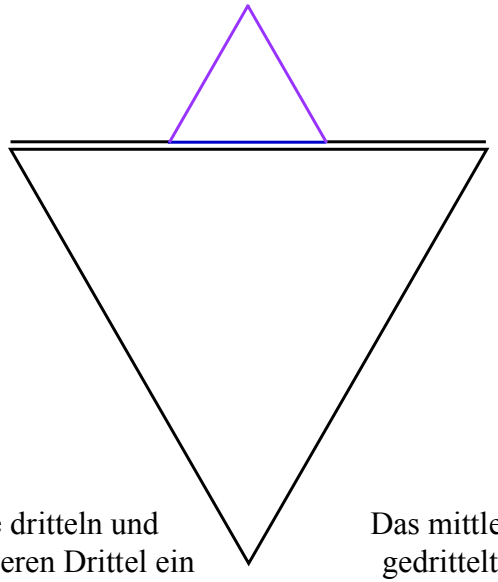
## 3.2.6 Zeichnen

Kochsche Seitendrittelungen (Schneeflocken)



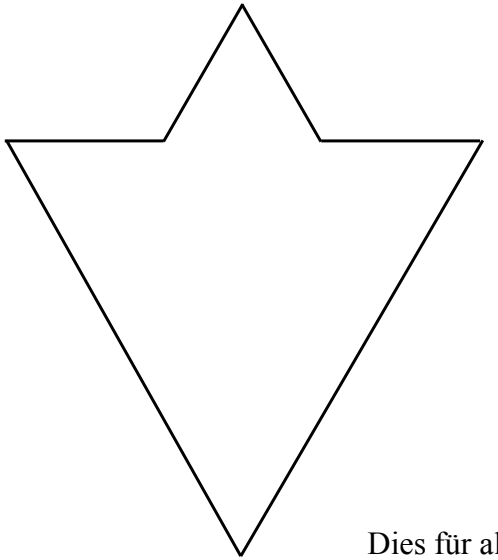
Ausgangsfigur ist ein gleichseitiges Dreieck





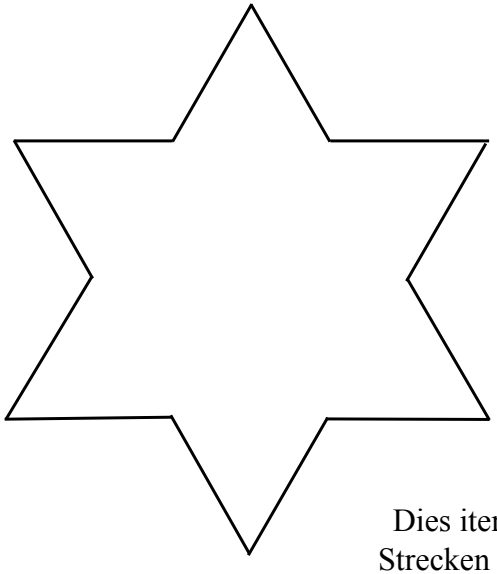
Eine Seite dritteln und auf dem mittleren Drittel ein gleichseitiges Dreieck aufsetzen

Das mittlere Drittel der gedrittelten Seite nun weglassen. Dies liefert:



Dies für alle drei Seiten
des Dreiecks durchführen:

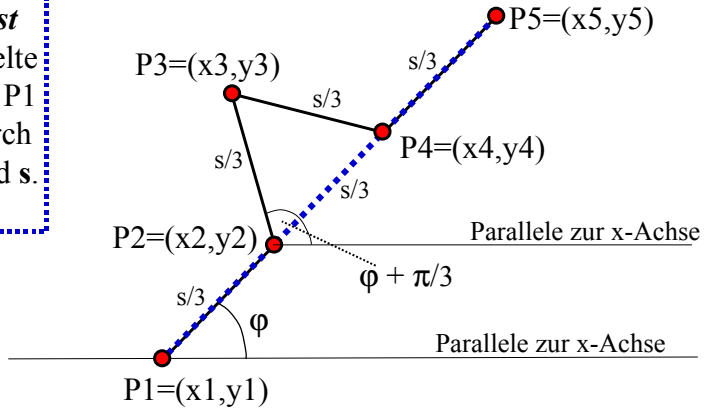




Dies iterativ für alle
Strecken wiederholen.



***Gegeben ist*** die gestrichelte Strecke von P1 nach P5 durch **x1, y1, φ** und **s**.




s = Länge der Strecke von P1 nach P5
s/3 = Entfernung P1 nach P2 und P4 nach P5
φ = Winkel der Strecke mit der x-Achse (im Bogenmaß!)

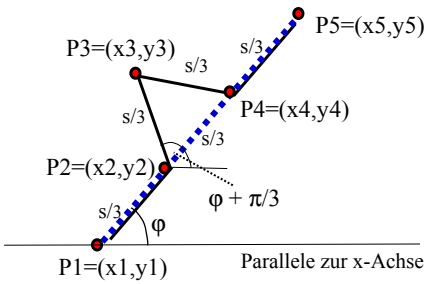
Nun müssen wir für den allgemeinen Fall die vier Strecken P1-P2, P2-P3, P3-P4 und P4-P5 durch die Werte x1, y1, φ, s beschreiben.



Strecke P1-P2: x1, y1, φ, s/3.

Strecke P2-P3: x2, y2, φ+π/3, s/3
mit x2 = x1 + (s/3)·cos(φ)
und y2 = y1 + (s/3)·sin(φ).

Strecke P3-P4: x3, y3, φ-π/3, s/3
mit x3 = x2 + (s/3)·cos(φ+π/3)
und y3 = y2 + (s/3)·sin(φ+π/3).




Strecke P4-P5: x4, y4, φ, s/3
mit x4 = x1 + (2s/3)·cos(φ) und y4 = y1 + (2s/3)·sin(φ).

Dies iterieren wir für jede Strecke, bis die Länge einer Strecke wegen der Drittelungen unterhalb einer vorzugebenden Größe gesunken ist; in diesem Fall zeichnen wir die Strecke P1-P5 und beenden die jeweilige Rekursion.

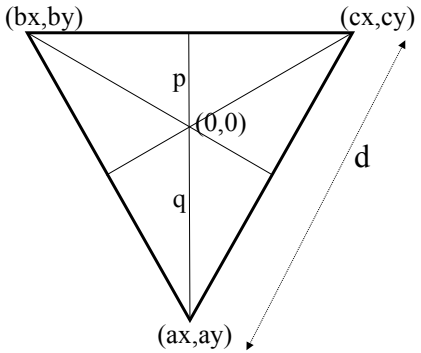
Nun brauchen wir nur noch die "Zeichnen"-Ausgabefunktion. Wir legen ohne Rücksicht auf die Realität fest: (draw-line x1 y1 x2 y2). Die Einheit ist cm und der Koordinatenursprung liegt in der Mitte des Bildschirms.

Wir bezeichnen den Schritt, ein Dreieck auf eine Strecke aufzusetzen, hier mit "dorn", da das Ergebnis eine Art Dorn ist.

```
(define (dorn x1 y1 phi s Min)
   (if (< s Min)
      (draw-line x1 y1 (+ x1 (* s (cos phi))) (+ y1 (* s (sin phi))))
      (let* ((s-drittel (/ s 3.0))  (zwei-s-drittel (+ s-drittel s-
drittel))
            (x2 (+ x1 (* s-drittel (cos phi))))
            (y2 (+ y1 (* s-drittel (sin phi))))
            (x3 (+ x2 (* s-drittel (cos (+ phi pi-drittel)))))
            (y3 (+ y2 (* s-drittel (sin (+ phi pi-drittel)))))
            (x4 (+ x1 (* zwei-s-drittel (cos phi))))
            (y4 (+ y1 (* zwei-s-drittel (sin phi)))) )
       (dorn x1 y1 phi s-drittel Min)
       (dorn x2 y2 (+ phi pi-drittel) s-drittel Min)
       (dorn x3 y3 (- phi pi-drittel) s-drittel Min)
       (dorn x4 y4 phi s-drittel Min)  ) ) )
```



Dies war die wörtliche Übertragung der Herleitungen nach Scheme. Die Ausgangspunkte A=(Ax,Ay), B=(Bx,By) und C=(Cx,Cy) sind durch den Koordinatenursprung im Schwerpunkt des gleichseitigen Dreiecks und die Seitenlänge "Strecke" d gegeben. Aus der Skizze und dem Pythagoras ermittelt man leicht folgende Gleichungen:




$p+q = h$ (Höhe),
$p^2+(d/2)^2 = q^2$,
$h^2 + (d/2)^2 = d^2$,
$ax = 0$, $ay = -q$,
$bx = -d/2$, $by = p$,
$cx = d/2$, $cy = p$.
Es folgt:
$p = (d/2)/\text{Wurzel}(3)$,
$q = d/\text{Wurzel}(3) = 2p$.



```
(define pi 3.14159265358979323846)
(define pi-drittel (/ pi 3.0))
(define (Schneeflocke d Min)
(let*  ((Ax 0) (Ay (/ (- d) (sqrt 3.0)))
        (Bx (/ (- d) 2)) (By (/ Ay 2)) (Cx (- Bx)) (Cy By))
   (dorn Ax Ay (* 2 pi-drittel) d Min)
   (dorn Bx By 0 d Min)
   (dorn Cx Cy (+ pi pi-drittel) d Min) ) )
(Schneeflocke 300 10)
```

*Wegen eines Softwarefehlers der DrScheme-Version konnte dieses Programm nicht getestet werden. Dies sei daher den Leser(inne)n überlassen. (Hinweise: Man verwende im Teachpack draw.ss. den Datentyp "posn" für Positionen; Funktion make-posn erzeugt eine Position aus zwei Zahlen, (draw-solid-line P1 P2 'blue) zieht eine blaue Linie von posn P1 nach posn P2, (start a b) liefert eine Zeichenfläche von a mal b Pixeln. Weiterhin liegt der Koordinatenursprung unten links.)*



*Gesamtes Programm* (mit dem Teachpack draw.ss und Verschieben des Ursprungs):

```
(define (dorn x1 y1 phi s Min)
    (if (< s Min)
        (draw-solid-line (make-posn x1 y1) (make posn (+ x1 (* s (cos phi))) (+ y1 (* s (sin phi)))))
        (let* ((s-drittel (/ s 3.0))  (zwei-s-drittel (+ s-drittel s-drittel))
                (x2 (+ x1 (* s-drittel (cos phi))))
                (y2 (+ y1 (* s-drittel (sin phi))))
                (x3 (+ x2 (* s-drittel (cos (+ phi pi-drittel)))))
                (y3 (+ y2 (* s-drittel (sin (+ phi pi-drittel)))))
                (x4 (+ x1 (* zwei-s-drittel (cos phi))))
                (y4 (+ y1 (* zwei-s-drittel (sin phi)))) )
         (dorn x1 y1 phi s-drittel Min)
         (dorn x2 y2 (+ phi pi-drittel) s-drittel Min)
         (dorn x3 y3 (- phi pi-drittel) s-drittel Min)
         (dorn x4 y4 phi s-drittel Min)  ) ) )
(define pi 3.14159265358979323846)
(define pi-drittel (/ pi 3.0))
(start 400 400)  -- Koordinatenursprung liegt zunächst unten links, d.h., nach (200,200) verschieben
(define (Schneeflocke d Min)
(let*((Ax 200) (Ay (+ (/ (- d) (sqrt 3.0)) 200))
          (Bx (+(/ (- d) 2) 200)) (By (+ (/ Ay 2) 200)) (Cx (- 200 Bx)) (Cy By))
    (dorn Ax Ay (* 2 pi-drittel) d Min)
    (dorn Bx By 0 d Min)
    (dorn Cx Cy (+ pi pi-drittel) d Min) ) )
(Schneeflocke 300 10)
```

## 3.2.7 m gültige Ziffern (dezimal)

Das eigentliche Ziel lautet: Drucke eine Tabelle der Funktion f.
Vom Benutzer sind später vorzugeben:
- Funktion f
- Genauigkeit der Berechnung als Zahl der Ziffern m
- kleinster x-Wert minx
- größter x-Wert maxx
- Schrittweite deltax.

Weiterhin muss die Anzahl maxz-pro-z der Zeichen, die pro Zeile maximal gedruckt werden dürfen, und die Anzahl z-pro-z der Zeichen, die pro Zeile gedruckt werden sollten, festgelegt werden.



Wir betrachten nur folgendes Teilproblem: Gegeben sind eine Zahl x > 0 und eine natürliche Zahl m. Ermittle zu x und m (= Genauigkeit) die Darstellung $x = g \cdot 10^e$ mit einer ganzen Zahl g, die aus genau m Dezimalziffern besteht und einer ganzen Zahl e.
Es gilt also: $10^{m-1} \le |g| < 10^m$, wodurch die Darstellung eindeutig ist.
Imperativer Algorithmus: Setze e auf Null.
Falls $|g| \ge 10^m$ ist, betrachte g := g/10 und e := e+1,
falls $|g| < 10^{m-1}$ ist, betrachte g := g*10 und e := e-1.
So erhält man schließlich die gewünschte Darstellung $x = g \cdot 10^e$.
Daraus folgt für die Berechnung von g und e in Scheme (hier muss man noch zwischen exakter und inexakter Darstellung unterscheiden):

```
(define (berechne-g  x  m)
    (cond  ((> (expt 10 (- m 1)) x) (berechne-g (* x 10) m))
           ((>= x (expt 10 m)) (berechne-g (/ x 10) m))
           (else x)  ) )
(define (berechne-e x m)
    (inexact->exact (round (log10 (/ x (berechne-g x m))))))
(define (log10 x) (/ (log x) (log 10)))
```



Programm zur dezimalen Ausgabe einer positiven Zahl x mit genau m vorgegebenen gültigen Ziffern

```
(define (berechne-g  x  m)
    (cond  ((> (expt 10 (- m 1)) x) (berechne-g (* x 10) m))
           ((>= x (expt 10 m)) (berechne-g (/ x 10) m))
           (else x)  ) )
(define (berechne-e x m)
    (inexact->exact (round (log10 (/ x (berechne-g x m))))))
(define (log10 x) (/ (log x) (log 10)))
;; x muss positiv sein und m eine natürliche Zahl
(define  x  0.123456789) (define  m  2)
(display x) (newline) (display (exact->inexact x)) (newline)
(display m) (newline)
(display (berechne-g x m)) (newline)
(display (exact->inexact (berechne-g x m))) (newline)
(display (inexact->exact (berechne-e x m))) (newline)
(display (inexact->exact (round (exact->inexact (/ (berechne-g x (+ m 1)) 10 ))))
(display " mal 10 hoch ") (display (berechne-e x m)) (newline)
```



Ausgabe des obigen Programms zur dezimalen Ausgabe von positiven Zahlen x mit m vorgegebenen gültigen Ziffern

```
;Ergebnis für obige Werte:
0.123456789
0.123456789
4
1234.5678899999998
1234.5678899999998
-4
1235 mal 10 hoch -4
```

```
;Ergebnis für x=317/7 und m=7:
317/7
45.285714285714285
7
31700000/7
4528571.428571428
-5
4528571 mal 10 hoch -5
```

```
;Ergebnis für x=0.006543, m=6:
0.006543
0.006543
6
654300.0000000001
654300.0000000001
-8
654300 mal 10 hoch -8
```

Was geschieht für m=0?

## 3.3 Funktionen höherer Ordnung

3.3.1 Erläuterung

Aus der Mathematik sind wir "Meta-Ebenen" gewohnt: Wir führen Elemente ein, bilden hieraus eine Menge, bilden aus den Mengen eine Menge von Mengen (z.B. die Potenzmenge), formen hieraus eine Menge von Mengen von Mengen usw. Dies übertragen wir auch auf Funktionen: Wir ordnen durch eine Funktion Elementen andere Elemente zu, und diese Elemente können wiederum Funktionen sein.

Auch in der Informatik gehen wir ständig mit Algorithmen um, die nicht auf "normale Daten", sondern auf Algorithmen angesetzt werden und als Ergebnis wieder Algorithmen liefern. Am bekanntesten sind Compiler; auch Anpassungen und die Programmierung von Benutzungsoberflächen gehören hierzu.



Funktionen, deren Argumente oder deren Ergebnis Funktionen sind, bezeichnet man als Funktionen höherer Ordnung.

Standardbeispiele aus der Mathematik sind die Integration und die Differentiation. Das unbestimmte Integral, das einer Funktion seine Stammfunktion zuordnet, ist z. B. eine Funktion höherer Ordnung.

Bereits behandelt hatten wir die Iteration einer Funktion f:

```
(define (f-iter f m)
        (if (= 0 m) (lambda (x) x)
        (lambda  (x)  ((f-iter f (- m 1)) (f x)) )) )
```



Ein anderes Beispiel ist die Bearbeitung aller Elemente einer Liste mit der gleichen Funktion:

```
(define (summe L) (if (null? L) 0 (+ (car L) (summe (cdr L)))))

(summe '(3 4 5))        ==> 12
(summe (list (summe '(1 2 3))  (summe '(4 5)) (summe '(6 7 8 9))))
                        ==> 45

(define (summe f L startwert)
    (if (null? L) startwert (f (car L) (summe f (cdr L) startwert))))
(summe + '(3 4 5) 0)            ==> 12
(summe * '(3 4 5) 1)            ==> 60
```



3.3.2 Die map-Funktion

Eine Funktion $f: A \to B$ lässt sich auf $F: A^k \to B^k$ erweitern durch

$$F(a_1, a_2, ..., a_k) = (f(a_1), f(a_2), ..., f(a_k)).$$

Wenn $(a_1\ a_2\ ...\ a_k)$ eine Liste ist, so bezeichnet man in Scheme die Funktion F durch F = map f. Das Ergebnis von map ist stets eine Liste.

*Syntax von map:* (map <proc> <list$_1$> <list$_2$> ... <list$_k$>)

Hier ist <proc> eine Funktion, deren Argument jedes der <list$_i$> sein kann und die genau einen Wert zurückliefert. Die Reihenfolge, in der <proc> auf die Listen <list$_i$> angewendet wird, ist nicht festgelegt. (Beispiel siehe drei Folien weiter.)

Beispiele:

```
(map + '( 1 2 3) '(4 5 6))
   ==> (5 7 9)
(map + '( 9  4 1) '(2 -3))
   ==> Fehler, weil die Listen nicht die gleiche Länge haben
(map + '( 9  4 1) '(2 -3 0) '(5 1 -1))
   ==> (16 2 0)
(map (lambda (x) (/ (* x (+ x 1)) 2)) '(1 2 3 4 5 6 7 8))
   ==> (1 3 6 10 15 21 28 36)
(map car '((a b c) (d e) (f g h i j k l) (m) (n o p) (q r s)) )
   ==>  (a d f m n q)
```

map kann man z. B. für Projektionen und für Tabellierungen verwenden.



Hinweis:

Man kann ein Analogon zu map auf Listen leicht definieren:

```
(define (parallel f)
   (lambda (x) (if (null? x) '( )
      (cons (f (car x)) ((parallel f) (cdr x))))))
```

Z. B. für (define quadrat (lambda (x) (* x x))) erhält man aus ((parallel quadrat) '(1 2 3 4 5 6 7 8 9)) das Resultat
(1 4 9 16 25 36 49 64 81)

map f bzw. (parallel f) sind also Funktionen höherer Ordnung, da sie aus einer Funktion f: $A \to B$ eine Funktion von $A^k$ nach $B^k$ (bzw. allgemein von A* nach B*) generieren.



Gefahrenhinweis: In Scheme sind Seiteneffekte wie in imperativen Sprachen möglich. Mit "set!" realisiert man Wertzuweisungen:

(set! <Variable> <Ausdr>)

setzt die <Variable> auf den Wert von <Ausdr>.
Folgendes Beispiel für einen Seiteneffekt: Die Funktion zählt, wie oft sie aufgerufen wird:

```
(let ((z 0))
      (map (lambda (x) (set! z (+ z 1)) z) '(7 8 9)) )
```

*Auswertung:* Die Funktion (lambda (x) (set! z (+ z 1)) z) wird dreimal angewendet, nämlich auf 7, 8 und 9. Da diese Reihenfolge aber nicht festgelegt ist, kann sie also zunächst auf 7, dann auf 8 und dann 9 angewendet werden, wobei das Ergebnis (1 2 3) entsteht; die Funktion kann jedoch auch in der Reihenfolge 8, 9, 7 angewendet werden, was das Ergebnis (3 1 2) liefert. Somit ist jede Permutation von (1 2 3) als Ergebnis möglich; das Ergebnis hängt also von der Implementierung ab! Dies eröffnet undurchschaubare Fehlerquellen. In "rein funktionalen Sprachen" (zu denen Scheme nicht gehört!) sind derartige Seiteneffekte daher gar nicht erst möglich.



*Beispiel zu map: Tabelle der Ulam-Collatz-Funktion*
(siehe Grundvorlesung 7.5.1 mit Veranschaulichungen)

Ulam(x) = if x = 1 then 0 else
if x gerade then Ulam(x/2)+1 else Ulam(3*x+1)+1 fi

```
(define a 30)
(define (Ulam x)
        (if (<= x 1)  0
            (if (= (remainder x 2) 0) (+ (Ulam (/ x 2)) 1)
                                      (+ (Ulam (+ (* 3 x) 1)) 1))))
(define (ZL x) (if (<= x 0) ( ) (cons x (ZL (- x 1))) ) )
(define Zahlenliste (reverse (ZL a)))
(map cons Zahlenliste (map Ulam Zahlenliste))
```

==> ((1 . 0) (2 . 1) (3 . 7) (4 . 2) (5 . 5) (6 . 8) (7 . 16) (8 . 3) (9 . 19) (10 . 6) (11 . 14) (12 . 9) (13 . 9) (14 . 17) (15 . 17) (16 . 4) (17 . 12) (18 . 20) (19 . 20) (20 . 7) (21 . 7) (22 . 15) (23 . 15) (24 . 10) (25 . 23) (26 . 10) (27 . 111) (28 . 18) (29 . 18) (30 . 18))

## 3.3.3 Was geschieht hier? (Undurchschaubarkeit)

*Beispiel 1:*

```
(define id (lambda (x) x))
(define quadrat (lambda (x) (* x  x)))
(display ((id quadrat) 6)) (newline)
(display (quadrat (id 6)))
```

*Beispiel 2:*

```
(define (zweimal f) (lambda (x) (f (f x))))
(define nachf (lambda (x) (+ x 1)))
(define n-hoch-n (lambda (x) (expt  x  x)))
((zweimal nachf) 6)
((zweimal n-hoch-n) 2)
((zweimal n-hoch-n) 6)
;;; überrascht vom Ergebnis??
```



*Beispiel 3:*

```
(define (ap f g) (lambda (x y) (f (g x y) (g y x))))
(display ((ap - -) 8 3)) (newline) (display ((ap * +) 8 3))
```

*Beispiel 4:*

```
(define zweimal (lambda (x y) (x (x y))))
(define kub (lambda (x) (* x x x)))
(display (zweimal kub (zweimal kub 3)))
```

*Beispiel 5:*

```
(define F1 (lambda (x)
   (if (> (F2 (- x 1)) (* x x)) (F2 (+ x 1) ) (F1 (- x 1)))))
(define F2 (lambda (x)
   (if (> (remainder x 6)(remainder x 5)) (* x x) (* 2 (F1 (- x 2))))))
(display (F1 2))
```



*Beispiel 6:*

```
(define F1 (lambda (x)
   (if (> (F2 (- x 1)) (* x x)) (F2 (- x 1) ) (F1 (- x 1)))))
(define F2 (lambda (x)
   (if (or (< x 0)(> (remainder x 6)(remainder x 5)))
       (* x x) (* 2 (F1 (- x 1))))))
(display (F1 1))    ==>  8
```

*Beispiel 7:*

```
(define F1 (lambda (x)
   (if (> (F2 (- x 1)) (* x x)) (F2 (- x 1) ) (F1 (- x 1)))))
(define F2 (lambda (x)
   (if (or (< x 0) (> (remainder x 6)(remainder x 5)))
       (* x x) (+ (F2 (- x 1)) (F1 (- x 1))))))
(display (F1 14))      ==>  551
```



*Beispiel 8 (vgl. Grundvorlesung 7.5.2):*

```
(define (MC x) (if  (> x 100)
                    (- x 10)
                    (MC (MC (+ x 11)))))
(display (MC 150)) (newline) (display (MC 100)) (newline)
(display (MC 15)) (newline) (display (MC -3))
```

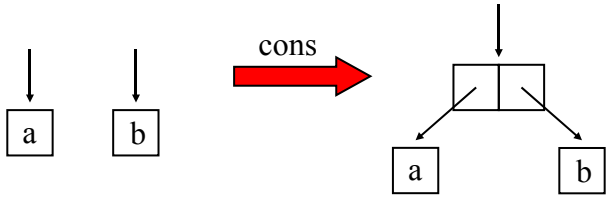
*Beispiel 9:*

```
(define (mittel f)
  (lambda (x) (if (= x 0)  0
                  (+ (* x (f x)) ((mittel f) (- x 1))))))
(define (h i) (if (= i 0) 0 (+ (/ 1 i) (h (- i 1)))))
(display ((mittel (mittel h)) 2)) (newline)
(display ((mittel (mittel h)) 3)) (newline)
(display ((mittel (mittel h)) 4))
```

# 3.4 Listen

## 3.4.1 Paare

Die zentrale Datenstruktur in Scheme (wie auch in LISP) sind binäre Bäume. Der Operator cons (= "constructor") fasst zwei Objekte zu einem binären Baum mit einer "inhaltsleeren" Wurzel zusammen ("dotted pair"): (cons a b) bedeutet

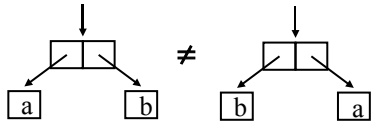



Auf die Komponenten kann man mittels car und cdr zugreifen, gesprochen "kahr" und "kudder". Die Bezeichnungen stammen von den Assemblerbefehlen der IBM-704 (ein viel benutzter Rechner aus den Jahren von 1954 bis 1960) car = content of address register und cdr = content of decrement register.

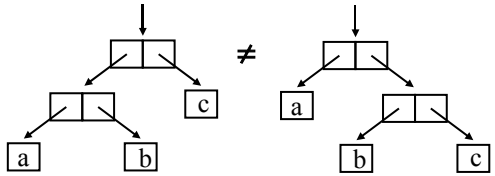


Die externe Darstellung von (cons a b) ist (a . b) . Ein durch cons gebildetes Objekt ist ein "Paar" ("pair"). Wie bei binären Bäumen ist cons weder kommutativ noch assoziativ, d. h.:

(cons a b) ≠ (cons b a)

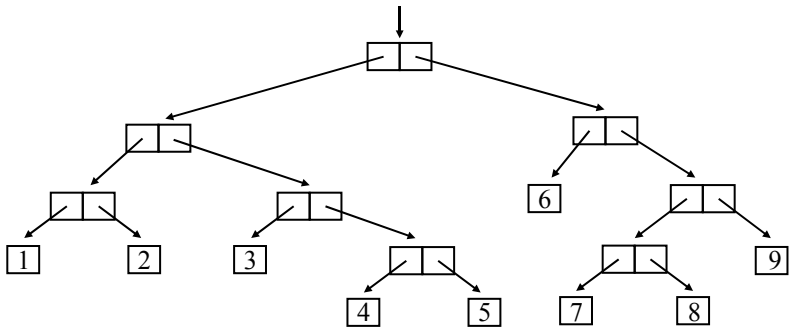



(cons (cons a b) c) ≠
(cons a (cons b c))



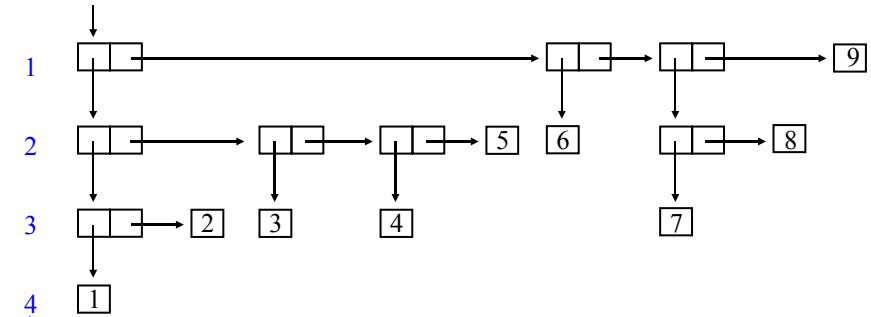



Beispiel:
(cons (cons (cons 1 2) (cons 3 (cons 4 5))) (cons 6 (cons (cons 7 8) 9)))
Dieser Ausdruck liefert die Darstellung: (((1 . 2) 3 4 . 5) 6 (7 . 8) . 9)
Als Baum geschrieben liegt folgende Struktur in Scheme vor:






In der Regel stellt man die linken Verweise durch senkrechte und die rechten Verweise durch waagerechte Linien dar. So werden den Elementen zugleich Ebenen (Schichten, Levels) zugewiesen. Für unser Beispiel (cons (cons (cons 1 2) (cons 3 (cons 4 5))) (cons 6 (cons (cons 7 8) 9))) :




Da man jeden geordneten Baum in einen binären Baum umwandeln kann (Binarisierung, Grundvorlesung 8.2.6), lassen sich mit Listen also auch allgemeine Bäume darstellen. Man kann zugleich gerichtete Graphen hiermit beschreiben (s. u.).

car liefert somit den linken und cdr den rechten Unterbaum eines Knotens. Es gilt also:
(car (cons a b)) = a  und  (cdr (cons a b)) = b

Die Operationen car und cdr werden verallgemeinert, im Revised Report aber nur bis zu 4 Buchstaben "a" und "d":

(car (car L)) = (caar L),  (car (cdr L)) = (cadr L),
(cdr (car L)) = (cdar L),  (cdr (cdr L)) = (cddr L),
(car (car (car L))) = caaar L), (car (car (cdr L))) = (caadr L), ...
(cdr (cdr (cdr (car L))))) = (cdddar L), ...

Mittels set-car! und set-cdr! kann man die Werte der Unterbäume wie bei einer Wertzuweisung verändern. Zum Beispiel:
(define L (cons 'A (cons 'B 'C)))  L
(set-car! L 'D)  L  (set-cdr! L 'E)  L
==>  (A . (B . C))  (D . (B . C))  (D . E)
[Statt (A . (B . C)) wird in DrScheme (A B . C) ausgegeben usw.]



3.4.2 Listen werden rekursiv definiert (vgl. 3.1.5):

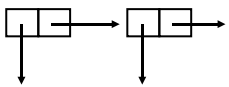
Die leere Liste ( ) ist eine Liste.

Wenn L eine Liste ist, dann ist auch (cons A L) eine Liste (für jedes Objekt A).

Die leere Liste in Scheme ist kein Paar, sondern ein spezielles Objekt mit eigenem Typ.
Eine Liste wird in der Form
$(A_1\ A_2\ ...A_k)$
geschrieben. Hiermit ist stets das Objekt
(cons $A_1$ (cons $A_2$ (cons $A_3$ ... (cons $A_{k-1}$ (cons $A_k$ ( ) )) ... )))
gemeint. Beispiel:
(cons 'A1 (cons 'A2 (cons 'A3 (cons 'A4 (cons 'A5 ( ))))))
liefert die Ausgabe  (A1 A2 A3 A4 A5) .
[Auch die Ausgabe
(A1 . (A2 . (A3 . (A4. (A5 . ( ) )))))  ist korrekt.]



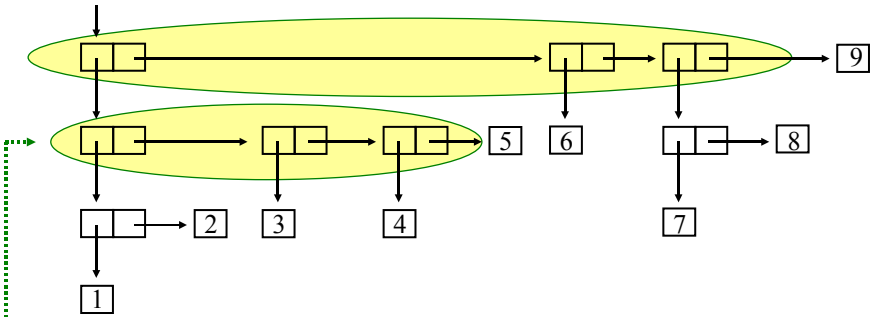
Listen lassen sich somit stets durch Bäume darstellen. In der Baumdarstellung sind "Listen-Teile" durch die Teilstruktur



zu erkennen. In der externen Darstellung werden diese Anteile dann in Listenschreibweise ( ...), also durch Zwischenraum getrennte Aufzählung ohne Punkte, notiert. Zum Beispiel liefert
(cons (cons 'A 'B) (cons 'C 'D))
die Ausgabe
((A . B) C . D)
weil die Form  (cons X (cons ...)  vorliegt.

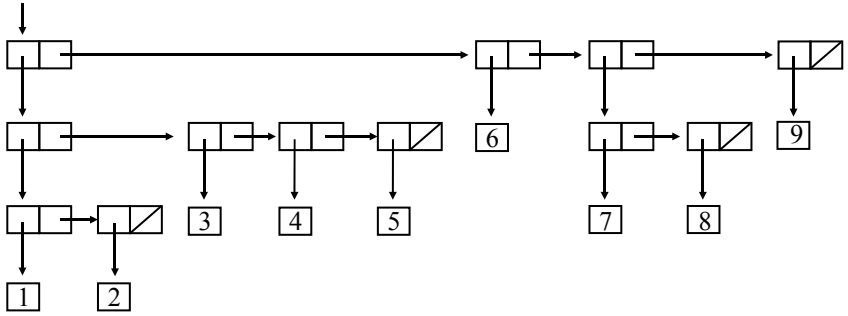


In unserem früheren Beispiel
(cons (cons (cons 1 2) (cons 3 (cons 4 5))) (cons 6 (cons (cons 7 8) 9)))




liegen die umrundeten Teile als Listenstrukturen vor, weshalb die externe Darstellung (Ausgabe in Scheme) lautet:
(((1 . 2) 3 4 . 5) 6 (7 . 8) . 9)
Listenstruktur mit 3 Elementen

Dieses Beispiel wandeln wir so um, dass nur noch Listen vorliegen:
**(cons (cons (cons 1 (cons 2 ( ))) (cons 3 (cons 4 (cons 5 ( ) ))))**
**(cons 6 (cons (cons 7 (cons 8 ( ))) (cons 9 ( ) ))))**
Dies liefert folgende Baumdarstellung, wobei wir die leere Liste durch ein Kästchen mit Nebendiagonale notieren:




Die Ausgabe lautet daher: (((1 2) 3 4 5) 6 (7 8) 9)
Insbesondere wird jeweils die letzte leere Liste nicht dargestellt.



Betrachte speziell die Listen:

| | |
|---|---|
| (cons ( ) ( )) | ==> (( )) |
| (cons ( ) (cons ( ) ( ))) | ==> (( ) ( )) |
| (cons (cons ( ) ( )) ( ) ) | ==> ((( ))) |

Eine Folge von Paaren, die nicht mit einer leeren Liste endet, wird manchmal auch als "unvollständige Liste" ("improper list") bezeichnet. Dies sind die Listenteilstrukturen, die wir oben angesprochen hatten und die in der Ausgabe berücksichtigt werden, z. B. (A B C . D) anstelle von (A . (B . (C . D))) .

*Wichtiger Hinweis:* Die Elemente einer Liste werden in Scheme als nulltes, erstes, zweites ... Element durchnummeriert. Die Zählung beginnt also mit 0 (und nicht wie gewohnt mit 1).



## 3.4.3 Operationen (Funktionen) auf Listen

cons, car, cdr, caar, cadr, set-car!, set-cdr! ... siehe oben.

pair? prüft, ob das Objekt ein Paar ist, also von der Form (cons A B). [Die leere Liste ist kein Paar.]

list? prüft, ob eine Liste vorliegt, d. h., ein Objekt, das bei wiederholtem cdr schließlich die leere Liste liefert.

null? prüft, ob das Objekt die leere Liste ist.

(list $A_1$ ... $A_k$) liefert eine Liste mit den Objekten $A_1$, ..., $A_k$ in dieser Reihenfolge. Speziell liefert (list) die leere Liste.

(length <Liste>) liefert die Länge der Liste (im 1. Level)
(length '(A (B C D) E (F G) H)) ==> 5



(append <$Liste_1$> <$Liste_2$> ... <$Liste_k$>) hängt die Listen zu einer Liste aneinander (auf dem 1. Level).
(append '(A (B C D)) '(E) '((F G) H)) ==> (A (B C D) E (F G) H)

(reverse <Liste>) dreht die Reihenfolge der Elemente der Liste um.

(list-tail <Liste> k) liefert die liste ohne die k ersten Elemente. Hier werden die k ersten Elemente entfernt; k = 0 liefert die Liste selbst. (list-tail '(A B C) 1) ==> (B C)

(list-ref <Liste> k) liefert das k-te Element der Liste. Man beachte, dass Listen ab 0 durchnummeriert werden, dass sich das k also auf diese Nummerierung bezieht. (list-ref '(A B C) 2) ==> C

(memv X <Liste>) liefert die längste Teilliste (1. Level!), die mit dem Objekt X beginnt (falls X nicht existiert, wird #f und nicht etwa die leere Liste geliefert). Beispiel:
(memv 'A '((A B) B (A) A (B A) (B B))) ==> (A (B A) (B B))

Einschub: Gleichheit von Objekten (siehe Report R5RS)

Erinnerung an imperatives Programmieren: Wann sind zwei Objekte gleich? (vgl. Grundvorlesung 1.13, 2.12, 3.4.7, 3.6.2)

In Scheme gibt es drei Formen der Gleichheit:

(eqv? <Objekt1> <Objekt2>)

(eq? <Objekt1> <Objekt2>) Dies ist im Wesentlichen dasselbe wie eqv?, liefert jedoch manchmal implementierungsabhängig bei Zeichen und Zahlen eventuell #f, wenn eqv? #t geliefert hätte.

(equal? <Objekt1> <Objekt2>) Dies liefert true, wenn beide Objekte die gleiche Auswertung haben, d. h., wenn sie die gleiche Ausgabe erzeugen.

Wir gehen im Folgenden nur auf eqv? ein.



Einschub: Gleichheit von Objekten (siehe Report R5RS)

Es gilt:

(eqv? <Objekt1> <Objekt2>) liefert genau dann #t, wenn

entweder beide Objekte zu Atomen ausgewertet werden können und diese sich ergebenden Konstanten gleich sind

oder beide Objekte zusammengesetzt sind und das identisch gleiche Objekt sind (also nicht eine Kopie voneinander, sondern das an gleicher Stelle im Speicher stehende Objekt).

Hinweis: Atome sind alle nicht zusammengesetzten Objekte, d. h., alle Objekte, für die eines der Prädikate
number?, boolean?, char? symbol?, null?
den Wert #t ergibt.



Einschub: Gleichheit von Objekten (siehe Report R5RS)

```
(eqv? 'A 'A)                    ==>  #t
(define a 2) (eqv? a 2)         ==>  #t
(eqv? (/ 3 15) (/ 2 10))        ==>  #t
(eqv? (/ 1 2) 0.5)              ==>  #f   -- unterschiedliche interne Darstellung
(eqv? (/ 1 2) (inexact->exact 0.5))      ==>  #t
(eqv? #t #t)                    ==>  #t
(let (( z 7)) (eqv? z z))       ==>  #t
(eqv? '(1 2) '(1 2))            ==>  #f
(define q (lambda (x) (* x x))) (eqv? 4 (q 2))              ==>  #t
(define w (lambda (x) (cdr '(1 2)))) (eqv? (w 5) (w 5))     ==>  #t
(define w (lambda (x) (cdr '(1 2)))) (eqv? (w 4) (w 5))     ==>  #t
(define w (lambda (x) (cdr '(1 2)))) (eqv? '(1 2) (w 2))    ==>  #f
(eqv? (cons 'a 'b) (cons 'a 'b))          ==>  #f
(eqv? "heute" "heute")                    ==>  #f
(eqv? (lambda (x) x) (lambda (x) x))      ==>  #f
(eqv? (eqv? ( ) '1) (eqv? ( ) '1))        ==>  #t
```



(memq X <Liste>) muss also das erste Element in <Liste> finden, welches gleich X ist. Erfolgt dieser Vergleich mit eq?, so nimmt man die Funktion memq, soll eqv? benutzt werden, so verwende man memv, und im Falle von equal? nehme man member.

(assv <Objekt> <Liste>)
<Liste> muss eine Liste von Paaren sein. Zurückgeliefert wird das erste Paar in der Liste, dessen car gleich <Objekt> ist. Kommt dagegen <Objekt> nicht als linker Teil eines Paars vor, so wird #f zurückgegeben. Beispiel:

(define z '((1 5) (2 6) (1 7))) (assv 1 z) (assv 2 z) (assv 3 z)
liefert (1 5) (2 6) #f

assq und assoc verwendet man, wenn die Gleichheit statt mit eqv? mittels eq? bzw. equal? ermittelt wird.

In der Regel lassen sich diese Funktionen leicht in Scheme selbst beschreiben.

*Beispiel:* Definition von (reverse L) in Scheme.

```
(define (spiegeln L)
   (if (null? L)  ( ) (append (spiegeln (cdr L)) (list (car L))) ))
```

*Beispiel:* Definition von (memv X <Liste>) in Scheme.

```
(define (abdann X L)
   (cond ((null? L)  #f)
         ((eqv? X (car L)) L)
         (else (abdann X (cdr L))) ) )
; Beispielanwendung:
(define c '((A B) B (A) A (B A) (B B)))
(memv 'A c)  (memv '(A A) c)  (abdann 'A c)  (abdann '(A A) c)
```



## 3.4.4 Beispiel: *Sortieren einer Liste durch Einfügen*

*Hierzu siehe Grundvorlesung: Anfang von Abschnitt 10.3.*

```
(define (einfuege-sortieren vergleich Liste)
   (define (einfuegen s L)
      (cond ((null? L)  (list s))
            ((vergleich s (car L)) (cons s L))
            (else (cons (car L) (einfuegen s (cdr L)) ) ) ) )
   (if (null? (cdr Liste))  Liste
      (einfuegen (car Liste) (einfuege-sortieren vergleich (cdr Liste)))))
(einfuege-sortieren  >  '(24 5 31 7 3 10 6 24))
(einfuege-sortieren  <=  '(24 5 31 7 3 10 6 24))
```

==> (31 24 24 10 7 6 5 3)
(3 5 6 7 10 24 24 31)

Gibt man die leere Liste ein, so bricht die Berechnung mit einem Fehler ab. Warum? Korrigieren Sie das Programm!



Experimente:

Sie können nun die Laufzeit untersuchen, indem Sie das Programm auf große Listen ansetzen. Diese erhält man, indem man zufällig viele Zahlen erzeugt.
Für eine natürliche Zahl b liefert

(random b)

irgendeine natürliche Zahl von 0 bis b-1.

Folgendermaßen kann man daher eine Liste von 50000 natürlichen Zahlen zwischen 0 und 399999 erzeugen.

```
(define (zliste a b)
   (if (> a 0) (cons (random b) (zliste (- a 1) b)) ( )))
(define Zahlenliste (zliste 50000 400000))
```

Mein 4 Jahre alter Laptop benötigte für das Sortieren von 10000 Zahlen (ohne Ausgabe) mittels der Funktion "einfuege-sortieren" rund 23 Sekunden, bei 50000 Zahlen waren es 650 Sekunden. Dies bestätigt grob die quadratische Abhängigkeit von der Zahl der Zahlen.
(Über die Ungenauigkeit der Messungen per Stoppuhr oder per Systembeobachtung siehe Veranstaltungen zu Betriebssystemen.)



*Hinweis auf Fehlerfallen mit Klammern:*

Lässt man in obigem Programm ein Klammerpaar in der Zeile, die mit "cond" beginnt, weg, so erhält man ein lauffähiges Programm, welches aber etwas anderes als "einfuege-sortieren" liefert:

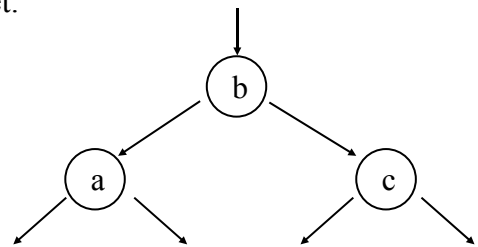
```
(define (einfuege-sortieren? vergleich Liste)
   (define (einfuegen s L)
      (cond (null? L)  (list s)
            ((vergleich s (car L)) (cons s L))
            (else (cons (car L) (einfuegen s (cdr L)) ) ) ) )
   (if (null? (cdr Liste))  Liste
      (einfuegen (car Liste) (einfuege-sortieren? vergleich (cdr Liste)))))
(einfuege-sortieren?  >  '(24 5 31 7 3 10 6 24))
```

Welches Ergebnis erhält man nun? Erklären Sie sich diese Änderung!

## 3.4.5 Sortieren mit Bäumen

*Hierzu siehe Grundvorlesung Abschnitt 3.7.7.*

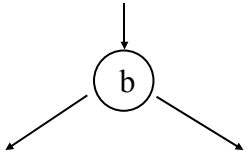
Wie in 3.4.1 angegeben kann man einen binären Baum mittels (cons X Y) aufbauen. Dann besitzen die Knoten jedoch noch keinen Inhalt. Diesen erhält man zum Beispiel, indem man noch mindestens ein Paar hinzufügt. Die gewohnte Darstellung lautet:



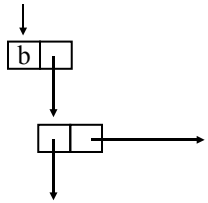



### 3.4.5.1 *Übliche Darstellung von Bäumen*

Einen Baumknoten K

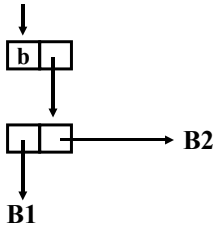



stellen wir dar in der Form




Also:
Inhalt des Knotens K = (car K)
linker Unterbaum = (cadr K)
rechter Unterbaum = (cddr K)






```
(define (make-treenode x B1 B2)
        (cons x (cons B1 B2) ) )
(define (links K) (cadr K))
(define (rechts K) (cddr K))
(define (inhalt K) (car K))
```

```
; Füge einen Schlüssel s in einen sortierten Baum B ein (insert-tree)
(define (insert-tree vergleich s B)
        (cond   ((null? B) (make-treenode s ( ) ( )))
                ((vergleich s (inhalt B))
(make-treenode (inhalt B) (insert-tree vergleich s (links B)) (rechts B)))
                (else
(make-treenode (inhalt B) (links B) (insert-tree vergleich s (rechts B)) ))))
```

```
; Aufbau (build-tree L) eines sortierten Baums B aus einer Liste L
(define (build-tree L)
            (if (null? L) ( )
                (insert-tree < (car L) (build-tree (cdr L))) ) )
```



*Nun ist die Liste in einen geordneten binären Baum (einen "Suchbaum") übertragen worden. Es fehlt zum Baumsortieren nur noch der abschließende inorder-Durchlauf durch den Baum:*

```
; den Baum B inorder in eine Liste auslesen
(define (inorder B)
  (if (null? B) ( )
     (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
```

Dieses Programm können wir dann auf eine Liste anwenden.

Das gesamte Programm findet sich auf der nächsten Folie.

## Gesamtprogramm zum Baumsortieren:

```
(define (make-treenode x B1 B2) (cons x (cons B1 B2) ) )
(define (links K) (cadr K))
(define (rechts K) (cddr K))
(define (inhalt K) (car K))
(define (insert-tree vergleich s B)
    (cond ((null? B) (make-treenode s ( ) ( )))
          ((vergleich s (inhalt B))
          (make-treenode (inhalt B) (insert-tree vergleich s (links B)) (rechts B)))
          (else
          (make-treenode (inhalt B) (links B) (insert-tree vergleich s (rechts B)) ))))
(define (build-tree vergleich L)
          (if (null? L) ( )
          (insert-tree vergleich (car L) (build-tree vergleich (cdr L))) ) )
(define (inorder B)
  (if (null? B) ( )
    (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
(inorder (build-tree > '(45 67 8 9 12 4 90 68 65 22 34 3 8 1 6 7 19 22 3 5 7 8 10)))
==> (90 68 67 65 45 34 22 22 19 12 10 9 8 8 8 7 7 6 5 4 3 3 1)
```

Wendet man dieses Programm auf die Liste mit 50.000 Elementen aus 3.4.4 an, d.h. fügt man die dortigen Ausdrücke

```
(define (zliste a b)
    (if (> a 0) (cons (random b) (zliste (- a 1) b)) ( )))
(define Zahlenliste (zliste 50000 400000))
(inorder (build-tree < Zahlenliste))
```

an das Programm an, so ist diese Liste (ohne Ausgabe) nach 3,4 Sekunden sortiert - anstelle der 650 Sekunden mit dem Sortieren durch Einfügen. Auch eine Liste mit 500000 Elementen benötigt nur 36 Sekunden zum Sortieren.

(Probieren Sie dies selbst aus.)



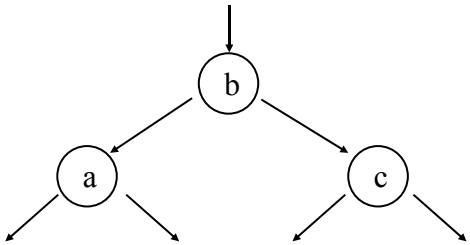


## 3.4.5.2 Beispiel:

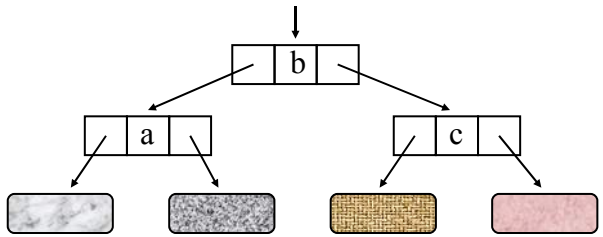
*Sortieren mit Bäumen, andere Baumdarstellung*

Die in 3.4.5.1 gewählte Darstellung für einen Baum ist nicht die einzig denkbare Möglichkeit. Aus der Vielzahl der Möglichkeiten greifen wir folgende heraus.

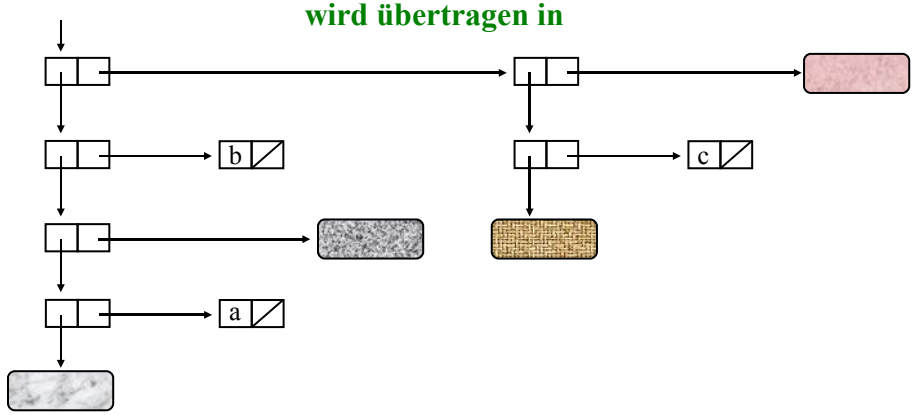
Die gewohnte Darstellung



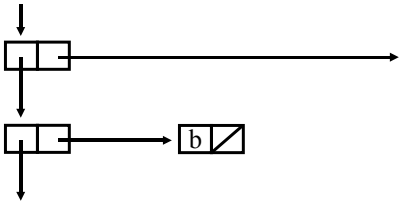

wird nun wie folgt nach Scheme übertragen:




**wird übertragen in**



**Der Knoten K eines Baums wird nun also dargestellt durch**

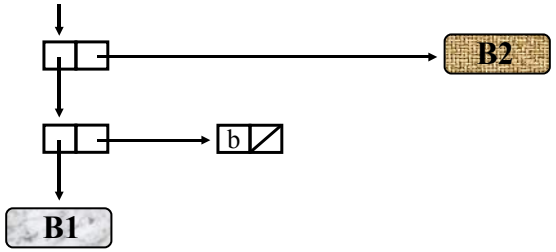



Auf den Inhalt des Knotens K greift man zu mittels (cadar K), den linken Nachfolgeknoten erhält man durch (caar K) und den rechten Nachfolger durch (cdr K). Die Hilfsfunktionen

make-treenode ... = bilde einen Knoten mit Inhalt ...
(links K) = Wurzel des linken Unterbaums
(rechts K) = Wurzel des rechten Unterbaums

müssen nun entsprechend definiert werden.






```
(define (make-treenode x B1 B2)
        (cons (cons B1 (cons x ( ))) B2) )
(define (links K) (caar K))
(define (rechts K) (cdr K))
(define (inhalt K) (cadar K))
```

Mit diesen Abänderungen für die Hilfsfunktionen kann man das Programm aus 3.4.5.1 wörtlich übernehmen und erhält:



Gesamtprogramm zum Baumsortieren (zweite Darstellung):

```
(define (make-treenode x B1 B2) (cons (cons B1 (cons x ( ))) B2) )
(define (links K) (caar K))
(define (rechts K) (cdr K))
(define (inhalt K) (cadar K))  -- nur der umrandete Teil ist neu
(define (insert-tree vergleich s B)
    (cond ((null? B) (make-treenode s ( ) ( )))
          ((vergleich s (inhalt B))
          (make-treenode (inhalt B) (insert-tree vergleich s (links B)) (rechts B)))
          (else
          (make-treenode (inhalt B) (links B) (insert-tree vergleich s (rechts B)) ))))
(define (build-tree vergleich L)
          (if (null? L) ( )
          (insert-tree vergleich (car L) (build-tree vergleich (cdr L))) ) )
(define (inorder B)
  (if (null? B) ( )
    (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
(inorder (build-tree < '(45 67 8 9 12 4 90 68 65 22 34 3 8 1 6 7 19 22 3 5 7 8 10)))
==> (1 3 3 4 5 6 7 7 8 8 8 9 10 12 19 22 22 34 45 65 67 68 90)
```



Wendet man dieses Programm auf die Liste mit 50.000 Elementen aus 3.4.4 an, d.h. fügt man die dortigen Ausdrücke

```
(define (zliste a b)
    (if (> a 0) (cons (random b) (zliste (- a 1) b)) ( )))
(define Zahlenliste (zliste 50000 400000))
(inorder (build-tree < Zahlenliste))
```

an das Programm an, so ist diese Liste (ohne Ausgabe) nach knapp 4 Sekunden sortiert. Für 500000 Elemente braucht dieses Programm 45 Sekunden.

Diese Darstellung benutzt je Knoten ein Paar mehr als die Darstellung in 3.4.5.1 und erfordert somit mehr Speicherplatz und Zugriffszeit. Diese Darstellung ist aber gut geeignet, wenn der Inhalt jedes Knotens zum Beispiel ein Paar ist mit (cadar K) als Schlüssel und (cddar K) als Dokument.

## 3.4.5.3 "Nebenprodukte" dieses Abschnitts

*Baumdurchläufe:*

```
(define (inorder B)
  (if (null? B) ( )
     (append (inorder (links B)) (list (inhalt B)) (inorder (rechts B)))))
(define (preorder B)
  (if (null? B) ( )
     (append (list (inhalt B)) (preorder (links B)) (preorder (rechts B)))))
(define (postorder B)
  (if (null? B) ( )
     (append (postorder (links B)) (postorder (rechts B)) (list (inhalt B)))))
```

*Wiederverwendung von Programmteilen mittels Polymorphie*

*Abstrakter Datentyp (Geheimniskonzept / hidden information)*



## 3.4.6 Potenzmengen

Rekursive Definition der Potenzmenge P(M) einer Menge $M = \{m\} \cup R$: $P(\emptyset) = \{\emptyset\}$ und $P(M) = P(R) \cup \{\{m\} \cup Q \mid Q \in P(R)\}$.
M als Liste darstellen. Setze m = (car M), R = (cdr M), T = P(R).
Dies lässt sich gut mit der map-Funktion beschreiben.

```
(define (Potenzmenge M)
   (if (null? M)
      (list '( ))
      (let ((T (Potenzmenge (cdr M)) ))
         (append T (map (lambda (Q) (cons (car M) Q)) T)) )))
```

```
(Potenzmenge '( ) )
(Potenzmenge '( 1) )
(Potenzmenge '( 1 2 3 4))
```

```
==> (( ))
(( ) (1))
(( ) (4) (3) (3 4) (2) (2 4) (2 3) (2 3 4) (1) (1 4) (1 3) (1 3 4) (1 2) (1 2 4) (1 2 3) (1 2 3 4))
```



## 3.4.7 Anmerkungen zu Zeichenketten (strings)

Eine Folge von Zeichen ist ein Wort. Der zugehörige Typ heißt in Scheme "string". Die Konstanten werden in "..." einge-schlossen, Sondersymbole durch \ abgetrennt, siehe 3.1.5.

Die Länge eines strings (length) ist die Anzahl der Zeichen im String, also eine nichtnegative ganze Zahl. Der leere String "" hat die Länge 0. Die einzelnen Zeichen eines Strings haben einen Index, nämlich ihre Nummer im String minus 1. (Die Zeichen sind also ab 0 durchnummeriert.)

| | |
|---|---|
| (string? x) | stellt fest, ob das Objekt x ein String ist. |
| (make-string k a) | erzeugt einen String der Länge k bestehend aus k-mal dem Zeichen, das zu a gehört. |
| (string a b ....) | erzeugt aus den Zeichen a, b, ... einen String. |
| (string-length s) | liefert die Länge des Strings s. |
| (string-ref s k) | liefert das (k-1)-te Zeichen des Strings s. |
| (substring s i j) | liefert den Teilstring von s ab Index i bis einschließlich Index (j-1), für $0 \le i \le j \le$ (length s). |
| (string->list s) | liefert die Liste der Zeichen des Strings s. |
| (list->string L) | wandelt eine Liste von Zeichen L in einen String um. |
| (string-copy s) | liefert eine Kopie des Strings s. |



# 3.5 Prinzipien der funktionalen Programmierung und zugehörige Programmiersprachen

## 3.5.1 Prinzipien (mit Varianten)

Charakteristika der imperativen Programmierung:
Das Programm ist eine Folge von Anweisungen; deren Ablaufreihenfolge wird durch bedingte Sprünge und Wiederholungen gesteuert. Variablen sind Behälter, deren Inhalte durch die Anweisungen verändert werden (die wichtigste Anweisung ist daher die Wertzuweisung). Die Menge dieser Inhalte zusammen mit der Position im Programm bilden den aktuellen Zustand des Programms. Man hat in der Regel direkten Einfluss auf die Speicherstruktur. Es gibt Konstrukte zum Strukturieren von Programmteilen (Block, Paket, Ausnahme, benannte Klammerungen, ...); wenn Teile zu eigenen aufrufbaren Programmteilen (Prozeduren) zusammengefasst werden können, spricht man auch von *prozeduraler Programmierung*. Daten werden in Datenstrukturen organisiert, wobei Verweise (Referenzen) auf Daten (d. h. auf Speicherbereiche) genutzt werden können. Letztlich iteriert ein imperatives Programm eine Abbildung auf dem Zustandsraum. Gedanklich steht im Hintergrund eine "komfortable" von-Neumann-Maschine.

Charakteristika der funktionalen Programmierung:
Das Programm ist eine Folge von Ausdrücken und Funktionen; Variablen dienen zu ihrer Benennung; Berechnungen erfolgen durch Anwenden von Funktionen auf Argumente und durch Auswertung von Ausdrücken. Die auszuwertenden Objekte und deren Ergebnisse können wiederum Funktionen sein (Funktionen höherer Ordnung). Im Idealfall sind Seiteneffekte nicht erlaubt.

Die einzelnen Programmiersprachen eines Programmierstils unterscheiden sich zum Beispiel durch:
- zulässige Objekte, Benennung und Verwendbarkeit
- Strukturierungsmöglichkeiten, Makros, Abkürzungen
- Seiteneffekte (vor allem Zustandsänderungen globaler Variablen)
- Typisierung (strikt, bedingt, untypisiert), Typsystem
- Polymorphie (parametrisieren, Generizität, überladen)
- Auswertungsstrategien (call by value/name/ref., lazy eval, ...)
- Abweichung von der "reinen Lehre"



Typsysteme in der Programmierung.

Die Zuordnung Bezeichner ↔ Objekt heißt "Bindung".
Diese Zuordnung wird in der Regel durch eine Deklaration hergestellt und gilt im gesamten zugehörigen Programmteil (Block, Lebensdauer des Bezeichners). Umdefinitionen in Unterbereichen sind möglich (lokale und globale Bezeichner): Es wird stets die zuletzt eingeführte Definition verwendet, wobei diese "statisch geschachtelt" im Sinne der Schachtelung des Programmtextes oder "dynamisch geschachtelt" im Sinne der Ausführungsreihenfolge während des Programmablaufs erfolgen kann.
Statische Bindung: Unveränderliche Festlegung durch die Deklaration.
Dynamische Bindung: Eindeutige Festlegung erst während der Programmausführung (z. B. bei rekursiven Prozeduren).
(Hier gibt es noch weitere Varianten.)



Eine Bindung wird meist mit Zusatzbedingungen versehen.
Die wichtigste ist der Typ. Eine *Programmiersprache* heißt
- statisch typisiert, wenn jedes Objekt (Konstante, Variable, ...) einen festen Typ während seiner Lebensdauer besitzt und dieser Typ vor der Programmausführung ermittelt werden kann,
- dynamisch typisiert, wenn sich der Typ eines Objekts durch Zuweisungen während des Programmlaufs ändern kann, aber zwischen solchen Zeitpunkten unverändert bleibt,
- typfrei, wenn der Typ eines Objekts erst zu dem Zeitpunkt festgestellt wird, zu dem dieses Objekt für eine Operation oder Funktion gebraucht wird (erst aus der aktuellen Interpretation der Operation ergibt sich der Typ des Objekts).

Eine *Typbindung* heißt
- stark, wenn mit dem Objekt nur Operationen, die für dessen Typ zulässig sind, durchgeführt werden,
- schwach, wenn gewisse Ausnahmen hiervon zugelassen sind.



Ein Datentyp legt die Menge der Werte und die auf ihnen zulässigen Operationen fest. (Ein abstrakter Datentyp legt nur Eigenschaften der Wertemenge und ihrer Operationen fest, er definiert im mathematischen Sinne also eine Algebra.)

Ein Typsystem dient mehreren Zwecken:

- Lesbarkeit, Verständlichkeit von Problemlösungen

- Fehlervermeidung

- Effizienz (z. B.: Typkontrolle zur Übersetzungszeit statt zur Laufzeit)

- Flexibilität, einfachere Verwendung im Falle des Überladens

Es gibt natürlich auch Folge-Fragen: Wann sind zwei Typen gleich, wann haben zwei Bezeichner den gleichen Typ, wie erfolgt die Typanpassung in Ausdrücken, ...?

## Auswertungsstrategien

call by name (textuelles Einfügen und anschließendes Auswerten; hierbei können/sollen Variablen lokal umdefiniert werden: statische oder dynamische Umgebung von Argumenten)

call by value (Argument auswerten und den Wert übergeben)

lazy evaluation (verzögere die Auswertung von Argumenten bis zu dem Zeitpunkt, zu dem sie wirklich gebraucht werden; man wird dann z. B. beim Aufruf von

(define (f A B) (if (= 3 4) A B ) )

das Argument A niemals auswerten. Diese Bedarfsauswertung wird mittlerweile bei vielen funktionalen Sprachen zur Auswertung benutzt. (Vorsicht bei undefinierten Argumenten, siehe Vorlesung über Formale Semantik.)



*Weitere Stichwörter (siehe Grundvorlesung und Literatur):*
Imperatives Programmieren, Informatik-Variablen, ...
externe / interne Darstellung
Blockstruktur, Speicherverwaltung
Seiteneffekte
Modularität, Kapselung, Geheimnisprinzip
Semantik, Verifikation, Sicherheit
interpretative (Sofort- und Teil-) Ausführung
call by name, call by value
verzögerte Auswertung (lazy evaluation)
Funktionen als Parameter und als Ergebnis
abstrakte Datentypen
Polymorphie (param., generic, überladen,...)
Wiederverwendbarkeit
Vererbung

Folgerung:

Eine extreme Vielzahl an Programmiersprachen!



## 3.5.2 Funktionale Programmiersprachen

Wie schon gesagt: Es gibt für jeden Programmierstil (imperativ, funktional, prädikativ, objektorientiert, ...) eine Vielzahl von Programmiersprachen. Die bekanntesten funktionalen Programmiersprachen sind zurzeit (2006):

λ-Kalkül (= der von Church 1936 entwickelte grundlegende Kalkül)
Lisp (von McCarthy 1960), Inter-Lisp, Emacs-Lisp
Common Lisp, Scheme, Dylan
Mathematica, Erlang
(APL), FP
Joy
Sisal
ML, lazy-ML, SML, Hope
Caml, Caml Light
Beta
OCAML
Gofer, Miranda, Haskell



Ein bisschen Klassifizierung:

| rein funktional? | Funktionen höherer Ordnung? | Auswertung | Typisierung | Sprachen |
|---|---|---|---|---|
| nein | nein | strikt | statisch | |
| nein | nein | strikt | dynamisch | |
| nein | nein | lazy | statisch | |
| nein | nein | lazy | dynamisch | |
| nein | ja | strikt | statisch | |
| nein | ja | strikt | dynamisch | |
| nein | ja | lazy | statisch | |
| nein | ja | lazy | dynamisch | |
| ja | nein | strikt | statisch | |
| ja | nein | strikt | dynamisch | |
| ja | nein | lazy | statisch | |
| ja | nein | lazy | dynamisch | |
| ja | ja | strikt | statisch | |
| ja | ja | strikt | dynamisch | |
| ja | ja | lazy | statisch | |
| ja | ja | lazy | dynamisch | |

# 4. Objektorientierte Programmierung (ooP)

4.1 Einführung in die Sprache Java

4.2 Beispiele

4.3 Konzepte und Objektorientierte Sprachen



## Gliederung von 4.1 Einführung in Java





Der objektorientierte Ansatz geht auf Arbeiten aus dem Jahre 1969 von Alan Kay zurück, der (auch hierfür) 2003 den Turing Award erhielt. Erste Ideen finden sich bereits in der Programmiersprache SIMULA 65, die ein einfaches Klassenkonzept mit sog. Koroutinen besitzt. Der Prototyp der ooP ist Smalltalk 80, eine objektorientierte Sprache auf Basis funktionaler Programmierung, die in den 70er Jahren in der Firma Xerox entstand und später im Projekt 'Squeak' weiterentwickelt wurde. Als erste rein objektorientierte Sprache mit kommerziellem Einsatz gilt Eiffel (Bertrand Meyer, 1988). Als Weiterentwicklung der relativ maschinennahen Sprache C hat sich seit 1983 C++ einen großen Marktanteil erobert, den sie sich seit kurzem immer mehr mit den Nachfolgesprachen Java und C# teilt.

Grundidee der Objektorientierung ist es, funktionsfähige Einheiten zu kapseln; diese bieten Strukturen und Funktionalitäten an, von denen man aber nicht angibt, wie sie realisiert sind. Man trennt also das "Was" vom "Wie". Mit dieser Grundidee lassen sich vor allem sehr umfangreiche Systeme leichter und sicherer modellieren, entwickeln, zusammenfügen, implementieren, anpassen und pflegen.



Folgende Prinzipien haben sich für das objektorientierte Programmieren herausgebildet (vgl. Abschnitt 4.6 der Grundvorlesung):

- Es gibt nur Objekte (keine anderen Einheiten).
- Was ein Objekt leisten kann, wird durch eine Spezifikation (oder Schnittstelle) beschrieben; wie ein Objekt diese Leistungen realisiert, bleibt verborgen.
- Objekte besitzen Zustände (dies entspricht den Werten ihrer Variablen) und Methoden (dies sind im Wesentlichen Algorithmen) und sie können miteinander kommunizieren.
- Objekte werden in Klassen zusammengefasst; die Objekte sind Instanzen oder Ausprägungen von Klassen.
- Klassen können ihre Strukturen an neue, andere Klassen weitergeben (Vererbung, inheritance, Klassenhierarchien).
- Gewisse Aspekte können offen gehalten werden und Variablen können für Objekte verschiedener Klassen stehen (Polymorphie, Abschnitt 4.4 der Grundvorlesung); die zu verwendenden Operationen hängen von der aktuellen Umgebung ab (dynamische Bindung, vgl. funktionale Programmierung, Abschnitt 3.5.1).

Hinweise zur Literatur: Zum objektorientierten Programmieren gibt es sehr viele Bücher. Die objektorientierte Vorgehensweise wird mittlerweile auch in den Schulen (und gerne mit Java) vermittelt, sodass viele Erstsemester bereits entsprechende Programmiererfahrungen besitzen.

Objektorientierung wird im künftigen Bachelorstudium im ersten Semester unterrichtet werden. Standardlehrbücher sind zum Beispiel mit den Autorennamen Balzert, Doberkat, B. Meyer, Oesterreich, Rumbaugh, Sneed usw. verbunden.

Daneben gibt es eine Fülle von Einführungen in die Objektorientierung, die sich in der Regel nur als Einführung in eine spezielle Sprache, insbesondere Java, und dann meist nur in eine aktuelle Version (z. B. JDK 1.4 oder 1.5) erweisen. Bei dieser Literatur ist es wichtig, sich rechtzeitig immer auf die nächste Version einzustellen.



Manche Einführungsbücher in die Informatik starten mit der Sprache Java und erläutern hieran wichtige Prinzipien. Beispiele sind:

Balzert, H., "Lehrbuch Grundlagen der Informatik", Spektrum Akademischer Verlag, Heidelberg, 2. Auflage, 2004

Gumm, H., Sommer, M., "Einführung in die Informatik", Oldenbourg Verlag, München, 7. Auflage, 2006

Horstmann, C.S., "Computing Concepts with Java 2 Essentials", Wiley, 2nd edition, 2006

Küchlin, W., Weber, A., "Einführung in die Informatik - objektorientiert mit Java", Springer, 3. Auflage, 2005

usw.

Zu Büchern über Java gebe ich keine Hinweise, da der Markt derzeit überflutet ist. Zusätzlich gibt es mindestens 10 Kurse im Internet, die Sie sich herunterladen können.



Im Bereich "Datenstrukturen und Algorithmen" weisen diverse Lehrbücher bereits im Titel auf Java hin, z. B.:

Lang, H.W., "Algorithmen in Java", Oldenbourg, München 2002,

Goodrich, M.T., Tamassia, R., "Data Structures and Algorithms in Java", Wiley, 2. Auflage, 2001

Saake, G., Sattler, K.U., "Algorithmen und Datenstrukturen, eine Einführung in Java", dpunkt-Verlag, 3. Auflage, 2006

Sedgewick, R., "Algorithmen in Java", Pearson Studium, 3. Auflage, 2003

Weiss, M.A., "Date Structures and Algorithm Analysis in Java", Addison Wesley, 2nd edition, 2007

(Dieser Hinweis unterstreicht zum einen, dass die Leser die Algorithmen konkret ausprobieren können, zum anderen ist er auch ein Verkaufsargument, um Praktiker anzusprechen; andere Bücher, die auf eine Sprachabhängigkeit verzichten, sind allein deshalb nicht besser oder schlechter - doch dies müssen Sie im Einzelfall selbst beurteilen, wenn Sie sich in einer Buchhandlung umsehen.)



# 4.1 Einführung in die Sprache Java

## 4.1.1 Ein erstes Beispiel

$x_1, x_2, x_3, \ldots$ bezeichnet man als eine Folge. Wenn eine solche Folge in einem Programm gegeben ist, kann man sie ab dem ersten bis zu einem n-ten Element ausdrucken. Wir wollen nun eine *Folge von Zahlen* betrachten.

Dies formulieren wir als Datentyp. Hierzu brauchen wir:

- das aktuell betrachtete Element (wir nennen es "aktuell"),
- ein erstes Element (wir nennen es "elem1"),
- eine Vorschrift, wie man den Nachfolger zu einem Element ermittelt (wir nennen diesen Algorithmus "naechstesElem"),
- eine Ausgabeanweisung für ein Element (dies nennen wir "drucke"),
- eine Ausgabeanweisung für die ersten n Elemente der Folge (wir nennen dies "druckeFolge (n)").

Wir sind nun nicht an einem einzelnen Programm interessiert, sondern an einem Schema, das man in anderen Programmen verwenden kann, ohne dass man angeben muss, wie die Folge aufgebaut ist. Formulierung in Ada-ähnlicher Darstellung:

```
schema zahlenfolge is
begin
  integer elem1, aktuell;
  function naechstesElem (integer x) return integer is begin ... end;
  procedure drucke (integer x) is begin ... end;
  procedure druckeFolge (positive n) is
     begin  drucke(elem1); aktuell := elem1;
          for i in 2..n loop
               aktuell := naechstesElem(aktuell);
               drucke(aktuell);
          end loop;
     end;
  end;
end;
```



Dies lässt sich direkt nach Java übertragen, wobei man einige Ersetzungen vornehmen muss:

"begin ... end" wird zu "{...}".

"is" weglassen.

"function" und "procedure" werden einheitlich als Funktionen (sog. "Methoden") aufgefasst. Hier wird zunächst der Ergebnisdatentyp angegeben (wenn es wie bei Prozeduren keinen gibt, so schreibt man "void"), dann folgt der Name, dann die Parameterliste.

"schema" wird zu "class".

"integer" und "positive" werden zu "long".

Die for-Schleife wird dargestellt durch for (int i=2; i <= n; i++). Hierbei ist i eine Integer-Laufvariable, die mit dem Wert 2 beginnt; sie wird wiederholt, solange i <= n ist; nach jedem Schleifendurchlauf wird i um 1 erhöht ("i++").

":=" wird zu "=".

Dann gibt es noch Besonderheiten, siehe übernächste Folie.



```
class zahlenfolge {
     long elem1, aktuell;
     zahlenfolge( ) {elem1=0; aktuell=0;}
     long naechstesElem (long x) { return aktuell; }
     void drucke (long x) {
          System.out.println (x);
     }
     void druckeFolge (long n) {
          drucke(elem1);
          aktuell = elem1;
          for (int i = 2; i <= n; i++) {
               aktuell = naechstesElem(aktuell);
               drucke(aktuell);
          }
     }
}
```

Beachten Sie das Layout! In Java sollte man sich an „gute Regeln“ halten!



Die Besonderheiten sind:

1) Das Ausdrucken auf das Standard-Ausgabegerät erledigt eine spezielle Methode print, die in einer Klasse mit Namen "System.out" definiert ist und die direkt im Programm genutzt werden darf. Soll anschließend zur nächsten Zeile übergegangen werden, so schreibt man println.

2) Man muss mindestens einen "Konstruktor" angeben, der den Namen der Klasse trägt. Diese spezielle Methode dient dazu, ein konkretes Objekt dieser Klasse zu bilden. Bei uns lautet er: zahlenfolge( ) {elem1=0; aktuell=0;}

3) Es darf nichts undefiniert bleiben. Daher haben wir den Rumpf ("body") der Methode naechstesElem vorläufig als { return aktuell; } angegeben.

Eine Klasse ist also ein Schema, aus dem konkrete Objekte gebildet werden. Möchte man ein solches Objekt anlegen mit dem Namen "Folge1", so schreibt man die Deklaration

zahlenfolge Folge1 = new zahlenfolge( );

Das reservierte Wort new bewirkt, dass eine Instanz (= eine konkrete Kopie) der Klasse zahlenfolge erzeugt und sofort die Anweisungen des Konstruktor-Rumpfs zahlenfolge ausgeführt wird (Initialisierung). Der Konstruktor ist somit eine Methode ohne Ergebnistyp (er ist also vom Typ "void", aber man lässt das Wort "void" in der Definition des Konstruktors weg).

Eine Instanz einer Klasse nennt man "Objekt".

Ein mächtiges Hilfsmittel ist die Vererbung. Hierzu gehen wir nun zu einer speziellen Folge von Zahlen über.



Wir betrachten Folge der "Dreieckszahlen" $x_1 = 1$ und $x_j = x_{j-1} + j$ für $j = 2, 3, ...,$ also die Folge 1, 3, 6, 10, 15, ... .

Dies ist eine Folge von Zahlen. Wir können also die bereits definierte Klasse zahlenfolge übernehmen. Hinzufügen müssen wir die "Nummer" j als neue Variable. Weiterhin müssen wir die Berechnung des nächsten Elements aus einem Element x anpassen, also neu definieren:

"Erhöhe Nummer und addiere diese Zahl zu x."

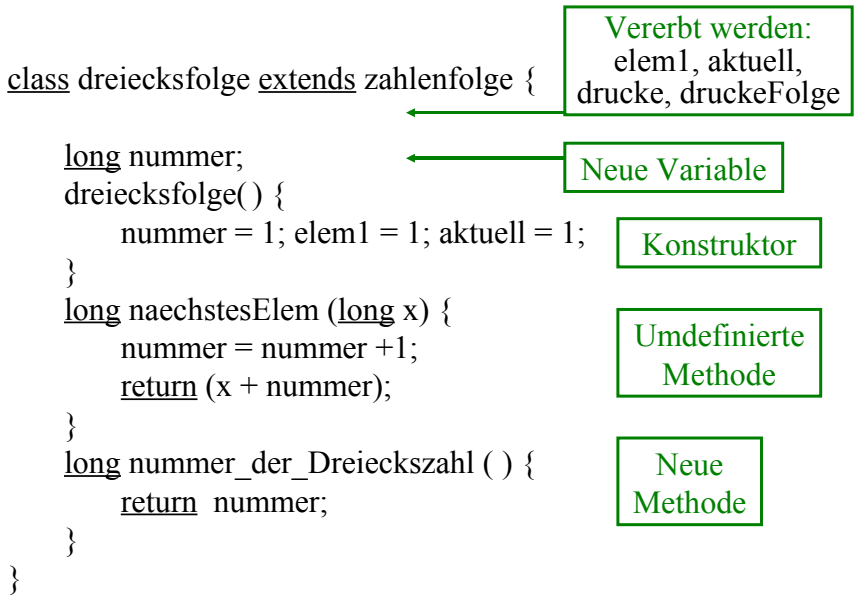
```
nummer = nummer +1;
return ( x + nummer );
```

In Java erweitert (reservierte Wort: extends) man nun die Klasse zahlenfolge zur Klasse dreiecksfolge ("Unterklasse") und notiert nur die Neuerungen und Veränderungen:



```
class dreiecksfolge extends zahlenfolge {

    long nummer;
    dreiecksfolge( ) {
        nummer = 1; elem1 = 1; aktuell = 1;
    }
    long naechstesElem (long x) {
        nummer = nummer +1;
        return (x + nummer);
    }
    long nummer_der_Dreieckszahl ( ) {
        return  nummer;
    }
}
```





Um diese beiden Klassen nun in einem Programm zu nutzen, fügen wir eine weitere Klasse mit irgendeinem Namen (wir wählen willkürlich "probieren") wie folgt hinzu (die Zeile mit *main* ignorieren Sie zunächst):

```
class probieren {
    public void static main (String[ ]  args) {
        zahlenfolge f = new zahlenfolge( );
        f.drucke(5);  f.druckeFolge(5);
        dreiecksfolge d = new dreiecksfolge( );
        d.drucke(5);  d.druckeFolge(5);
    }
}
```

Ausgabe: 5 0 0 0 0 0 5 1 3 6 10 15



Neben der Vererbung spielt das Überladen eine wichtige Rolle. Wie in der Grundvorlesung beschrieben, dürfen Funktionsnamen mehrfach verwendet werden, sofern durch den Kontext eindeutig klar ist, welche Funktion jeweils gemeint ist. In Java wird dies durch den Vektor der Argumentdatentypen festgestellt.

Zum Beispiel könnten wir bei der dreiecksfolge statt mit 1 mit einer anderen Zahl z beginnen wollen. In diesem Fall müssten wir elem1 und aktuell mit z initialisieren.

In Java kann man einfach einen weiteren Konstruktor in diese Klasse einfügen, der genau dies bewirkt.

Wir ändern also dreiecksfolge ab, indem wir eine weitere Konstruktor-Methode hinzufügen:



```
class dreiecksfolge extends zahlenfolge {
    long nummer;
    dreiecksfolge( ) {
        nummer = 1; elem1 = 1; aktuell = 1;
    }
    dreiecksfolge(long z) {
        nummer = 1; elem1 = z; aktuell = z;
    }
    ...
}
```

In der Klasse probieren kann man dann zum Beispiel schreiben:

```
dreiecksfolge dd = new dreiecksfolge(7);
dd.druckeFolge(5);
```

was die Ausgabe 7 9 12 16 21 bewirkt.



Aus Ada kennen Sie Schutzmechanismen. Diese gibt es in Java ebenso. Man kann Klassen, Methoden usw. mit Zusätzen ("Attribute" oder "modifier" genannt) versehen, z.B. bei Methoden:

public: relativ frei verfügbar / benutzbar von außerhalb des Pakets (= einer festgelegten Umgebung).

protected: diese Methode lässt sich nur aus dem gleichen Paket oder aus Unterklassen aufrufen.

private: nur in derselben Klasse aufrufbar.

kein Zusatz: die Methode heißt dann "friendly"; sie kann nur von Objekten von Klassen des gleichen Pakets aufgerufen werden.

Die relativ einfach aufgebaute Sprache Java wird durch die Fülle von Klassen, die in Bibliotheken abgelegt sind, und durch viele Hilfswerkzeuge und Entwicklungsumgebungen so unüberschaubar, dass man bereits von einem Berufsfeld des "Java-Programmierens" sprechen kann.

Wir gehen auf die professionellen Möglichkeiten nicht ein, sondern beschränken uns auf die grundlegenden Ideen und wie diese genutzt werden, um einige Probleme schön, elegant, effizient, korrekt, ... zu lösen. Insbesondere durch die Anwendungen in Netzen wurde Java stark verbreitet. Jede(r) von Ihnen wird daher irgendwann sicher erneut mit dieser Sprache (oder einer Nachfolgeversion) konfrontiert werden.



## 4.1.2 Grundbausteine der Sprache (siehe Grundvorlesung 3.2)

Java ist prinzipiell so aufgebaut wie andere Sprachen auch: Es gibt Grundsymbole (Literale) aus denen sich elementare Strukturen und Anweisungen ergeben, die durch geeignete Daten-/Kontrollstruktur-Operationen zu größeren Einheiten zusammengefügt werden. Die syntaktische Struktur wird durch die EBNF festgelegt.

Die meisten Sprachen und ihre Entwicklungsumgebungen werden ständig durch Weiterentwicklungen abgelöst. Für Java benötigt man ein JDK (Java Development Kit), eine JVM (Java Virtual Machine) und eine rasch anwachsende Zahl vieler Werkzeuge. Aktuell ist Anfang 2007 die JDK 1.6 und das Java SE 5 (SE = Standard Edition). Alle Hilfsmittel kann man aus dem Netz herunterladen.



### 4.1.2.1 Zeichensatz und Zeichen (siehe Grundvorlesung 2.4.7)

Als Zeichensatz verwendet Java den 16-stelligen Unicode (ISO-Standard 10646). Hiervon wird in Westeuropa meist nur der 8-stellige Latin1-Zeichensatz ausgenutzt .

Grundelement ist also das Zeichen (character).
Die übliche Unterteilung der Zeichen lautet:
Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Trennzeichen: ,(das Komma), ., ;, (, ), [, ], {, }
Leerzeichen: Die Latin1-Zeichen mit den Nummern 32, 9 und 12: Zwischenraum (space, Nummer 32), Tabulatorzeichen (HT, 9) und Seitenumbruch (FF, 12).
Zeilenende: Die Latin1-Zeichen mit den Nummern 10 und 13: Zeilenvorschub (LF, 10) und Wagenrücklauf (CR, 13). Die Folge LF CR wird als ein Zeilenende aufgefasst.
Operationszeichen: +, -, *, /, &, <, >, =, |, ^, %, !, ?, ~, :
Buchstaben: die üblichen Buchstaben (zum Schreiben von Text in irgendeiner Sprache) sowie $ und _



Darstellung der Zeichen: Jedes Zeichen (= Element des Unicode) wird wie üblich in Apostrophe eingeschlossen. Das Zeichen A wird also als 'A' dargestellt. Will man das Apostroph, die Anführungsstriche oder nicht auf der Tastatur vorhandene Zeichen darstellen, so muss man (wie in Scheme) das Zusatzzeichen \ ("Escape-Sequenz") verwenden. Beispiele hierfür:

| | |
|---|---|
| \' | bezeichnet das Apostroph ' |
| \" | bezeichnet die Anführungsstriche " |
| \\ | bezeichnet \ (backslash oder Rückwärtsquerstrich) |
| \u | beginnt die hexadezimale Darstellung eines Unicode-Zeichens. \u00A5 ist also das Unicodezeichen mit der 16-stelligen Bit-Repräsentation 0000000010101001 |
| \b | bezeichnet das Zurücksetzen um ein Zeichen (Backspace) |
| \r | bezeichnet den Wagenrücklauf (carriage return) |
| \n | bezeichnet den Zeilenvorschub (line feed) |
| \t | bezeichnet das Tabulatorzeichen |

Beispiele: '0' '§' ' ' 'Œ' 'λ' '\"' '\\' '\u0010' 'Щ' '∩'
(Hinweis: Escape-Sequenzen dürfen überall im Programm auftauchen; sie werden schon zur Compilezeit ersetzt.)

### 4.1.2.2 Zeichenketten

Aus Zeichen werden Zeichenketten (strings) zusammengesetzt. Wie üblich werden diese als Folge von Zeichen, die in Anführungsstriche eingeschlossen sind, dargestellt. Für Apostroph, Anführungsstriche und nicht gängige (auf der Tastatur nicht vorhandene) Zeichen ist der Backslash \ zu verwenden. Beispiele für Zeichenketten:

"Informatik in Stuttgart" "" (leere Zeichenkette)

"Der Java-Standardtext heißt: \"Hello World!\""

"Die \"Informatik in Stuttgart\""

"Um das Apostroph \' darzustellen, muss man \\\' schreiben."

"Lottozahlen\n 5\t 7\t30\t31\t42\t44\nliefert 6 Zahlen und 3 Zeilen"

(Dagegen liefert "\u0022" einen Fehler, weil zuerst \u0022 zum Zeichen " ausgewertet wird und dann """ nicht sinnvoll interpretiert werden kann.)



### 4.1.2.3 Reservierte Wörter in Java (JDK 1.5)

Folgende 50 Wörter haben eine feste Bedeutung in Java und dürfen nicht anders verwendet oder umdefiniert werden:

| | | | | |
|---|---|---|---|---|
| abstract | assert | boolean | break | byte |
| case | catch | char | class | const |
| continue | default | do | double | else |
| enum | extends | final | finally | float |
| for | goto | if | implements | import |
| instanceof | int | interface | long | native |
| new | package | private | protected | public |
| return | short | static | strictfp | super |
| switch | synchronized | this | throw | throws |
| transient | try | void | volatile | while |



Kurz gefasste Auflistung der Bedeutungen der reservierten Wörter (Vieles kennen Sie schon aus Ada oder Scheme):

| | |
|---|---|
| abstract | Angabe, dass eine Klasse oder Methode ohne Implementierung bleibt (abstrakte Klasse in Java) |
| assert | Ausnahmefall, falls Bedingung hinter assert false ist |
| boolean | elementarer Datentyp der Wahrheitswerte |
| break | sofortiges Beenden einer Kontrollstruktur |
| byte | 8-Bit langer Datentyp der ganzen Zahlen $-2^7..2^7-1$ |
| case | Fall in einer Fallunterscheidung (switch) |
| catch | Abfangen einer Ausnahmesituation |
| char | elementarer Datentyp der Zeichen |
| class | Beginn einer Klassendefinition |
| const | [noch ohne Bedeutung, erst künftig vorgesehen] |
| continue | Weitermachen beim Schleifenanfang |
| default | Standardfall in einer Fallunterscheidung |
| do | Schleife mit Abbruchprüfung nach dem Schleifenrumpf |



| | |
|---|---|
| double | 64-Bit langer Datentyp der reellen Zahlen |
| else | Beginn des alternativen Falls einer if-Anweisung |
| enum | zur Definition eines eigenen Aufzählungstyps |
| extends | Bildung einer Unterklasse durch Vererbung |
| final | Keine Unterklasse erlaubt bzw. Implementierung oder Wert nicht in Unterklassen veränderbar |
| finally | Abschluss gewisser Ausnahmen (catch ... finally ...) |
| float | 32-Bit langer Datentyp der reellen Zahlen |
| for | Laufschleife |
| goto | [noch ohne Bedeutung, erst künftig vorgesehen] |
| if | Alternative, Beginn der if-Anweisung |
| implements | Implementierung einer Schnittstelle durch eine Klasse |
| import | Einbindung einer anderen Klasse |
| instanceof | Test, ob ein Objekt zu einer Klasse gehört |
| int | 32-Bit langer Datentyp der ganzen Zahlen $-2^{31}..2^{31}-1$ |

| | |
|---|---|
| interface | Beginn einer Schnittstellendefinition |
| long | 64-Bit langer Datentyp der ganzen Zahlen |
| native | wird für Methoden, die in einer anderen Programmiersprache geschrieben sind, verwendet |
| new | erzeugt ein Objekt eines Referenzdatentyps |
| package | Paket-Zusammenfassung |
| private | Verstecken von Details, Verbieten von Zugriffen |
| protected | Schutz vor Zugriffen |
| public | Klasse ist erzeugbar und erweiterbar im gleichen Paket oder durch Importieren |
| return | sofortiges Verlassen einer Methode |
| short | 16-Bit-Datentyp der ganzen Zahlen $-2^{16}..2^{16}-1$ |
| static | Zuordnung einer Methode usw. zu einer Klasse |
| strictfp | erzwingt, dass Gleitpunktoperationen mit einer bestimmten Darstellung durchgeführt werden, um auf verschiedenen Plattformen stets gleiche Ergebnisse zu erhalten |
| super | Verweis auf die Oberklasse |



| | |
|---|---|
| switch | Beginn einer Fallunterscheidung |
| synchronized | Synchronisationsvorschrift bei Nebenläufigkeit |
| this | Verweis auf die eigene Klasse |
| throw | Übergang zu einer Ausnahmebehandlung |
| throws | Angabe, dass gewisse anzugebende Ausnahmen (in dieser Methode) ausgelöst werden können |
| transient | bezeichnet die Komponenten, die beim Speichern eines Objekts nicht gespeichert werden sollen |
| try | Beginn eines Blocks, in dem bestimmte Ausnahmen auftreten können (die in einem anschließenden catch- oder finally-Block behandelt werden) |
| void | Attribut einer Methode ohne Rückgabewert |
| volatile | Variablen, die durch nebenläufige Prozesse verändert werden können |
| while | Schleife mit Abbruchprüfung vor dem Schleifenrumpf (vgl. do) |



### 4.1.2.4 Bezeichner in Java (= identifier)

Bezeichner beginnen mit einem Buchstaben (beachte: $ oder _ sind erlaubt). Danach kann eine beliebige Folge von Buchstaben (einschließlich $ und _ ) und Ziffern stehen. Einschränkungen:

Als Bezeichner dürfen nicht verwendet werden:

- die reservierten Wörter,

- die Wörter ("Konstanten") true, false, null.

Auch die drei Wörter (für wichtige vordefinierte Klassen) Object, String, System sollte man nicht als Bezeichner verwenden, da hierbei leicht Konflikte entstehen können.

Wichtig: Achten Sie auf Groß- und Kleinschreibung! In Java beginnen Bezeichner für Methoden in der Regel mit einem kleinen, Bezeichner für Klassen mit einem großen Buchstaben.



### 4.1.2.5 Literale (vgl. Grundvorlesung 3.2.2)

Literale bezeichnen Konstanten elementarer Datentypen oder Zeichenketten. Elementare Datentypen (in Java meist "primitive" Datentypen genannt) sind [in Klammern die zugehörigen Java-Schlüsselwörter]:

- Boolesche Werte (boolean): false, true. (Default: false)

- Zeichen (char). (Default: \u0000)

- ganze Zahlen (byte, short, int, long), auch mit Vorzeichen, auch oktale und hexadezimale Darstellung ist möglich.

- Gleitpunktzahlen (float, double), auch mit Vorzeichen.

Weiterhin zählt die Null-Referenz null zu den Literalen.

Noch vorzustellen sind die numerischen Literale (ihr Default-Wert ist stets die Null) und deren Darstellung in Java:

byte (8-Bit-Darstellung): ganze Zahlen von -128 bis +127,

short (16-Bit-Darstellung): ganze Zahlen von -32768 bis +32767,

int (32-Bit-Darstellung): ganze Zahlen von $-2^{31}$ bis $+2^{31}-1$,

long (64-Bit-Darstellung): ganze Zahlen von $-2^{63}$ bis $+2^{63}-1$.

Gibt man nichts an, so ist "int" gemeint. 64-Bit-Darstellungen müssen am Ende ein großes oder kleines L besitzen. Oktale und hexadezimale Darstellungen sind erlaubt; sie beginnen mit 0 und dürfen nur die Ziffern 0 bis 7 verwenden, bzw. sie beginnen mit 0x und dürfen dann die Ziffern 0 bis 9 sowie die Buchstaben A bis F (oder auch a bis f) enthalten. Beispiele:

17 0 -5 144L 789123654L 0xA0F -0xFABEL
056 00 0xaffe



float (32-Bit-Darstellung) und double (64-Bit-Darstellung) sind Darstellungen für reelle Zahlen. Sie werden stets dezimal notiert. Ein Dezimalpunkt und/oder ein Exponent "e <ganze Zahl>" sind zulässig (E statt e ist erlaubt). Float-Literale erhalten ein nachgestelltes f oder F, double-Literale ein nachgestelltes d oder D; d darf auch fehlen. Eine Ziffernfolge ohne eines dieser Zusatzmerkmale wird als ganze Zahl interpretiert. Zum Beispiel ist 45 ganzzahlig, 45f dagegen eine Gleitpunktzahl.

Beispiele für double-Zahlen:
1.23 1.23e3 -9e-9 23D 15. 0.123 .123 0.123d .123d

Beispiele für float-Zahlen:
1.23f 1.23e3F -9e-9f 23f -0.123f .123F

float umfasst den Bereich bis maximal $\pm 10^{38}$, double bis $\pm 10^{308}$.



### 4.1.2.6 Kommentare (vgl. Grundvorlesung 3.2.2)

Es gibt drei Ausprägungen von Kommentaren in Java.

| | |
|---|---|
| // ... | Der Kommentar endet automatisch mit dem Zeilenende. |
| /* ... */ | Der Kommentar zwischen /* und */ darf beliebig lang sein. |
| /** ... */ | Wie /* ... */, doch der Kommentar kann von Java-Werkzeugen weiter verarbeitet werden, und zwar zur Übernahme in eine dokumentierte Fassung des Programms. Hier darf man auch Steuerzeichen einstreuen; bitte selbst ausprobieren. |



### 4.1.2.7 Operatoren in Java

| | |
|---|---|
| +, -, *, / | Übliche Operatoren auf Zahlen (/ bezeichnet zugleich die ganzzahlige Division) |
| % | Modulo-Bildung ganzer Zahlen |
| <<, >>, >>> | bitweises Verschieben nach links, nach rechts und nach links mit Nullextension |
| &, \|, ^ | bitweise Boolesches and, or und xor |

Schreibt man hinter jeden der obigen 11 Operatoren ein "=", so wird der Operator mit einer Zuweisung gekoppelt gemäß:
y += 8 bedeutet das Gleiche wie y = y + 8

| | |
|---|---|
| <, <=, >, >=, ==, != | die üblichen Vergleichsoperatoren |
| !, &&, \|\| | logisches not, and und or |
| ++, -- | Inkrement und Dekrement jeweils um 1 |
| = | ist der Zuweisungsoperator (in Ada: ":=") |
| + | Konkatenieren von Strings. |

Operatoren treten in Ausdrücken auf. Meist werden sie inorder verwendet. Man muss daher Prioritäten angeben. Die Liste der von unten nach oben wachsenden Prioritäten lautet:

| | |
|---|---|
| nachgestellte Postfixoperatoren: | ++ -- . [] (<Parameterliste>) |
| einstellige Präfixoperatoren: | + - ++ -- ! ~ |
| Referenz und casting: | new (<Datentyp>) |
| Mult./Div: | * / % |
| zweistellige Addieroperatoren: | + - |
| Verschieben (bitweise): | << >> >>> |
| Vergleich/Element: | < > <= >= instanceof |
| Gleichheit/Ungleichheit: | == != |
| and (bitweise): | & |
| exor (bitweise): | ^ |
| or (bitweise): | \| |
| and (boolean): | && |
| or (boolean): | \|\| |
| Bedingung im Ausdruck: | ? : |
| alle 12 Zuweisungsoperatoren: | = += -= *= *(usw.)* |



## 4.1.2.8 Ausdrücke

Ausdrücke werden wie üblich entsprechend der Stelligkeiten und Typen aus Operatoren und Werten/Variablen/Funktionen gebildet.

Auch Zuweisungen sind (Zuweisungs-)Ausdrücke x = y*(a+b). Daher ist auch x = q = r = y*(a+b) ein (Zuweisungs-) Ausdruck (sog. Mehrfachzuweisung). Ebenso ist

x -= q += z = (3+2)*8

zulässig. Die Auswertung der Zuweisungsoperatoren "-=", "+=" und "=" erfolgt von rechts nach links. Interpretiert man diesen Ausdruck als Anweisung (Wertzuweisung), so ist er gleichbedeutend mit der folgenden Folge von Anweisungen ("=" ist der normale Zuweisungsoperator):

z = (3+2)*8; q = q + z; x := x - q;



Autoinkrement und Dekrement: ++ und --.

Diese Operatoren können vor oder hinter einer Variablen stehen:

i++ bedeutet: Verwende zunächst den Wert von i und erhöhe anschließend den Wert von i um 1.

++i bedeutet: Erhöhe zuerst den Wert von i um 1 und verwende diesen Wert.

Analog für i-- und --i.

Beispiel:

```
int a = 1;              // a besitzt anschließend den Wert 1.
int b = ++a;            //Anschließend sind a gleich 2 und b gleich 2.
int c = b++;            //Anschließend sind c gleich 2 und b gleich 3.
int d = b--  +  ++a;    //Danach sind a gleich 3, d gleich 6, b gleich 2.
int e = ++c - d++;      //Danach sind c gleich 3, e gleich -3, d gleich 7.
```



Wenn # ein Operator ist, so bedeutet x #= e den Ausdruck x = x # e . Dabei wird die Adresse von x aber nur einmal berechnet. Diese Semantik führt zusammen mit Seiteneffekten zu schwer verstehbarem Code; zugleich ist die obige Aussage, x#=e sei gleich x=x#e nur noch für einfache Variablen richtig (Beispiel siehe nächste Folie).

Auch das Autoinkrement i++ (mit der Bedeutung i = i+1) kann je nach Programmumgebung andere Berechnungen auslösen, da innerhalb eines Ausdrucks stur von links nach rechts unter Beachtung der Prioritäten ausgewertet wird. Ein Beispiel ist:

```
a[i] = a[i] * ++i;      dies ist äquivalent zu:
                        i = i+1; a[i] = a[i-1] * i;
a[i] = ++i * a[i];      dies ist äquivalent zu:
                        i = i+1; a[i] = a[i] * i;
```

(Hier lauern beliebig viele Fehlerquellen.)

*Beispiel:* Gleiche Deklaration, zwei gleiche Zuweisungen?

```
int i = 0; int [ ] a = {0, 1, 2, 3, 4};
int g(int k) {
    i++; return (k+2);
}
```

*Zuweisung 1:* Die Adresse von a[g(i)] wird zweimal berechnet:

```
a[g(i)] = a[g(i)] + 5;    /* setzt a[3] auf den Wert a[2]+5 (== 7),
                             anschließend hat i den Wert 2. */
```

*Zuweisung 2:* Die Adresse von a[g(i)] wird nur einmal berechnet:

```
a[g(i)] += 5;             /* setzt a[2] auf den Wert a[2]+5 (== 7),
                             anschließend hat i den Wert 1. */
```



In Java (wie auch in C und in funktionalen Sprachen) werden Anweisungen zunächst wie Ausdrücke behandelt. Ein Ausdruck wird in Java zur Anweisung durch das Anfügen eines ";". So ist i++ zunächst ein Ausdruck, der zu einer Wertzuweisung durch Anhängen eines Semikolons wird: i++;

Auch die Alternative **if b then x else y** kann man als dreistelligen Operator auf Ausdrücken auffassen mit der Bedeutung: Ist b erfüllt, so sei x das Ergebnis, anderenfalls y.

In Java gibt es diesen dreistelligen Operator für Ausdrücke in der Form **b ? x : y**

*Beispiel:* (a < b) ? b : a
berechnet das Maximum von a und b, und

(a < b) ? ((c < a) ? c : a) : ((c < b) ? c : b)
berechnet das Minimum der drei Zahlen a, b und c.



## 4.1.3 Syntax von Java (basierend auf älterer Version, nicht ganz vollständig)

*Für das Folgende gibt es (wie stets) keine Garantie für Fehlerfreiheit.*
EBNF, d.h.: Eckige Klammern = ein- oder keinmal. Geschweifte Klammern = beliebig oft (inkl. keinmal). Runde Klammern dienen zum Zusammenfassen. Anführungszeichen klammern Terminalzeichen ein. Terminalzeichen sind zusätzlich in blau geschrieben. Kursives ist selbsterklärend. Nichtterminalzeichen stehen hier nicht in spitzen Klammern. Startsymbol der EBNF ist compilation unit. Den Abschluss jeder Regel bildet ein Punkt. Intervalle x..y bedeuten, dass jeder Wert von x bis y hier stehen darf. Es gibt noch Einschränkungen, die hier nicht sichtbar sind. Hier sind nun die wichtigsten 50 Regeln:

```
compilation_unit ::=
        [ package_statement ] { import_statement } { type_declaration }.
package_statement ::= "package" package_name ";" .
import_statement ::= "import" ( ( package_name "." "*" ";" ) |
                                ( class_name | interface_name ) ";" ) .
type_declaration ::=
        [d_comment] (class_declaration | interface_declaration) ";".
d_comment ::= "/**" Folge von Zeichen ohne */ "*/" .
```



```
class_declaration ::=
        {modifier} "class" identifier ["<" identifier {"," identifier} ">" ]
        [ "extends" class_name ] [ "implements" interface_name
        { "," interface_name } ] "{" {field_declaration} "}" .
interface_declaration ::= { modifier } "interface" identifier
        [ "extends" interface_name { "," interface_name } ]
        "{" { field_declaration } "}" .
field_declaration ::= ( [ d_comment ]
        ( method_declaration | constructor_declaration |
variable_declaration ) ) | static_initializer | ";" .
method_declaration ::= {modifier} type identifier "(" [ parameter_list ] ")"

        { "[" "]" } [ "throws" identifier {"," identifier} ]
        ( statement_block | ";" ) .
constructor_declaration ::= {modifier} identifier "(" [ parameter_list ] ")"

        [ "throws" identifier {"," identifier} ] statement_block .
statement_block ::= "{" { statement } "}" .
variable_declaration ::= { modifier } type variable_declarator
```

```
variable_declarator ::= identifier { "[" "]" } [ "=" variable_initializer ] .
variable_initializer ::= expression |
        ( "{" [variable_initializer {"," variable_initializer} [ "," ] ] "}" ) .
static_initializer ::= "static" statement_block .
parameter_list ::= parameter { "," parameter } .
parameter ::= type identifier { "[" "]" } .
statement ::= variable_declaration | ( expression ";" ) |
statement_block | if_statement | do_statement |          while_statement
| for_statement | assert_statement |
        try_statement | switch_statement |
        ( "synchronized" "(" expression ")" statement ) |
        ( "return" [ expression ] ";" ) | ( "throw" expression ";" ) |
        ( identifier ":" statement ) | ( "break" [ identifier ] ";" ) |
        ( "continue" [ identifier ] ";" ) | ";" .
if_statement ::= "if" "(" expression ")" statement [ "else" statement ] .
do_statement ::= "do" statement "while" "(" expression ")" ";" .
while_statement ::= "while" "(" expression ")" statement .
```



```
assert_statement ::= "assert" logical_expression [ ":" string ] ";"
for_statement ::= "for" "(" ( variable_declaration | ( expression ";" ) |
";" ) [ expression ] ";" [ expression ] ")" statement .
try_statement ::= "try" statement
        { "catch" "(" parameter ")" statement } [ "finally" statement ] .
switch_statement ::= "switch" "(" expression ")"
        "{" { ( "case" expression ":" | "default" ":" ) statement } "}" .
expression ::= numeric_expression | testing_expression |
        logical_expression | string_expression | bit_expression |
        casting_expression | creating_expression | literal_expression |
        "null" | "super" | "this" | identifier | ( "(" expression ")" ) |
        ( expression ( ( "(" [ arglist ] ")" ) | ( "[" expression "]" ) |
        ( "." expression ) | ( "," expression ) |
        ( "instanceof" ( class_name | interface_name ) ) ) ) .
numeric_expression ::= ( ( "-" | "++" | "--" ) expression ) |
        ( expression ( "++" | "--" ) ) | ( expression ( "+" | "+=" |
        "-" | "-=" | "*" | "*=" | "/" | "/=" | "%" | "%=" ) expression ) .
```



```
testing_expression ::= expression
        ( ">" | "<" | ">=" | "<=" | "==" | "!=" ) expression .
logical_expression ::= ( expression ( "&" | "&=" | "|" | "|=" | "^" |
        "^=" | ( "& &" ) | "||=" | "%" | "%=" ) expression ) |
        ( "!" expression ) | ( expression "?" expression ":" expression ) |

        "true" | "false" .
string_expression ::= ( expression ( "+" | "+=" ) expression ) .
bit_expression ::= ( "~" expression ) |
        ( expression ( ">>=" | "<<" | ">>" | ">>>" ) expression ) .
casting_expression ::= "(" type ")" expression .
creating_expression ::= "new" ( ( classe_name "(" [ arglist ] ")" ) |
                ( type_specifier [ "[" expression "]" ] { "[" "]" } ) |
                                ( "(" expression ")" ) ) .
literal_expression ::= integer_literal | float_literal | string | character .
arglist ::= expression { "," expression } .
type ::= type_specifier { "[" "]" } .
type_specifier ::= "boolean" | "byte" | "char" | "short" | "int" | "float" |
        "long" | "double" | class_name | interface_name .
```



```
modifier ::= "public" | "private" | "protected" | "static" | "final" |
"native" | "synchronized" | "abstract" | "transient" | "volatile".
package_name ::= identifier | ( package_name "." identifier ) .
class_name ::= identifier | ( package_name "." identifier ) .
interface_name ::= identifier | ( package_name "." identifier ) .
integer_literal ::= ( ( "1..9" {"0..9"} ) | " 0 " {"0..7"} |
        ( "0" "x" "0..9 a..f" {"0..9 a..f"} ) ) [ "l" | "L" ] .
float_literal ::= ( decimal_digits "."
        [ decimal_digits ] [ exponent_part ] [ float_type_suffix ] ) |
        ( "." decimal_digits [ exponent_part ] [ float_type_suffix ] ) |
        ( decimal_digits [ exponent_part ] [ float_type_suffix ] ) .
decimal_digits ::= "0..9" { "0..9" } .
exponent_part ::= ("e" | "E") [ "+" | "-" ] decimal_digits .
float_type_suffix ::= "f" | "F" | "d" | "D" .
character ::= " Element des unicode-Zeichensatzes " .
string ::= " " " {character} " " " .
identifier ::= "a..z,$,_" {"a..z,$,_,0..9, unicode-Zeichen oberhalb 00C0"} .
```

*Hinweise:*
Statt a..f darf in der Regel für integer integral auch A..F stehen.
Das Gleiche gilt für a..z in der Regel für identifier.
("a..z,$,_ ,0..9") ist als Vereinigungsmenge von Intervallen und Elementen zu lesen.)
Man kann zusätzliche Nichtterminalzeichen einführen, z. B.:

```
decimal_digit ::= " 0..9 "
separator ::= "," | "." | ";" | "(" | ")" | "[" | "]" | "{" | "}"
operator_character ::=  "+" | "-" | "*" | "/" | "%" | "&" | "|" | "!" | "^" |
                        "?" | "~" | "!" | "<" | ">" | "=" | ":"
```

*Obige Syntax orientiert sich an der ursprünglichen Definition von Java (1995). Für Java 1.5 ist sie aber unvollständig. Hierfür gibt es eine umfangreichere Syntax mit feiner gegliederten Nichtterminalen.* In unseren Regeln fehlen die später hinzugefügten Sprachelemente, z.B. Regeln für enum. *Schema hierfür:*

```
enumDeclaration ::= "enum" identifier [implementsList] enumBody .
enumBody ::= "{" enumConstant { "," enumConstant }
             [ ";" { classOrInterfaceBodyDeclaration } ] "}" .
enumConstant ::= identifier [arglist] [classOrInterfaceBody] .
implementsList ::= implements classOrInterfaceType {"," classOrInterfaceType} .
```



## 4.1.4 Typen, Klassen, Objekte

### 4.1.4.1 Datentypen

Es gibt die *primitiven Datentypen* (in Klammern die Zahl der Bytes, die für ihre Realisierung erforderlich sind; Zahlen werden intern im Zweierkomplement dargestellt):

boolean (1), char (2), byte (1), short (2),
int (4), long (8), float (4), double (8),

die selbstdefinierten *Aufzählungstypen* der einfachen Form

enum <Name> { <Liste der Elemente> } ;

und *Referenz-Datentypen* zu array, class und interface (diese werden als Verweis / Zeiger / Adresse / Referenz auf den Speicherbereich, in dem ihre Komponenten stehen, realisiert).



### 4.1.4.2 Variablen

Variablen werden wie in der imperativen Programmierung als Behälter eingesetzt. Jede Variable speichert einen elementaren Wert oder einen Verweis auf einen Speicherbereich. Variablen müssen vor ihrer Verwendung deklariert werden. Dabei ist ihr Wert entweder undefiniert oder sie können explizit initialisiert werden.

In Java muss jede Variable, bevor sie erstmals gelesen wird, einen Wert erhalten haben. Dies wird bereits zur Übersetzungszeit überprüft, allerdings nur auf oberster Typhierarchie, s. u.

Die Deklaration erfolgt durch Angabe des Datentyps gefolgt von einer Liste der Variablen und eventuell der initialisierenden Zuweisung. Für die Initialisierung von Referenzdatentypen ist der Operator new (oder die Konstante null) zu verwenden.



*Beispiele:*

```
short a, b, c;  char x = 'B';
float r;  float pi = 3.1415926;  double q = -4.443e-13;

dreiecksfolge dd = new dreiecksfolge(7);
dreiecksfolge d = new dreiecksfolge( );

float[ ] zeugnisnoten = new float[11] ;
float[ ] noten = null;
float[ ] zn = {1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.7, 4.0, 5.0};
```

Der new-Operator initialisiert zugleich die Komponenten mit den Standardwerten ("default", siehe 4.1.2.5). Ansonsten wird einer Variablen kein Anfangswert automatisch zugewiesen. Der Wert von noten[1] ist also undefiniert (und dies erkennt der Compiler nicht), während zn[1] gleich 1.3 ist.

Typanpassung:

Java ist im Prinzip streng typisiert. Variablen müssen ein Typ haben und behalten diesen; Operatoren verlangen, dass ihre Argumente den zugehörigen Typ besitzen. Daher muss man in der Regel den Typ eines Wertes explizit anpassen (diese Anpassung nennt man casting):

(<gewünschter Typ>) <Variable>

Beispiel: float a, b; int z; ... a = b + (float) z; ...

In einem Fall erfolgt eine implizite Anpassung in Java, wenn Zahlen von ganzzahligen nach reellwertigen Werten umgewandelt werden und hierbei die Genauigkeit des Wertes nicht verringert wird (int liefert den ganzzahligen Anteil):

int i = 3; int j = 5; float k = 1.5f; double d = 2.5D;

| | |
|---|---|
| i = (int) k (liefert 1) | d = d/j (liefert 0.5D) |
| k = j (liefert 5.0) | i = d/j Fehler, wegen der Genauigkeit |
| d = i/j (liefert 0.0D) | |
| d = ((double)i) /j (liefert 0.6D) | i = (int)d/j (liefert 0) |

*Empfehlung:* Stets explizite Typanpassung verwenden!



4.1.4.3 Felder / arrays

Wenn T ein Datentyp ist, so ist T[ ] der Feld-Datentyp über T. Die einzelnen Komponenten besitzen die Indizes 0, 1, 2, ... . Felder sind dynamisch, d.h., sie werden erst zur Laufzeit angelegt, und die obere Feldgrenze kann daher durch einen Ausdruck angegeben werden (die untere Grenze ist immer 0). Ist die obere Grenze festgelegt, kann man sie nicht mehr verändern.

int t = 12; int v = t+t; ...
float[ ] werte = new float[(t*t-v)/13];

erzeugt somit unter dem Namen "werte" ein Feld von 9 float-Elementen, die mit 0.0 initialisiert sind.



Erzeugen eines Feldes:

- durch Auflisten der Elemente
  int[ ] q = {7, 3, 5, 7, 9} (das 5-elementige Feld (7,3,5,7,9))
- durch den new-Operator
  int[ ] r = new int[<Integer-Ausdruck>];
- durch null (das Feld bleibt zunächst unspezifiziert).

Die Länge (=Anzahl der Elemente) eines Feldes zählt ***nicht*** zum Datentyp array. Es gibt also keine mehrdimensionalen Felder, vielmehr muss man <Datentyp> [ ] [ ] [ ] feld für das gewohnte dreidimensionale Feld schreiben. Andererseits kann man nun auch Dreiecksmatrizen und andere Strukturen definieren. Zum Beispiel:



float [ ] [ ] rechteck_matrix = new float [30] [20];

int [ ] [ ] pascal_dreieck = {{1}, {1,1 }, {1,2,1}, {1,3,3,1},
{1,4,6,4,1}, {1,5,10,10,5,1}, {1,6,15,20,15,15,6,1}};

Für die Variable pascal_dreieck sind also genau die Indizes [0] [0], [1] [0], [1] [1], [2] [0], [2] [1], [2] [2], [3] [0], ... zulässig.

Mit jeder Dimension des array-Datentyps ist stets die Anzahl der Elemente (= Länge) als Attribut verbunden. Man schreibt <Variable>.length.

Zum Beispiel gilt:
rechteck_matrix.length hat den Wert 30,
rechteck_matrix[i].length hat den Wert 20 (für jedes i von 0 bis 29),
pascal_dreieck.length hat den Wert 7,
pascal_dreieck[0].length hat den Wert 1,
pascal_dreieck[3].length hat den Wert 4.

## 4.1.4.4 Anweisungen

Alle Anweisungen dürfen eine Marke besitzen in der Form

<Bezeichner> : <Anweisung>

Typen von Schleifen:

for ([<Initialisierungen>]; [<Bedingung>]; [<Veränderungen>])
    <Anweisung>

while (<Bedingung>) <Anweisung>

do <Anweisung> while (<Bedingung>)

Es gibt also die Laufschleife (for), die while-Schleife (beginnend mit while) und die repeat-Schleife (beginnend mit do und der Wiederholungsbedingung nach while am Ende). Bei der for-Schleife darf die Laufvariable auch neu deklariert werden.



*Beispiele (mit folgender Deklaration):*

```
int [ ] quadrat = new int [20];   int k = 1; int m = 15;

for (int i = 0; i < 20; i++)  quadrat[i] = float(i*i);
```

Ergebnis: Diese Schleife legt 0, 1, 4, 9, ..., 361 im Feld quadrat ab.

```
while (k < 20)  {
        quadrat[k] = quadrat[k] + quadrat[k-1];
        k++;
}
```

Ergebnis: verändert die Werte von quadrat nach einer Art Fibonacci-Schema (stellen Sie fest, welche Werte berechnet werden).

```
do {quadrat[m] = float(3*m); m -=1;} while (m > 20)
```

Ergebnis: verändert die Werte von quadrat[15] in 45 und von m in 14. Die do-while-Schleife wird mindestens einmal durchgeführt.



Lesen Sie sich nochmals die Beschreibung von Schleifen in der Grundvorlesung Abschnitt 2.1.5, A8 durch! Die for-Schleife in Java entspricht der (ziemlich unstrukturierten) allgemeinen for-Schleife (A8d). Wie dort geben wir auch hier ein Äquivalent zur for-Schleife an und empfehlen, die for-Schleife in Java stets nur wie eine Laufschleife zu verwenden.

*Beachte: Das Java-Konstrukt*

```
for (init_1, init_2, ..., init_i; <Bedingung>;
        verändern_1, ..., verändern_k)  <Anweisung>
```

*ist gleichbedeutend mit*

```
{ init_1; init_2; ..., init_i;
  while (<Bedingung>); { <Anweisung>;
        verändern_1; ...; verändern_k}
}
```



Sprachelemente, um den Kontrollfluss zu beeinflussen (hinzu kommt noch return zum Beenden von Methoden siehe 4.1.4.5).

break; beendet die innerste, break umgebende Anweisung switch, for, while oder do-while (ähnlich zu exit in Ada; man verlässt die laufende Anweisung).

continue [<Marke>]; beendet den laufenden Schleifen-durchlauf. continue ist nur innerhalb von Schleifen zulässig. Der Rest des Schleifen-rumpfs wird übersprungen; man bleibt aber innerhalb der Schleife. Wird eine Marke angegeben, so ist die Schleife gemeint, die mit dieser Marke benannt wurde.

*if-Anweisung:* Die ein- und zweiseitige *Alternative* (= if-Anweisung) hat die gewohnte Bedeutung. Der Ausdruck muss einen Wahrheitswert liefern. Die von uns angegebene Syntax in 4.1.3

```
if_statement ::= "if" "(" expression ")" statement [ "else" statement ].
```

ist mehrdeutig, da zum Beispiel für

```
if (x!=0) if (x<0) a=1; else a=2;
```

unklar ist, zu welchem if das else gehört. Prinzipiell legt man hierfür das letzte if fest. Obige Anweisung bedeutet also

```
if (x!=0) { if (x<0) a=1; else a=2; }
```

Kurze Aufgabe: Wäre folgende Syntax eindeutig?

```
"if" "(" expression ")" statement
["else" "if" "(" expression")" statement ]
["else" statement ]
```

Für geschachtelte if-Anweisungen sollte man daher stets { ... } oder notfalls die switch-Anweisung verwenden.



Die switch-Anweisung realisiert *Fallunterscheidungen*. Allerdings entspricht die Semantik nicht der Fallunterscheidung von Ada. Hinter switch muss ein Ausdruck vom Ergebnistyp byte, short, int, char (oder ein selbstdefinierter enum-Typ) stehen. Der Wert dieses Ausdruckes sei x. Nun werden die case-Fälle der Reihe nach durchgegangen, bis der erste Fall für x zutrifft oder bis man auf default trifft. Die zugehörige Anweisung wird ausgeführt und danach alle folgenden Anweisungen (!) in der switch-Anweisung. Will man nur eine Anweisung ausführen, so muss man explizit ein "break" einfügen (break beendet den Rumpf der switch-Anweisung, s. o.). Häufiges Schema:

```
switch (<spezieller_Ausdruck_siehe_oben>) {
        case <Konstante_1>: <Anweisungsfolge_1>
        case <Konstante_2>: <Anweisungsfolge_2>
        ...
        case <Konstante_k>: <Anweisungsfolge_k>
        default: <Anweisungsfolge>
}
```



*Beispiel:* Addiere 1 modulo 4

```
switch (k) {
        case 0:  k = 1; break;
        case 1:  k = 2; break;
        case 2:  k = 3; break;
        default: k = 0; break;
}
```

Umrechnen von Noten in Text

```
switch (note) {
    case 1: System.out.print("sehr ");
    case 2: System.out.print("gut"); break;
    case 3: System.out.print("befriedigend"); break;
    case 4: System.out.print("ausreichend"); break;
    case 5: System.out.print("mangelhaft"); break;
    default: System.out.print("wechseln Sie das Fach"); break;
}
```



*Noch ein Beispiel:*

```
for (int [ ] q = {0, 2, 1, 3}, int k = 0; k < q.length; k++)  {
   switch (q[k])  {
        case 0: System.out.print("Wenn hinter ");
        case 1: System.out.print("\b Fliegen Fliegen");
                if (k=2) break;
        case 2: System.out.print(" fliegen,"); break;
        default: System.out.println(" nach.");
   }
}
```

\b ist der "Backspace", d.h., das Zeichen davor wird gelöscht.

In Java gibt es auch "Blöcke". Syntax:

statement_block ::= "{" { statement } "}"

Anweisungen werden in Java mit einem ";" abgeschlossen, sodass in der Syntax keine Trennzeichen zwischen den Anweisungen explizit auftreten.

Einen Einfluss auf die Speicherverwaltung haben die Blöcke in Java nicht.

Da jedes statement wieder ein Block sein kann, lassen sich Blöcke beliebig schachteln.

Blöcke dienen zum einen dem Zusammenfassen von Anweisungen, zum anderen definieren sie die Lebensdauer bzw. den Sichtbarkeitsbereich von Bezeichnern, vgl. 1.11.5 der Grundvorlesung. Wird zum Beispiel in einem Block eine Variable deklariert, so ist sie sichtbar ab der Deklarationsstelle bis zum Ende des Blockes (das Ende ist durch "}" gegeben). Die Begriffe global und lokal: wie üblich.

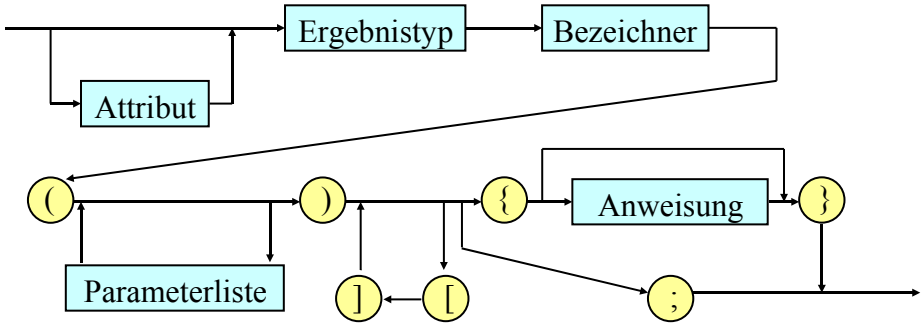
In Java braucht man zwischen Lebensdauer und Sichtbarkeit nicht zu unterscheiden, da es in Java verboten ist, eine Variable in einem Unterblock umzudeklarieren.



## 4.1.4.5 Methoden (Algorithmen/Funktionen)

Die Deklaration einer Methode hat den syntaktischen Aufbau

{ modifier } type identifier "(" [ parameter_list ] ")"
{ "[" "]" } ( statement_block | ";" )






Beispiel Fakultätsfunktion (iterativ berechnet)

```
long fak (int a) {
    if (a = 0) return 1L;
    else {
        long ergebnis = 1;
        for (int i = 1; i <= a; i++)
            ergebnis *= (long) i;
        return ergebnis;
    }
}
```

Beispiel Fakultätsfunktion (rekursiv berechnet)

```
long fak (int a) {
    if (a = 0) return 1L;
    else return (long) a * fak(a-1);
}
```



Beispiele ggT und Maximum

```
int ggT (int a, int b) {
        if (b == 0) return a;
        else return ggT(b, a % b);
}
```

```
int max (int a, int b, int c) {
        return max(a, max(b,c));
}
int max (int x, int y) {
        if (x > y) return x;
        else return y;
}
```

Überladen ist in Java erlaubt, wenn sich die Funktionen in ihren Argumenten in Anzahl und/oder Typ unterscheiden

return (x < y) ? y : x

genügt, vgl. 4.1.2.8

*Beispiel Suchen in einem Feld:* Suche den Schlüssel s in einem long array a in den Grenzen von links bis rechts (siehe Grundvorlesung 6.5.1); zurückgegeben wird der Index bzw. links-1.

```
int n = ...;  ...
long [ ] feld = new long [n]; ...
```

```
int suchen (long [ ] a, int links, int rechts, long s) {
      int mitte;
      while (links <= rechts)  {
            mitte = (links+rechts)/2;
            if (a[mitte] = s)  return mitte;
            else   if (a[mitte] < s)  links = mitte + 1;
                   else rechts = mitte - 1;
      };
      return (links - 1);
}
```

Aufruf im Programm:  suchen(feld,0,n-1,schluessel).



Methoden brauchen keinen Ergebnistyp zu besitzen. In diesem Fall lautet der Ergebnistyp void.

Java überprüft, dass jeder mögliche Zweig in einer Methode auf ein return mit einem Ausdruck des zugehörigen Ergebnistyps stößt. Im Falle des Ergebnistyps void muss entweder das Ende der Methode oder ein parameterloses return erreicht werden.

Weiterhin darf jede Methode Attribute ("modifier") besitzen, vgl. 4.1.1. Diese lauten (Verwendung siehe später oder Literatur):

| | |
|---|---|
| public, protected, private | regeln den Zugriff durch andere |
| static | Klassenmethode |
| abstract | ohne Implementierung |
| final | nicht in Unterklassen veränderbar |
| synchronized | Synchronisation von Prozessen |
| native | Plattformabhängigkeit |



Parameterübergabe in Java

(1) Formale Parameter werden als lokale Variable der Methode aufgefasst. Beim Aufruf ("call") der Methode werden sie mit den Werten der aktuellen Parameter ("Übergabeparameter") initialisiert.

(2) Die Übergabe erfolgt call-by-value.

***call-by-value*** heißt hier:
Ist der Parametertyp primitiv, so wird nur die Wert (nicht aber eine Referenz auf eine Variable) übergeben. Ist der aktuelle Parameter eine solche Variable, so kennt die Methode diese Variable nicht und kann daher ihren Wert auch nicht verändern.
Ist der Parametertyp ein Referenzdatentyp, so wird die Adresse des Objekts übergeben und die Methode kann die Komponenten des Objekts verändern.
(Vgl. Abschnitt 4.2 der Grundvorlesung.)



Aus der Grundvorlesung (Abschnitt 1.7 usw.) übernehmen wir zur Illustration das Beispiel Vertausche.

```
void vertausche (float X, float Y) {
      float H = X;
      X = Y;
      Y = H;
}
int a = 2; int b = 7;
vertausche (a, b);
```

Hier werden nur die Werte 2 und 7 übergeben. Nach dem Methodenaufruf haben die Variablen a und b unverändert die Werte 2 bzw. 7.
Wir betrachten daher nun den Referenzdatentyp "array" als formalen Parameter und vertauschen zwei seiner Komponenten

```
void vertausche (float[ ] X, int i, int j) {
      float H = X[i];
      X[i] = X[j];
      X[j] = H;
}
int [ ] feld = {3, 5, 8, 9};
int a = 1; int b = 3;
vertausche (feld, a, b);
```

Hierdurch wird das array feld = (3, 5, 8, 9) in (3, 9, 8, 5) abgeändert. Man beachte, dass X die Referenz auf das Objekt feld beim Aufruf erhält und dass diese Referenz auch nicht verändert wird, also dem call-by-value-Prinzip unterliegt. Jedoch kann die Methode nun auf die Komponenten des Objekts feld zugreifen, weil dessen Adresse in X steht.



Standardbeispiel Gerade/Ungerade:

```
boolean gerade (int a) {
        if (a == 0) return true;
        else return ungerade(a-1);
}
boolean ungerade (int a) {
        if (a == 0) return false;
        else return gerade(a-1);
}
```

Hinweis: In Java benötigt man keine gesonderte Vorab-Spezifikation wie in Ada. Java ermittelt wie Scheme zuvor alle auftretenden Namen und übersetzt oder interpretiert den Text erst anschließend.



### 4.1.4.5 Einführung in Klassen

In einer Klasse werden Datenstrukturen (mit Selektoren bzw. Regelung des Zugriffs auf Komponenten), Algorithmen (Methoden), erforderliche Variablen und Angaben zur Initialisierung (Konstruktor) in Form eines Schemas zusammengefasst.

Ein Objekt ist eine Instanz (= mit Werten versehene Kopie) einer Klasse. Es wird mittels new erzeugt, wobei ein Speicherbereich und eine Referenz auf den Anfang des Speicherbereichs angelegt werden und die Initialisierung durchgeführt wird.

Ein Java-Programm ist eine Menge von Klassendefinitionen. Eine Klasse muss ausgezeichnet werden (zum Beispiel durch "main"), damit das Programm eindeutig gestartet werden kann.



*Aufbau einer Klasse:*

```
class_declaration  ::=
        {modifier}  "class"  identifier ["<" identifier {"," identifier} ">" ]

        [ "extends" class_name ] [ "implements"  interface_name
        { "," interface_name } ]  "{"  {field_declaration}  "}"  .
```

In Java beginnt ein Klassenname stets mit einem großen Buchstaben.

modifier: sinnvoll sind nur "public" und alternativ "abstract" oder "final".

extends gibt bei Vererbung die vererbende Oberklasse an.

Ein "interface" ist im Wesentlichen das, was man in Ada als Spezifikation, allerdings mit Mehrfachvererbung, bezeichnet. implements bedeutet, dass diese Klasse eine konkrete Implementierung einer oder mehrerer solcher Spezifikationen bildet.

field_declaration ist eine Variablen-, Methoden- oder Konstruktordeklaration.

*Beispiel* (vergleiche 4.6 der Grundvorlesung):

```
class Punkt {
    private float x;
    private float y;
    Punkt () {x=0.0f; y =0.0f;}
    Punkt (float a, float b) {x = a; y = b;}
    float ausgabeX ( ) {return x;}
    float ausgabeY ( ) {return y;}
    String ausgabe ( ) {
        return "X-Koordinate: " + x + ", Y-Koordinate: " + y;
    }
    private boolean diagonal () {return x*x == y*y;}
    void verschieben (float diffx, float diffy)
        { x = x + diffx; y = y + diffy; }
    void verschieben (float v) { verschieben (v, v); }
    void strecken (float s) {x *= s;  y *= s; }
}
```





```
class Kreis {
    private float radius;
    private Punkt mitte;
    Kreis () {einheitsKreis (0.0f, 0.0f, 1.0f);}
    void einheitsKreis (float a, float b, float c)
        {mitte.x = a; mitte.y = b; radius = c; }
    float q (float z) {return z*z;}
    Punkt ausgabePunkt ( ) {return mitte;}
    float ausgabeR ( ) {return radius;}
    String ausgabe ( ) {
        return "Mittelpunkt: (" + mitte.x + ", " + mitte.y + "),
        Radius: " + radius; }
    private boolean aufRand (float a, float b)
      {return q(radius) == q(mitte.x-a)+q(mitte.y-b);}
    void verschieben (float diffx, float diffy)
      {mitte.x += diffx; mitte.y += diffy; }
    void vergroessern (float s) {radius *= s; }
}
```



In der Grundvorlesung haben wir gefordert, dass Programmeinheiten Parameter besitzen dürfen; insbesondere können diese Parameter auch Datentypen oder Algorithmen sein. In Java ist dies zulässig. Man gibt nach dem Klassennamen Bezeichner in spitzen Klammern für Typparameter an, die zunächst nicht festgelegt werden (Generizität). Beispiel: Folge von Elementen:

```
class Folge <Element> {
   Element inhalt;
   Folge<Element> rest;
   Folge<Element> (Element a) {inhalt = a; rest = null;}
   Folge<Element> (Element a, Folge<Element> b)
        {inhalt = a; rest = b;}
}
```

Man sieht zugleich, dass Klassen rekursiv verwendet werden können, um z.B. Listen, Bäume oder Graphen zu definieren.



Ähnlich wie in Ada müssen die konkreten Typen bei Variablendeklarationen angegeben werden. Will man einer Variablen X eine Folge von Zeichenketten zuordnen, so schreibt man

```
Folge <String>  X = new Folge <String> ("Wetter");
```

Dies erzeugt eine Variable vom Typ Folge aus Zeichenketten, die mit der Zeichenkette "Wetter" und rest = null initialisiert wurde. Analog:

```
Folge <Integer>  X = new Folge <Integer> (25);
X = new Folge <Integer> (51, X); ...
```

Als Typen müssen Klassennamen verwendet werden. Die primitiven Typen müssen daher durch die ihnen zugrunde liegenden Klassen ("Hüllenklassen", siehe 4.1.5.2) ersetzt werden. Die Hüllenklasse von int ist Integer, weshalb im Beispiel Integer und nicht int steht.

4.1.4.6 Programmstart mittels main:

Genau eine Klasse des Java-Programms enthält eine Methode der Form

```
public static void main (String[ ] args) { <Rumpf> }
```

oder mit drei Punkten im Parameterteil (ab Version JDK 1.5)

```
public static void main (String ... args) { <Rumpf> }
```

Als Parameter für main dienen sog. Kommandozeilenparameter, die wir zunächst ignorieren. Mit dieser main-Methode wird das Programm vom Java-System gestartet.

Das Standard-Java-Programm lautet:

```
class HelloWorld {
        public static void main (String ... arg) {
         System.out.println("Hello World");
        }
}
```



Dieses Programm muss in einem File mit Namen .../HelloWorld.java abgespeichert sein. Man startet dieses Programm, indem man zunächst den Java-Compiler (javac) hiermit aufruft:

```
javac  HelloWorld.java
```

und anschließend das erzeugte Programm mittels

```
java  HelloWorld
```

ausführen lässt.

Bitte Java-Versionen vom Netz herunterladen, siehe etwa:

```
www.eclipse.org
```

oder

```
www.bluej.org
```

oder

```
java.sun.com
```

oder fragen Sie Ihren Tutor oder Assistenten.



Sie werden sofort eines feststellen: Java besitzt diverse Zusatzregeln. Zum Beispiel muss das Programm genau unter dem Klassennamen, der das Programm beschreibt, abgelegt werden (in der Form <Name>.java), sodann muss man die Kommentare des Compilers bei Fehlern interpretieren lernen, weiterhin muss man beim Ablaufenlassen des Programms oft hinter "java" ein " -classpath " einfügen und darf nun nur noch den Klassennamen (ohne Erweiterung) angeben usw.

Java ist auch nicht für Anfängerprogramme konzipiert, sondern entfaltet seine Mächtigkeit erst, wenn man die bereits vorhandenen Klassen im eigenen Programm einsetzt. Die vordefinierten Klassen und die großen Klassenbibliotheken von Java bilden daher einen riesigen Baukasten, aus dem man ständig immer komplexere Programme zusammenstecken kann.



## 4.1.5 Strings (Zeichenketten) (und Hüllenklassen)

### 4.1.5.1 Character

Die Datentypen char und String hatten wir zwar schon in 4.1.2.1 und 4.1.2.2 besprochen, doch kann man mit den dortigen Informationen noch keine Textverarbeitung durchführen. Beim Typ char muss man klären, wie man diesen Typ durchlaufen kann, also z. B. eine Codierungstabelle der um 3 Zeichen im Alphabet verschobenen Buchstaben aufbaut, die in Ada lauten würde:

```
H: Character array ('a'..'z') of Character;        -- mit der Initialisierung
for i in 'a'..'z' loop
        H(i) := H(Character'Val(Character'Pos(i)+3));
end loop;
```

Weiterhin sind alle zulässigen Operationen anzugeben (4.1.5.2).

*Prinzip:* Der Datentyp char in Java wird mit den natürlichen Zahlen von 0 bis $2^{16}$-1 = 65535 gleichgesetzt. Dies sind die vorzeichenlosen ganzen int-Zahlen, die mit 16 Bit darstellbar sind. Allerdings ist Java stark typisiert, weshalb char und int nicht einfach addiert werden dürfen. Vielmehr muss zuvor stets eine Typumwandlung (casting, 4.1.4.2) erfolgen. Für char-Variablen a und c sind also a=a+1, c++ oder a=*c nicht erlaubt, wohl aber a = (char) ((int) a * (int) c). Beispiel:

```
public class Test_Char1 {
   static int q = 105; static char a = '\u0041'; static char c = (char) 66;
   public static void main (String [] args) {
      System.out.print(q + ", " + (char) q + ", " + a + ", " + c + "; ");
      System.out.print( (char) (9+(int)a) + ", " + (char) ((int) c -10) );
      for (int i = 40; i < 67; i += 2)
         System.out.print(", " + (char)i + " -> " + (char)(i+3));
   }
}
```

liefert:

105, i, A, B; J, 8, ( -> +, * -> -, , -> /, . -> 1, 0 -> 3, 2 -> 5, 4 -> 7, 6 -> 9, 8 -> ;, : -> =, < -> ?, > -> A, @ -> C, B -> E



### 4.1.5.2 Hüllenklassen (wrapper classes)

Die Variablen der primitiven Datentypen speichern ihre Werte unmittelbar, d.h., die zugehörigen Methoden oder Eigenschaften sind nicht zu sehen. Die vollständigen Datentypen (Klassen) existieren natürlich auch. Man nennt sie Hüllenklassen (wrapper classes) und sie können mit folgenden Namen angesprochen werden (beachte: die primitiven Typen beginnen mit einem kleinen, die Klassen mit einem großen Buchstaben):

| Primitiver Datentyp | Zugehörige (vordefinierte) Hüllenklasse |
|---|---|
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |



Hüllenklassen sind erforderlich, wenn man nicht die Werte eines primitiven Datentyps, sondern die Objekte übergeben will/muss oder wenn die zugrunde liegende Klasse gefordert ist (4.1.4.5). In diesen Klassen sind zugleich alle zulässigen Operationen und Attribute gespeichert, zum Beispiel die größte und kleinste darstellbare Zahl eines numerischen Datentyps oder die Umwandlung in ein Objekt einer anderen Klasse (vgl. unten 4.1.5.4).



### 4.1.5.3 Klassen für Zeichenketten

In Ada ist ein String im Wesentlichen ein array aus Zeichen. In Java ist es eine Folge von Zeichen (Zeichenkette, Sequenz). In Java gibt es drei vordefinierte Klassen für Zeichenketten, die voneinander unabhängig, insbesondere also keine Ober- oder Unterklassen voneinander sind:

| | |
|---|---|
| String | Objekte dieser Klasse können nach ihrer Initialisierung nicht mehr verändert werden. *(im Paket java.lang)* |
| StringBuffer | Veränderbare Zeichenketten bzgl. Länge und Inhalt. *(im Paket java.lang)* |
| StringTokenizer | Besitzt zusätzliche Methoden, um eine Zeichenkette in Teile ("token") zu zerlegen. *(im Paket java.util)* |

Methoden der Klasse String:

- String concat (String s): hängt s an den vorhandenen String an. Statt u.concat(v) kann man auch u + v schreiben.
- Konstruktor String (char [] c), der aus einem Feld von Zeichen die Folge der Zeichen als String liefert.
- public int length () liefert die Länge des aktuellen Strings.
- public int indexOf (char b) liefert die Position, an der im aktuellen String zum ersten Mal das Zeichen b steht (oder -1, falls b im String nicht vorkommt).
- public char charAt (int i) liefert das i-te Zeichen des Strings (beachte, dass die Nummerierung von 0 bis length-1 geht).
- public boolean equals (... q) liefert true, falls q ein Objekt der Klasse String ist und den gleichen Text wie der aktuelle String besitzt. Anderenfalls wird false geliefert.



Weitere Methoden der Klasse String:

- boolean startsWith (String s), prüft, ob der aktuelle String mit dem String s beginnt.
- boolean endsWith (String s), prüft, ob der akzuelle String mit dem String s endet..
- String substring (int i, int j), liefert den Teilstring des aktuellen Strings von der Position i bis zur Position j. Falls i = j ist, so wird ein String der Länge 1 geliefert; ist i > j, so wird der leere String zurückgegeben.
- indexOf (String s), liefert den kleinsten Index, ab dem s ein Teilstring des aktuellen Strings ist; falls s nicht im aktuellen String als Teilstring vorkommt, wird -1 zurückgegeben.

Wir werden diese Methoden in 4.2.3 "Erkennen von Teiltexten" benutzen.



Methoden der Klasse StringBuffer:

- StringBuffer append (String s) hängt s an den vorhandenen StringBuffer an.
- StringBuffer insert (int i, String s) fügt den String s ab der Position i in den aktuellen StringBuffer ein. (Der StringBuffer wird also auch um die Länge von s verlängert.)
- void setCharAt (int i, char c) überschreibt das Zeichen, das an der Position i steht, mit dem Zeichen c.
- Konstruktor StringBuffer (String s) initialisiert einen neuen StringBuffer mit der Zeichenkette s.
- Konstruktor StringBuffer ( ) erzeugt einen StringBuffer ohne Inhalt.
- int length ( ) liefert die aktuelle Länge des StringBuffers.
- public int indexOf (char b), public char charAt (int i), public boolean equals (... q) und manche andere Methoden verhalten sich wie die gleich benannten Methoden der Klasse String.



Die Klasse StringTokenizer fasst einen String als Folge von Teilstrings auf, die im ursprünglichen String durch Trennzeichen abgegrenzt sind. Es gibt viele Methoden hierfür, z. B.:

- StringTokenizer (String s) Konstruktor zur Erzeugung einer Folge von Zeichenketten aus s, die durch ' ', '\r', '\n' oder '\t' getrennt sind (diese Trennzeichen treten im erzeugten Objekt nicht mehr auf).
- StringTokenizer (String s, String trenn) Konstruktor zur Erzeugung eines Objekts aus s, wobei die Menge der Trennzeichen genau die in trenn enthaltenen Zeichen sind.
- public String nextToken( ) gibt den Teilstring bis zum nächsten Trennzeichen zurück.

Durch den Konstruktor legt man die Menge der Trennzeichen fest, nach denen ein String zerlegt werden soll; dies lässt sich aber im Laufe der Bearbeitung wieder ändern. Bei Texten könnte man den Punkt als Trennzeichen verwenden und sich dann von Satz zu Satz mittels nextToken( ) voran arbeiten. Zur Analyse von Java-Programmen könnte man die Trennzeichen '{', '}', ';' verwenden usw.

### 4.1.5.4 Umwandeln in Strings ("toString" und "valueOf")

Jede Klasse K sollte sich selbst präsentieren können oder jedes Objekt sollte seinen Zustand in lesbarer Form ausgeben können. Daher findet sich in jeder gängigen Java-Klasse eine aus der Klasse Object vererbte oder umdefinierte Methode der Form toString().

Beispiel: In der (Hüllen-) Klasse Integer könnte es folgende Methode toString geben, die allerdings nicht korrekt arbeit (warum?):

```
static String toString (int x) {
   if (x < 0) return "-" + natToString(-x);
   else return natToString(x);
}
static String natToString (int x) {
   if (x == 0) return "";
   else return natToString (x/10) + (x % 10);
}
```

(*Hinweis:* Um dies zu testen, füge man zum Beispiel

```
public static void main (String[] args) {
   int z = 489125; int y = -9888678; String S = toString(286*5041);
   System.out.print(S + "  " + toString(z) + "  " + toString(y));
}
```

hinzu, lege dies alles in eine Klasse, übersetze sie und führe sie aus.)



Die (Hüllen-) Klasse Integer enthält also unter anderem:

- einen Konstruktor Integer (int z), um ein Zahl-Objekt mit dem Wert z (in 32-Bit-Darstellung) zu initialisieren,
- einen Konstruktor Integer (String s), um ein Objekt mit der Zahl, die durch den Text s dargestellt ist, in 32-Bit-Darstellung zu initialisieren,
- eine Klassenmethode String toString (int x), um eine int-Zahl in einen Text umzuwandeln,
- eine Klassenmethode Integer valueOf (String s), um aus dem Text s das zugehörige Integer-Objekt zu erhalten.

usw. Dies gilt analog auch für andere Klassen. Beispiel:

```
String langzahl = Long.toString(1234567890987654321);
Long z = Integer.valueOf (langzahl);
```

(Bitte selbst in Handbüchern weiter recherchieren.)



## 4.1.6 Klassenhierarchie, Vererbung

**Dieser Abschnitt wurde recht knapp in der Vorlesung erläutert. Lesen Sie die Inhalte bitte in Lehrbüchern oder im Internet nach!**

4.1.6.1 Vererbung (extends, this, super)

4.1.6.2 Die Klasse Object

4.1.6.3 Weitere Klassen (System, Bibliotheken)

4.1.6.4 abstrakte Klassen (abstract)

4.1.6.5 interface (implements)



## 4.1.7 Korrektheit (Zusicherungen und Ausnahmen)

### 4.1.7.1 Zusicherungen (siehe 7.1.5 Grundvorlesung)

Eine Zusicherung in einem Programm ist eine prädikatenlogische Formel (oder eine Bedingung), die genau an dieser Stelle erfüllt sein muss. In Java schreibt man hierfür

assert < Bedingung > [: <Text>];

*Beispiel:*

```
int ggT (int a, int b) {
   assert (b > 0) : "zweiter Parameter ist nicht positiv";
   ...  // weiter programmieren wie üblich
}
```

*Bedeutung:* Trifft die Bedingung zu, wird die nächste Anweisung durchgeführt; trifft sie aber nicht zu, wird der Text (falls er fehlt: eine Standardsystemmeldung) ausgegeben.

"assert" dient vor allem der korrekten Programmentwicklung: Man streut ins Programm genau die Bedingungen ein, die hier entsprechend der Semantik gelten müssen. Diese lässt man auch im fertigen Programm stehen; sie belasten die Laufzeit nicht, da man alle Zusicherungen beim Programmstart mittels "-da" ausschalten, aber bei fehlerhaftem Verhalten mittels "-ea" für erneutes Testen wieder einschalten kann.

Zusicherungen sind vor allem wichtig, wenn die Werte eines Objekts oder aller Objekte einer Klasse Nebenbedingungen erfüllen müssen oder wenn man sicher sein will, dass eine Methode am Ende ihrer Berechnung auch tatsächlich das geforderte Ergebnis liefert. Zugleich führt man durch die Zusicherungen auch (wenigstens in Ansätzen) einen Korrektsheitsbeweis mit sich, wie dies in der Grundvorlesung (in Kapitel 7) behandelt wurde.



## 4.1.7.2 Ausnahmen ("exception", siehe 4.1.8 Grundvorlesung)

Die Sicht in Java ist die Folgende: Wenn man in einem Programmteil auf eine Ausnahme (= ein Ereignis, das den normalen Programmablauf stört) treffen könnte, so schließt man diesen Programmteil in

```
try { ...}
```

ein ("try-Block"). Die möglichen Ausnahmen können anschließend in catch-Blöcken abgefangen werden. Schema hierfür (der finally-Block darf nur folgen, falls k > 0 ist):

```
try {
   ... if (<Bedingung>)
      throw new <Ausnahmekonstruktor>(<Parameterliste>);
   ...}
catch (<Ausnahmetyp1> <Bezeichner1>)
   {<Anweisungsfolge zur Behandlung der Ausnahme1>}
catch (<Ausnahmetyp2> <Bezeichner2>)
   {<Anweisungsfolge zur Behandlung der Ausnahme2>}
...
catch (<Ausnahmetyp_k> <Bezeichner_k>)
   {<Anweisungsfolge zur Behandlung der Ausnahme_k>}
[finally {<Anweisungsfolge>}]
```



Bedeutung dieser Konstruktion:
Eine Ausnahme wird in Java als Objekt aufgefasst, die von Ereignissen (Situationen, Bedingungen) "geworfen" wird. Dies kann zum einen durch das Programm selbst geschehen, zum andern kann das Java-Laufzeitsystem der Auslöser (Division durch 0, Überschreiten von Indexgrenzen, Speicher knapp, ...) sein.
Das geworfene Ausnahmeobjekt wird dabei in der Regel neu erzeugt, daher steht meist "new" vor der Ausnahme. Dem Ausnahmeobjekt können (entsprechend dem Konstruktor in der zugehörigen Klasse) aktuelle Parameter mitgegeben werden.
Daraus folgt: Eine Ausnahme muss irgendwo als Klasse deklariert werden und an der Stelle, wo sie geworfen wird, sichtbar sein.
Die üblichen Ausnahmen sind vordefiniert (wie in Ada) und werden, falls sie eintreten, automatisch bei der Ausführung des Programms vom Laufzeitsystem "geworfen". Man kann sie in einem catch-Block "fangen" und auf diese Weise die vordefinierte Methode abändern und die konkrete Ausnahmebehandlung selbst festlegen. Andere Ausnahmen, die man selbst neu einführt, muss man als Klasse definieren.

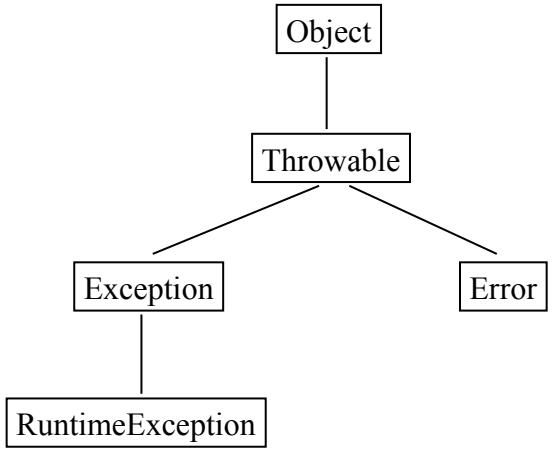


Das geworfene Ausnahmeobjekt muss nun "gefangen" werden. Dies erledigt das Java-Laufzeitsystem, es sei denn, wir deklarieren den Block, in dem die Ausnahme stattfinden kann, als "try-Block" und fügen anschließend catch-Blöcke für gewisse Ausnahmenan. Hier wird dann eine eigene Ausnahmebehandlung durchgeführt. Dadurch wird die Kontrolle dann an eine aufzurufenden Methode (oder mittels return an die Stelle, wo die Ausnahme stattfand) weitergegeben.

Im Anschluss an den try- oder einen catch-Block-Bereich wird auf jeden Fall der finally-Block ausgeführt. Dies dient vor allem dazu, irgendwelche letzten "Aufräumarbeiten" noch ausführen zu können, bevor das Programm normal beendet oder abgebrochen wird.

Empfehlung: Probieren Sie diese Konstruktionen an selbst gewählten Beispielen aus. Machen Sie sich kundig, welche Ausnahmen in Java vordefiniert sind.

Eine Ausnahme-Klasse wird in Java als Unterklasse zur Klasse "Exception" definiert, die wiederum eine Unterklasse der Klasse "Throwable" (= "kann geworfen werden") ist. Die Klassenhierarchie lautet:




Auf die Klasse "Error", in der Fehler, die zum Programmabbruch führen sollten, behandelt werden, gehen wir hier nicht ein.



*Künstliches Beispiel: "AusnahmeTest"*

Suche, ob der ggT von a und b im Feld zahlen enthalten ist. Hierzu sollen die Ausnahmen, dass man den ggT wegen b ≤ 0 nicht bilden möchte (Ausnahme: NichtPositiv) und dass eine Zahl nicht im Feld zahlen vorkommt (Ausnahme: NichtGefunden), verwendet werden. Analysieren Sie das Beispielprogramm und machen Sie sich die anschließend aufgelistete Ausgabe klar.

```
class AusnahmeTest {  // Layout wegen Platzproblemen nur z.T. vorschriftsmäßig.

   static class NichtPositiv extends Exception {
      NichtPositiv (int j) {
         System.out.println ("Zahl " + j + " ist nicht positiv. Abbruch. ");
      }
   }

   static class NichtGefunden extends Exception {
      NichtGefunden ( ) { return; }
     NichtGefunden (int d) {System.out.println(d);}
   }
```



```
   public static void main (String [] args) {
      int anfangA = 48; int b = 30; int endeB = 36;
      int x, y, r, i;   int [] zahlen = {1, 2, 4, 7, 9, 10, 12};
      while (b <= endeB) {
         x = anfangA; y = b;
         try {
            if (y <= 0) throw new NichtPositiv (y);
            System.out.print("Es geht los mit " + x + " und " + y);
            while (y != 0) {r = x%y; x = y; y = r;}
            System.out.println("; ihr ggT lautet " + x + ".");
            i = 0;   while ((i < zahlen.length) && (zahlen[i] != x)) i++;
            if (i == zahlen.length) throw new NichtGefunden (x);
            System.out.println("Index: " + i + ".");
         }
         catch (NichtPositiv e1) {
         System.out.println ("Der ggT von " + x + " und " + y + " wurde nicht berechnet.");
         }
         catch (NichtGefunden e) {
         System.out.println ("Der Wert der Exception " + e + " kommt im Feld nicht vor. ");
         }
         finally { b ++;
            if (b > endeB) System.out.println ("Ende.");
            else System.out.println("Neues b = " + b + ".");
   } } }
```



```
// Ausgabe (mit zahlen = {1, 2, 4, 7, 9, 10, 12}):
Es geht los mit 48 und 30; ihr ggT lautet 6.
6
Der Wert der Exception Ausnahme$NichtGefunden kommt im Feld nicht vor.
Neues b = 31.
Es geht los mit 48 und 31; ihr ggT lautet 1.
Index: 0.
Neues b = 32.
Es geht los mit 48 und 32; ihr ggT lautet 16.
16
Der Wert der Exception Ausnahme$NichtGefunden kommt im Feld nicht vor.
Neues b = 33.
Es geht los mit 48 und 33; ihr ggT lautet 3.
3
Der Wert der Exception Ausnahme$NichtGefunden kommt im Feld nicht vor.
Neues b = 34.
Es geht los mit 48 und 34; ihr ggT lautet 2.
Index: 1.
Neues b = 35.
Es geht los mit 48 und 35; ihr ggT lautet 1.
Index: 0.
Neues b = 36.
Es geht los mit 48 und 36; ihr ggT lautet 12.
Index: 6.
Ende.
```

Das Beispiel zeigt: Man kann Ausnahmen wie Abweichungen vom gewünschten Programmablauf behandeln und umgeht dadurch unübersichtliche if-else-Strukturen.

Hinweis: Wenn finally verwendet wird, muss zuvor mindestens eine catch-Anweisung stehen.

Beachte: Der finally-Block wird auf jeden Fall durchgeführt, auch wenn der try-Block ohne eine Ausnahme durchlaufen wurde. Hier kann man zum Beispiel ein "return" hinschreiben, wenn der try-Block selbst in einer gerufenen Methode steht, usw.



Da gibt es noch ein Problem: Die Ausnahme muss ja nicht in der jeweiligen Anweisung selbst erfolgen, sondern sie könnte in einem gerufenen Unterprogramm stattfinden. Jene Methode muss dann aber die zuständige Ausnahmebehandlung kennen. Um dies sicherzustellen, muss man die Ausnahmen, die möglicherweise von der gerufenen Methode geworfen werden müssen oder können, in ihrem Kopf (unmittelbar hinter der Parameterliste) nach dem Schlüsselwort "throws" angeben:

public void machen (int i) throws heute, gestern { ... }

bedeutet also, dass im Methodenrumpf { ... } die Ausnahmen "heute" und "gestern" vorkommen und möglicherweise geworfen werden und die Methode "machen" daher die entsprechenden Aufrufe vorsehen muss.



## 4.1.8 Threads (leichtgewichtige Prozesse)

Viele Programmteile lassen sich eine gewisse Zeit lang unabhängig von einander ausführen. Man verteilt sie dann auf verschiedene Prozessoren und startet sie dort. Hin und wieder müssen Synchronisationen durchgeführt werden, weil zum Beispiel Daten von einem Prozess an einen anderen zu übergeben sind; siehe Kapitel 1 dieser Vorlesung.

Solche Programmteile, die nebenläufig zueinander ablaufen können, nennt man "Thread" (engl.: Faden). Der Ablauf des Gesamtprogramms wird als eine Linie aufgefasst, die sich an gewissen Stellen in diverse Fäden aufspaltet, die später wieder zusammenlaufen (können), bzw.: Vom Hauptprozess zweigen diverse Seitenprozesse ab, die miteinander direkt oder über gemeinsame Variable kommunizieren können.



Wir demonstrieren dies an dem schon mehrfach benutzten Erzeuger-Verbraucher-System, siehe 1.1.3, 1.2.13 und 1.3.3.

*Zur Erläuterung:* Es gibt eine vordefinierte Klasse "Thread", die unter anderem die (statischen) Methoden "start" (ein thread wird gestartet), "run" (ein thread läuft ab), "sleep" (ein thread wartet), "yield" (Kontrolle an andere threads abgeben) besitzt. Um bei der Ausführung einer Methode oder einzelner Anweisungen nicht gestört zu werden, verwendet man das Schlüsselwort synchronized. Zum Beispiel bedeutet

synchronized (<Objekt A>) {<Anweisungsfolge>}

dass das Objekt A, während die <Anweisungsfolge> abläuft, für jeden Zugriff durch andere Objekte gesperrt ist. Im Falle

synchronized <Methode>

ist das Objekt zu dem Zeitraum, in dem seine <Methode> ausgeführt wird, gegen jeden Zugriff von außen abgeschirmt.

*Beispiel:* Erzeuger-Verbraucher-System. Zuerst das Lager, wobei wir Lagerinhalt[0] nicht, sondern nur Lagerinhalt[1] bis Lagerinhalt[max] verwenden:

```
class Lager {
   int Anzahl, Lagerkapazitaet;
   Lager(int max) {Anzahl = 0; Lagerkapazitaet = max;}
   long [] LagerInhalt = new long [Lagerkapazitaet+1];
   boolean istLeer( ) {return Anzahl == 0;}
   boolean istVoll( ) {return Anzahl >= Lagerkapazitaet;}
   synchronized void speichern (long a) {
      if (! istVoll( )) {Anzahl ++; Lagerinhalt[Anzahl] = a;}
   }
   synchronized long entnehmen ( ) {
      if (! istLeer( )) {
         long x = Lagerinhalt[Anzahl]; Anzahl --; return x; }
   }
}
```



Erzeuger (dieser arbeitet unendlich lange weiter; immer wenn das als Parameter an "meinLager" übergebene Lager nicht voll ist, kann er ein Element vom Typ long dort einspeichern):

```
class Erzeuger extends Thread {
   Lager  meinLager;       //Referenz auf ein Objekt vom Typ Lager
   Erzeuger (Lager x ) {meinLager = x}
   public void run ( ) {
      for ( ; true ; )
         if (! meinLager.istVoll( )) erzeuge_und_speichere( );
         else yield( );
   }
   public void erzeuge_und_speichere( ) {
      long erzeugt = <hier problemspezifisch etwas einsetzen>;
      meinLager.speichern(erzeugt);
   }
}
```



Verbraucher (dieser arbeitet unendlich lange; immer wenn das als Parameter an "meinLager" übergebene Lager nicht leer ist, kann er ein Element vom Typ long dort entnehmen):

```
class Verbraucher extends Thread {
   Lager  meinLager;
   Verbraucher (Lager x) {meinLager = x}
   public void run ( ) {
      for ( ; true ; )
         if (! meinLager.istLeer( )) konsumieren( );
         else yield( );
   }
   public void konsumieren( ) {
      long product = meinLager.entnehmen(erzeugt);
      <hier problemspezifisch "product" verarbeiten>
   }
}
```



Nun fehlt noch eine Testumgebung für das Erz.-Verbr.-System:

```
class TestEVS {
   public static void main (String [] args) {
      Lager schuppen = new Lager(10);
      Erzeuger firma1 = new Erzeuger(schuppen);
      Erzeuger firma2 = new Erzeuger(schuppen);
      Verbraucher mensch1 = new Verbraucher(schuppen);
      Verbraucher mensch2 = new Verbraucher(schuppen);
      firma1.start( );  firma2.start( );
      mensch1.start( );  mensch2.start( );
      < Füge noch Anweisungen für Ausgabe und Terminieren hinzu>
   }
}
```

Dieses System endet nicht. Man muss daher in die Klassen und in TestEVS noch Terminierungsmöglichkeiten einbauen.

4.1.9 Weitere Hinweise zu Java

**Dieser Abschnitt kan nicht mehr behandelt werden. Lesen Sie die Inhalte bitte in Lehrbüchern oder im Internet nach!**

4.1.9.1 Pakete

4.1.9.2 Style Guides
(Auf die Wichtigkeit, sich im Layout, bei der Wahl von Bezeichnern, bei den Kommentaren usw. an Richtlinien zu halten, wurde in der Vorlesung und in den Übungen nachdrücklich hingewiesen. ...)

4.1.9.3 Zufallszahlen (vgl. 4.2.1)

usw.



# 4.2 Beispiele

4.2.1 Zufällige Erzeugung beliebig großer Felder und Sortieren durch Mischen

Zufallszahlen (besser "Pseudozufallszahlen") werden durch
```
import java.util.*;
```
einem Programm zugänglich. Dort ist die Klasse Random definiert. Man erzeugt sich zunächst ein neues Objekt r dieser Klasse mit irgendeinem Anfangswert, zum Beispiel mit 51
Random r = new Random(51), und erhält dann mittels r.nextInt() die jeweils nächste ganzzahlige Zufallszahl.
Da man bei jedem Programmlauf nicht immer die gleiche Folge verwenden möchte, sollte man einen zufälligen Startwert wählen, typischerweise die aktuelle Systemzeit
System.currentTimeMillis(), also:
Random r = new Random(System.currentTimeMillis());
Die nächste Zahl transformiert man in das Intervall $0 \le x < max$ durch Math.abs(r.nextInt()) % max , wobei man die Absolutfunktion abs aus der ebenfalls vordefinierten Klasse Math verwendet.



Verwendung in einer Methode zur Erzeugung eines Felds von ganzzahligen Zufallszahlen:

```
public int[] erzeugeZufallsfeld (int anzahl, int max) {
   int[] a = new int [anzahl];
   Random r = new Random(System.currentTimeMillis());
   for (int m = 0; m < anzahl; m++)
      a[m] = Math.abs(r.nextInt( )) % max;
                                  //Zufallszahl von 0 bis Max-1
   return a;                      //Math.abs = Absolutbetrag
}
```

Diese Methode kann z. B. wie folgt benutzt werden, um ein Feld von 150 Zahlen zwischen 0 und 1999 zu erzeugen:

```
int n = 150;
int[] Z = new int [n];
Z = erzeugeZufallsfeld (n, 2000);
```



Sortieren durch Mischen.

Man schreibe ein Java-Programm, welches ein Feld ganzer Zahlen durch Mischen sortiert.

Hierbei greift man auf das Verschmelzen von bereits sortierten Feldern zurück, siehe Grundvorlesung "Einführung in die Informatik II" (Sommersemester 2006, Abschnitt 10.5.3). Die Anzahl der Vergleiche ist auch im schlechtesten Fall durch $n \cdot \log(n)$ nach oben beschränkt. Wir erläutern das Verfahren nicht noch einmal, sondern übertragen den Algorithmus aus der Grundvorlesung direkt nach Java.

Zunächst geben wir ein zu sortierendes Feld mit 15 Zahlen fest vor. Anschließend wird die Veränderung beschrieben, die nötig ist, um ein Feld aus n Zufallszahlen zwischen 0 und max-1 zu erzeugen.

```
import java.util.*;            // notwendig für das Objekt Random, s.u.
/**
 * VC, Uni Stuttgart, 22.1.07, siehe auch Übungsblatt 12, Aufgabe 2,
 * vergleiche Grundvorlesung Sommersemester 2006, Abschnitt 10.5.3
 **/

public class Sort {
   /**
    * In main die Zahlenfolge festlegen, „sortieren“ aufrufen, ausdrucken.
    * Ungünstigen Fall wählen: 2^4-1 = 15 Elemente im Feld.
    **/
   public static void main (String [] args) {
      int[] Z = {3, 6, 1, 23, 39, 64, 5, 22, 7, 2, 23, 17, 8, 7, 10};
      sortieren_durch_Mischen (Z, 0, Z.length-1);
      for (int m = 0; m < Z.length; m++)
         System.out.println (((m < 10) ? " " : "") + m + " :  " + Z[m]);
   }
```



```
   /**
    * Das Zahlenfeld A[links..rechts] nach der Größe aufsteigend sortieren.
    **/
   public static void sortieren_durch_Mischen (int [] A, int links, int rechts) {
      if (rechts <= links) return;
      else {            //rekursiv absteigen und dann bottom up verschmelzen
         int mitte = (links + rechts)/2;
         sortieren_durch_Mischen (A, links, mitte);
         sortieren_durch_Mischen (A, mitte+1, rechts);
         verschmelze (A, links, mitte, rechts);
      }
   }

   /**
    * Verschmelze zwei sortierte Folgen A[links..mitte] und
A[mitte+1..rechts]
    * zu einer sortierten Folge B[0..rechts-links+1]; anschließend B
    * zurückspeichern nach A
    **/
```



```
   private static void verschmelze (int [] A, int links, int mitte, int rechts) {
      assert (links <= mitte) && (mitte < rechts);
      int [] B = new int [rechts-links+1];
      int i = links;
      int j = mitte+1;
      int k = 0;                              // stets das kleinere Element nach B
      while ((i <= mitte) && (j <= rechts)) {
         if (A[i] <= A[j]) {B[k] = A[i]; i ++;}
         else {B[k] = A[j]; j ++;}
         k ++;
      }                  // restliche Elemente des verbliebenen Feldes anhängen
      if (i > mitte)
         while (j <= rechts) { B[k] = A[j]; j ++; k ++; }
      else
         while (i <= mitte) { B[k] = A[i]; i ++; k ++; }
      for (int m = 0; m < k; m++)  A[links+m] = B[m];       //zurück nach A
      return;
   }
}
```



Ausgabe des obigen Programms (das Feld Z = {3, 6, 1, 23, 39, 64, 5, 22, 7, 2, 23, 17, 8, 7, 10} wurde dort fest vorgegeben):

```
 0 :  1
 1 :  2
 2 :  3
 3 :  5
 4 :  6
 5 :  7
 6 :  7
 7 :  8
 8 :  10
 9 :  17
10 :  22
11 :  23
12 :  23
13 :  39
14 :  64
```

Wir übernehmen nun obige Methode
int [] erzeugeZufallsfeld (int anzahl, int max),
um das Programm mit beliebigen Feldern
testen zu können.

Füge also die (statische) Methode

```
static int[] erzeugeZufallsfeld (int anzahl, int max) {
    int[] a = new int [anzahl];
    Random r = new Random(System.currentTimeMillis());
    for (int m = 0; m < anzahl; m++)
       a[m] = Math.abs(r.nextInt( )) % max;
                                   //Zufallszahl von 0 bis max-1
    return a;                      //Math.abs = Absolutbetrag
}
```

hinzu und ersetze die Zeile

```
int[] Z = {3, 6, 1, 23, 39, 64, 5, 22, 7, 2, 23, 17, 8, 7, 10};
```

durch ein zufällig erzeugtes Feld mit n Elementen:

```
int n = 47;
int[] Z = new int [n];
Z = erzeugeZufallsfeld (n, 2000);
```



so erhält man für n = 47 zum Beispiel folgende Ausgabe (mit modifiziertem Druckbild, da hier 5 Ergebnisse in einer Zeile stehen):

| | | | | |
|---|---|---|---|---|
| 0 : 39 | 1 : 40 | 2 : 67 | 3 : 118 | 4 : 132 |
| 5 : 153 | 6 : 221 | 7 : 222 | 8 : 225 | 9 : 262 |
| 10 : 263 | 11 : 268 | 12 : 337 | 13 : 349 | 14 : 411 |
| 15 : 415 | 16 : 426 | 17 : 504 | 18 : 527 | 19 : 539 |
| 20 : 558 | 21 : 645 | 22 : 700 | 23 : 725 | 24 : 841 |
| 25 : 848 | 26 : 877 | 27 : 889 | 28 : 899 | 29 : 903 |
| 30 : 928 | 31 : 974 | 32 : 1004 | 33 : 1157 | 34 : 1182 |
| 35 : 1212 | 36 : 1299 | 37 : 1307 | 38 : 1452 | 39 : 1470 |
| 40 : 1483 | 41 : 1508 | 42 : 1621 | 43 : 1641 | 44 : 1697 |
| 45 : 1793 | 46 : 1922 | | | |



## 4.2.2 Median in linearer Zeit

4.2.2.1 Aufgabenstellung:

Finde zu n (long-) Zahlen den Median; dies ist das Element, das nach dem Sortieren an der Stelle $(n+1)/2$ steht.

Definition: "Sort(k)"

Gegeben ist eine Folge von n ganzen Zahlen $A_1, A_2, ..., A_n$. Das Element, das nach dem Sortieren an der Position k steht, bezeichnen wir mit Sort(k) der Folge $A_i$.

Eindeutige Charakterisierung des Wertes Sort(k):

$|\{ j \mid A_j \le \text{Sort(k)} \}| \ge k$ und zugleich $|\{ j \mid A_j > \text{Sort(k)} \}| \le n-k$.

Anmerkung: Sind alle $A_i$ paarweise verschieden, so gilt statt "≥ k" und "≤ n-k" die Gleichheit "= k" und "= n-k"

*Es gilt:* Median = Sort( (n+1)/2 ).



Zur Illustration: 145 Zahlen. Welches ist das Element Nr. 73?

2301, 4892, 8197, 7823, 6541, 2639, 7891, 6883, 9211, 6738, 3371, 10892, 4394, 13823, 11741, 2663, 4852, 3197, 7623, 7841, 6383, 10512, 6938, 4092, 8144, 7823, 6741, 2639, 7391, 6884, 9291, 6735, 5171, 10892, 4994, 13623, 12742, 2662, 4432, 3857, 5623, 10395, 2394, 1823, 1751, 2263, 4152, 3647, 7635, 7741, 6383, 1022, 6938, 4992, 8744, 4823, 6641, 7739, 5191, 6294, 4971, 7035, 6631, 11542, 4794, 1373, 15542, 2362, 4412, 3707, 5323, 5371, 4892, 4294, 1373, 11940, 2664, 4252, 3737, 7913, 7221, 6373, 11512, 6928, 4492, 2144, 7433, 6641, 12799, 7341, 6284, 9201, 4735, 5441, 10852, 4984, 12223, 11741, 2632, 2432, 3657, 5629, 10355, 4394, 1823, 1751, 7263, 4452, 6647, 8645, 7641, 6383, 1322, 3938, 4022, 8441, 4323, 6941, 7832, 5121, 6354, 4931, 7235, 6431, 9542, 1794, 3273, 4542, 2662, 4812, 2707, 8323, 6484, 9251, 3795, 5071, 6362, 4812, 2747, 5422, 5371, 1592, 4294, 2723, 6242.

## 4.2.2.2 Verfahren, um Sort(k) für n Elemente zu bestimmen

Vorgehen 1: Sortiere die Folge und nimm anschließend das Element an der Stelle k.
Zeitaufwand hierfür $O(n \cdot \log(n) + n) = O(n \cdot \log(n))$.
(Den Median von 1 Billionen Elementen zu finden, erfordert dann ungefähr 40 Billionen Vergleiche.)

Vorgehen 2: Führe einen Teilsortierschritt durch und mache dann mit dem Bereich, in dem der Median liegen muss, rekursiv weiter. Dies ist nur im Mittel ein O(n)-Verfahren; im schlimmsten Fall dauert es $O(n^2)$ Schritte (wie bei Quicksort).

Vorgehen 3: Berechne den Median der Fünfer-Mediane, führe mit diesem Wert einen Teilsortierschritt durch und mache mit dem Bereich, in dem der Median liegen muss, genau so weiter. Dies erfordert nur O(n) Schritte, allerdings ist die Konstante relativ groß, sodass es sich nur für sehr große n lohnt.



## 4.2.2.3 Wir betrachten das Verfahren 2.

*Was ist ein Teilsortierschritt (= "Quicksortschritt") ?*

Teilsortierschritt: Ergebnis sind 2 Teilfolgen, deren Elemente alle kleiner gleich oder alle größer gleich p sind.

```
public void Teilsortier (long [] A, int L, int R) {
long p, hilf; int i = L; int j = R;
if (i < j) {
 p := A[(i+j)/2];
 while (i <= j) {
   while (A[i] < p) i ++;
   while (A[j] > p) j --;
   if (i <= j) { hilf = A[i]; A[i] = A[j]; A[j] = hilf; i++; j--; }
  }
 }
}      // suche weiter in A[L..j] oder in A[j+1..R]
```



Das Vorgehen ist nun offensichtlich:

Es sei n = A.length die Zahl der Elemente, für die Sort(k) bestimmt werden soll, $1 \le k \le n$. Es sei *min* eine kleine Zahl, bis zu der es einfacher ist, das k-te Element direkt zu bestimmen (in der Praxis: *min* = 5).

Falls höchstens *min* Elemente vorliegen (also $n \le min$), so berechne man Sort(k) direkt, anderenfalls führe man einen Teilsortierschritt durch; hierbei wird das Feld A in ein Feld mit h (= j - L + 1) Elementen und in ein Feld mit (n-h) Elementen zerlegt. Falls $k \le h$ ist, so suche man im ersten Feld nach dem k-ten Element, anderenfalls im zweiten Feld nach dem (k-h)-ten Element.

Zur Implementierung: Wir wählen alle Variablen für die Berechnung global. Anfangs ist L=0 und R=A.length-1.



```
public class Sortk {                          // k, min, A werden fest gewählt
  static int min = 5; static int L, R, k;
  static long [] A;
  static void iteriereTeilsortieren() {       // k, A, L und R sind hier global
    long p, hilf;  int h;  int i = L;  int j = R;
    if (R - L > min)  {
      p = A[(i+j)/2];
      while (i <= j) {
        while (A[i] < p) i ++;
        while (A[j] > p) j --;
        if (i <= j) { hilf = A[i]; A[i] = A[j]; A[j] = hilf; i++; j--; }
      }
      h = j - L + 1;
      if (h < k) {k = k - h; L = j + 1;} else R = j;
      iteriereTeilsortieren();
    }
    else direktFinden();
  }
```

```
  static void direktFinden() {              // k, A, L und R sind global
     long m;                                // Bubblesort
     for (int i = L; i < R; i++)
        for (int j = i+1; j <= R; j++)
           if (A[j-1] > A[j]) {m=A[j]; A[j]=A[j-1]; A[j-1]=m;}
     if ((L <= R) && (1 <= k) && (k <= R-L+1))  {
        System.out.println("Das gesuchte Element ist "+A[L+k-1]+".");
        }
     else System.out.println("Falsche Eingabewerte.");
  }
  public static void main (String [] args) {
     long [] B = {23, 10, 15, 12, 1, 25, 9, 2, 24, 13, 16, 19, 28, 6, 8,
                  27, 3, 4, 21, 0, 26, 11, 18, 14, 17, 20, 7, 30, 29, 22, 5};
     A = B; L = 0; R = A.length-1; k = (A.length+1)/2;
     System.out.println(A.length+" Elemente, gesucht ist das "+k+"-te.");
     iteriereTeilsortieren ();
  }
}
```

Alle Initialisierungen wurden hier in die Methode main verlagert.



### 4.2.2.4 Verfahren 3: "Median der Fünfer-Mediane".

Ein schnelles rekursives Verfahren, um Sort(k) zu bestimmen:

Man iteriert Teilsortierschritte, wählt aber als Element p stets ein Element, welches garantiert, dass die Teilfolge, die weiter untersucht werden muss, höchstens a·n Elemente besitzt mit $0 < a \leq 7/10$.

Dies erreicht man, indem man die mittleren Elemente von Teilfolgen der Länge 5 nimmt und deren Median bestimmt.

Es folgt die genaue Beschreibung des Vorgehens:



Der Kopf der Methode, um Sort(k) zu berechnen, sei

public long **fuenfMed**(long [] A, int k).

A[links..rechts] sei ein Feld mit m=rechts-links+1 Elementen.

Falls m <= 5 ist, dann bestimme das k-te Element direkt.

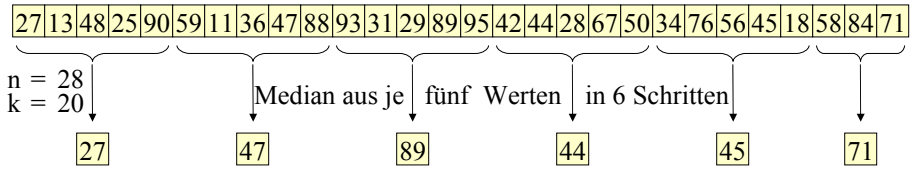
Anderenfalls bilde B[0..(m-1)/5] aus A, wobei B[i] das mittlere Element der fünf Elemente A[links+5*i],...,A[links+5*i+4] für i = 0, ..., (m-1)/5-1 und B[m/5] das mittlere Element der Elemente A[links+5*(m-1)/5)],...,A[rechts] ist. Rufe rekursiv für anzahl = (m-1)/5 +1 auf:

p = **fuenfMed** (B, (anzahl+1)/2)

Dies liefert den "Median der Fünfer-Mediane". Nun führe mit diesem p einen Teilsortierschritt durch und rufe dann rekursiv **fuenfMed** mit dem Teil des Feldes auf, in dem Sort(k) liegen muss.

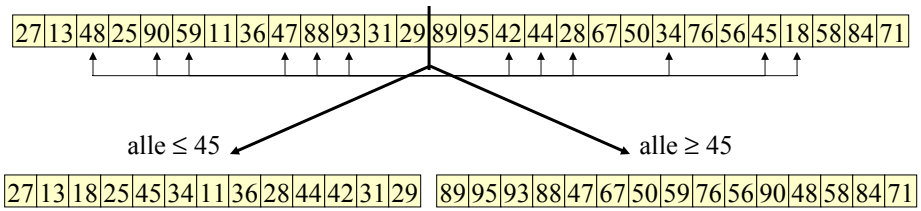


Gegeben ist eine ungeordnete Folge mit 28 Werten. Suche den Wert Sort(20).




Nun aus diesen ⌈n/5⌉-tel=6 Werten Sort((⌈n/5⌉+1)/2) bestimmen. Ergebnis: 45.

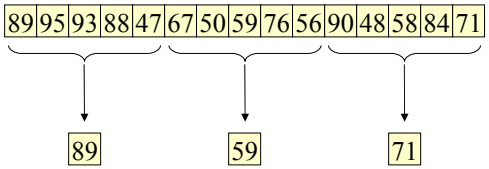
Mit dieser 45 einen Teilsortierschritt in der ursprünglichen Folge durchführen.




Dies sind 13 Elemente. Wegen 20 > 13 muss man also rechts weitersuchen.

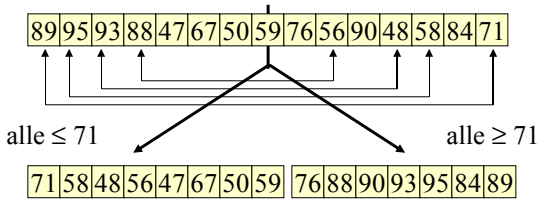
Suche *hier* das Element Sort(20-13) = Sort(7).

Weitermachen mit der Folge aus 15 Werten. Suche hier den Wert Sort(7).




Aus diesen ⌈n/5⌉-tel = 3 Werten Sort((⌈n/5⌉+1)/2)=Sort(2)=71 direkt bestimmen.

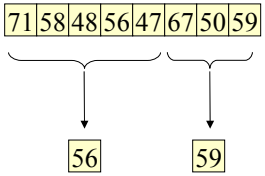
Mit dieser 71 einen Teilsortierschritt in der obigen Folge durchführen.




Dies sind 8 Elemente. Wegen k = 7 ≤ 8 muss man also links weitersuchen nach dem Element Sort(7).

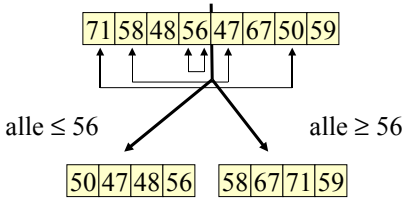


Weitermachen mit der Folge aus 8 Werten. Suche hier den Wert Sort(7).




Aus diesen ⌈n/5⌉-tel = 2 Werten Sort((⌈n/5⌉+1)/2)=Sort(1)=56 direkt bestimmen.

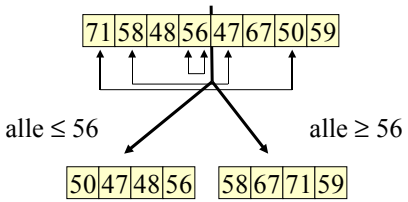
Mit dieser 56 einen Teilsortierschritt in der obigen Folge durchführen.




Links stehen 4 Elemente. Wegen k = 7 > 4 muss man also rechts weitersuchen nach dem Element Sort(7-4) = Sort(3).



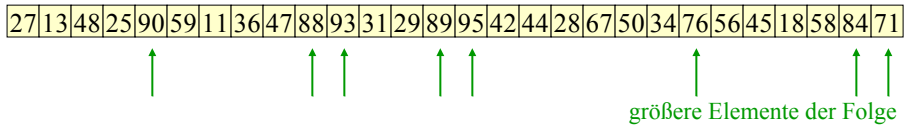
Da rechts aber nur noch 4 Elemente stehen, also weniger als 5, können bzw. müssen wir Sort(3) = 67 direkt bestimmen.
Somit ist 67 das 20. Element der ursprünglichen Folge.




Links stehen 4 Elemente. Wegen k = 7 > 4 muss man also rechts weitersuchen nach dem Element Sort(7-4) = Sort(3).



Nun können wir noch die Charakterisierung von Sort(k) demonstrieren, indem wir die Probe machen, ob das gesuchte Element tatsächlich die Zahl 67 gewesen ist.
Wir laufen also mit Sort(k) = 67 durch das Feld. Es muss gelten: $|\{j \mid A_j \le \text{Sort}(k)\}| \ge k$ und $\{j \mid A_j > \text{Sort}(k)\}| \le n-k$.




Es sind n = 28 und k = 20 sowie
$|\{j \mid A_j \le \text{Sort}(k)\}| = 20 \ge k$ und $|\{j \mid A_j > \text{Sort}(k)\}| = 8 \le n-k = 8$.
Die Charakterisierung von Sort(20) ist erfüllt und 67 ist somit tatsächlich das 20. Element der sortierten Folge.

Nun kommen wir zur Komplexität dieses Verfahrens.

Behauptung: Nimmt man als Pivot-Element p den Median der Fünfer-Mediane und sind alle Elemente der Folge verschieden, so besitzt nach dem Teilsortierschritt jedes der beiden Teilfelder mindestens $3 \cdot n/10 - 1$ und höchstens $7 \cdot n/10 + 1$ Elemente.

Beweis: Wir denken uns die Fünfer-Mediane $m_1, m_2, ..., m_{n/5}$ geordnet (Pivot-Element $p = m_r$ mit $r = n/10$). Für jedes i sei $m_i$ der Median von fünf Elementen $e_i, f_i, m_i, g_i, h_i$ der ursprünglichen Folge, dann ist o.B.d.A. $e_i \le f_i \le m_i \le g_i \le h_i$. Wichtig: Zu $m_i$ gibt es zwei Elemente $e_i$ und $f_i$, die kleiner oder gleich $m_i$ sind. Wegen $m_1 \le m_2 \le ... \le m_r$ sind $e_1, ..., e_r, f_1, ..., f_r, m_1, ..., m_{r-1}$ kleiner oder gleich $p = m_r$. Dies sind genau $3n/10 - 1$ Elemente. Sind alle Elemente verschieden, so liegen diese im linken Teilfeld. Analog für die Elemente, die größer oder gleich $m_r$ sind. Daraus folgt die Behauptung. Erläuternde Skizze:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $e_1$ | $e_2$ | $e_3$ | ... | $e_{r-1}$ | $e_r$ | $e_{r+1}$ | $e_{r+2}$ | ... | $e_{n/5}$ |
| $f_1$ | $f_2$ | $f_3$ | ... | $f_{r-1}$ | $f_r$ | $f_{r+1}$ | $f_{r+2}$ | ... | $f_{n/5}$ |
| $m_1$ | $m_2$ | $m_3$ | ... | $m_{r-1}$ | $m_r$ | $m_{r+1}$ | $m_{r+2}$ | ... | $m_{n/5}$ |
| $g_1$ | $g_2$ | $g_3$ | ... | $g_{r-1}$ | $g_r$ | $g_{r+1}$ | $g_{r+2}$ | ... | $g_{n/5}$ |
| $h_1$ | $h_2$ | $h_3$ | ... | $h_{r-1}$ | $h_r$ | $h_{r+1}$ | $h_{r+2}$ | ... | $h_{n/5}$ |

Farbig unterlegt sind jeweils $3 \cdot n/10 - 1$ Elemente. (Untersuchen Sie den Fall, dass gleiche Elemente in der Folge auftreten.)



**4.2.2.5 Zeitkomplexität**: Nun können wir die Zahl der Vergleiche T(n) dieses Verfahrens berechnen. Wir ersetzen hier (nicht ganz korrekt) $\lceil n/5 \rceil$ durch $n/5$.

*Das Verfahren verläuft folgendermaßen:*

Berechne zu jedem Fünferblock der Folge den Median (in 6 Schritten!) und bilde die Folge dieser Fünfer-Mediane (dies sind n/5 Elemente).

Berechne rekursiv von dieser Folge den Median p.

Führe mit diesem Element p auf der ursprünglichen Folge einen Teilsortierschritt durch.

Stelle fest, in welcher Teilfolge das Element Sort(k) liegen muss und fahre mit dieser Folge rekursiv fort, bis weniger als 6 Elemente übrig sind (dann bestimme das gesuchte Element direkt).

$$T(n) \le 6 \cdot n/5 + T(n/5) + n + T(7 \cdot n/10)$$

Zahl der Vergleiche ≈ Zeit



Für die Anzahl der Vergleiche gilt also höchstens

$$\mathbf{T(n) = 11 \cdot n/5 + T(n/5) + T(7 \cdot n/10)} \quad \text{mit} \quad T(5) = 6.$$

Die Lösung dieser Gleichung ist bekanntlich von der Form

$$T(n) = a \cdot n + b$$

für geeignete Zahlen a und b (weil $1/5 + 7/10 < 1$ ist).

Aufgabe für Sie: Überzeugen Sie sich, dass das Verfahren tatsächlich den Median findet und berechnen Sie die genauen Konstanten a und b. (Einfach einsetzen und Gleichung lösen.)

*Hinweis.* Es gilt: $T(n) < 22 \cdot n$, in der Praxis $T(n) <\approx 16 \cdot n$. Genaue Berechnung: siehe Literatur oder Vorlesung (EA)$^2$ im 6. Semester.

Durch Modifizierung des Verfahrens kann sogar $T(n) < 8 \cdot n$ erreicht werden.

Aufgabe: Schreiben Sie das zugehörige Java-Programm.



4.2.2.7 Die "Paarweise Ungleichheit"

Hinweis auf ein ähnlich aussehendes Problem, das **EDP** = element distinctness problem:

Gegeben ist eine Folge $A_1, A_2, ..., A_n$. Stelle fest, ob die Elemente paarweise verschieden sind.

Nahe liegendes Verfahren: Folge sortieren und danach in einem Durchlauf feststellen, ob ein Element doppelt vorkommt. Aufwand: $O(n \cdot \log(n))$ Vergleiche. Mit dem Algorithmus zur Lösung des Plateau-Problems (Grundvorlesung 7.3.2) kann man dann sogar mit dem gleichen Aufwand für jedes $1 \le k \le n$ berechnen, wie viele verschiedene Elemente es gibt, die genau k-mal in der Folge enthalten sind.

Das Problem EDP ist also nicht schwerer zu lösen als das Sortieren einer Folge. Es *könnte* aber sein, dass es auch zum EDP einen Linearzeit-Algorithmus gibt. Diese Frage ist bisher noch ungelöst.

Zur Illustration: 145 Zahlen. Gibt es zwei gleiche Zahlen?

2301, 4892, 8197, 7823, 6541, 2639, 7891, 6883, 9211, 6738,
3371, 10892, 4394, 13823, 11741, 2663, 4852, 3197, 7623,
7841, 6383, 10512, 6937, 4092, 8144, 7823, 6741, 2639, 7391,
6884, 9291, 6735, 5171, 10892, 4994, 13623, 12742, 2662,
4432, 3857, 5623, 10395, 2394, 1823, 1751, 2263, 4152, 3647,
7635, 7741, 6393, 1022, 6938, 4992, 8744, 4823, 6641, 7739,
5191, 6294, 4971, 7035, 6631, 11542, 4794, 1373, 15542,
2362, 4412, 3707, 5323, 5371, 4191, 4294, 1373, 11940, 2664,
4252, 3737, 7913, 7221, 6373, 11512, 6928, 4492, 2144, 7433,
6641, 12799, 7341, 6284, 9201, 4735, 5441, 10852, 4984,
12223, 11741, 2632, 2432, 3657, 5629, 10355, 4394, 1823,
1751, 7263, 4452, 6647, 8645, 7641, 2383, 1322, 3938, 4022,
8441, 4323, 6941, 7832, 5121, 6354, 4931, 7235, 6031, 9542,
1794, 3273, 4542, 2662, 4812, 2707, 8323, 6484, 9251, 3795,
5071, 6362, 4812, 2747, 5422, 5371, 1592, 4294, 2723, 6242.



## 4.2.3 Erkennung von Teilwörtern (substring-problem)

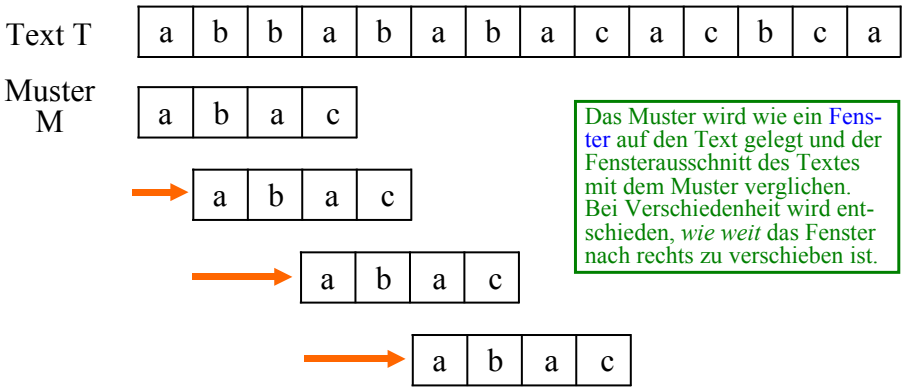
Aufgabe: Gegeben sind:
String T (= der Text)
String M (= das Muster).
Stelle fest, ob M als Teiltext (substring) in T vorkommt.

*Definitionen:* Es sei $\Sigma = \{a_1, a_2, ..., a_r\}$ ein Alphabet mit r Zeichen. $\Sigma^*$ ist die Menge der Wörter über $\Sigma$ mit der Operation "Konkatenation" (= Hintereinanderschreiben) und dem leeren Wort $\varepsilon$. Jedes Wort $w \in \Sigma^*$ lässt sich als $w = w_1 w_2 ... w_k$ schreiben mit $w_i \in \Sigma$; $k \geq 0$ heißt Länge von w.

u heißt Präfix oder Anfangswort von w, wenn es ein Wort v mit $w = uv$ gibt. u heißt Suffix oder Endwort von w, wenn es ein Wort v mit $w = vu$ gibt. u heißt Teilwort ("substring") von w, wenn es Wörter $v_1$ und $v_2$ mit $w = v_1 u v_2$ gibt. Falls zusätzlich $u \neq w$ gilt, so heißt u echtes Präfix, Suffix bzw. Teilwort.



Die Grundidee bei der Teilworterkennung besteht darin, ein Muster M links auf den Text T zu legen, gewisse Zeichen miteinander zu vergleichen und, sobald zwei verschiedene Zeichen vorliegen, das Muster nach rechts zu verschieben und erneut Vergleiche durchzuführen.




Das Muster wird wie ein Fenster auf den Text gelegt und der Fensterausschnitt des Textes mit dem Muster verglichen. Bei Verschiedenheit wird entschieden, *wie weit* das Fenster nach rechts zu verschieben ist.



Naive Lösung, um festzustellen, ob ein Wort $M = M_1...M_m$ Teilwort von $T = T_1...T_n$ ist: vergleiche buchstabenweise.

```
for (int i = 1; i <= n-m+1; i++) {
   j = 1;
   while ((j <= m) && (T_{i+j-1} = M_j)) j++;
   if (j = m+1) return true;
}
return false;
```

In der Praxis wird man die Stelle i als Ergebnis zurückgeben, ab der M ein Teilwort von T ist, bzw. "-1", falls dies nicht zutrifft. Nun muss man noch die Nummerierung der Strings bei 0 statt bei 1 beginnen. Dies führt sofort zu folgendem Java-Programm:

```
class Teilwort {
   static String Text = "text.....";   static String Muster = "....";
   public static int substring (String T, String M) {
      int LT = T.length(); int LM = M.length(); int j;
      if (LM <= LT) {
         for (int i = 0; i <= LT-LM; i++) {
            j = 0;
            while ((j<LM)&&(T.charAt(i+j)==M.charAt(j)))  j++;
            if (j == LM) return i;
      }  }
      return -1;
   }
   public static void main (String [] args) {
      String T = Text; String M = Muster; int Position = substring(T,M);
      System.out.print("Das Muster ist im Text ");
      if (Position < 0) System.out.print("nicht");
      else System.out.print("ab Position " + (Position+1));
      System.out.println(" enthalten.");
   }
}
```

*Ausgabe dieses Programms:* Das Muster ist im Text ab Position 5 enthalten.



In diesem Programm wird nur einmal nach dem Teilwort gesucht. Man kann die Methode substring aber leicht so abwandeln, dass *jedes* Auftreten des Teilwortes Muster notiert und (als String oder als Feld) zurückgegeben wird.
Man kann im Falle i = 0 bzw. i = LT-LM zusätzlich ausgeben, dass das Muster M Präfix bzw. Suffix des Textes T ist.

*Aufwand dieses Verfahrens:* $O(n \cdot m)$ für $n=|T|$ und $m=|M|$. Der schlechteste Fall tritt auch ein, zum Beispiel für die Wörter $T = aaaa...aa = a^n$ und $M = aa...ab = a^{m-1}b$.

Kann man dieses Verfahren beschleunigen? Ja, indem man die Information ausnutzt, dass man bei einer Ungleichheit $T_{i+j} \neq M_j$ ja bereits weiß, dass $T_{i+k} = M_k$ für $k = 0, 1, ..., j-1$ gilt. Man kann dann sofort $T_{i+j+1}$ mit dem $M_{h+1}$ vergleichen, für das gilt:

$h$ ist die größte Zahl mit $0 < h < j$ und $T_{i+j} = M_h$

bzw. $h = -1$, falls es kein solches nichtnegatives $h$ gibt.



Dieses Verfahren ist unter dem Namen "Knuth-Morris-Pratt" in der Literatur geläufig. Man stellt rasch fest, dass die Verschiebungen nur vom Muster abhängen und in Zeit O(m) zu Beginn des Verfahrens berechnet werden können.

In dem Beispiel $T = aaaa...aa = a^n$ und $M = aa...ab = a^{m-1}b$ verschiebt dieses Verfahren das Muster ständig um eine Stelle weiter, sobald man auf das "b" des Musters stößt; denn man hat ja bereits die Gleichheit der letzten (m-1) Buchstaben des Textes mit $a^{m-1}$ erkannt, d. h., dieses Vorgehen führt hier zu einem O(n+m)-Verfahren.

Man kann sich klarmachen, dass dieser Aufwand stets auch allgemein ausreicht. Details und Berechnung der Index-Erhöhungen wurden in der Vorlesung vorgestellt und finden sich in vielen Lehrbüchern.



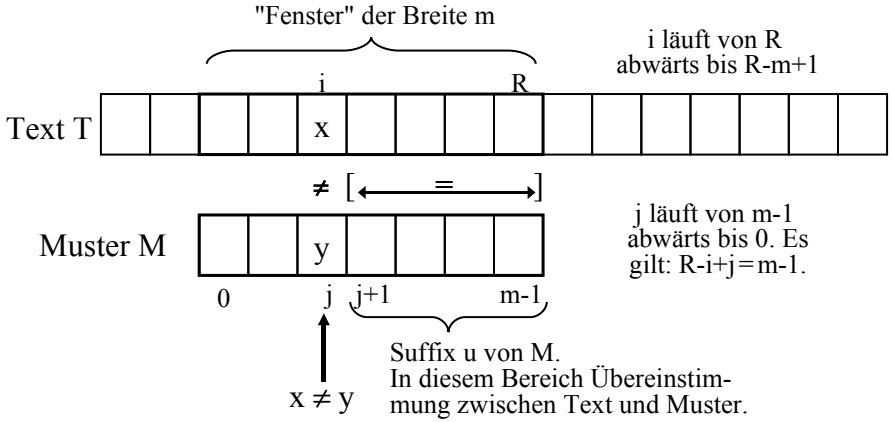
Eine andere Idee lautet: Vergleiche das Muster mit dem Text nicht von vorne nach hinten, sondern *von hinten nach vorn.*

Der Vorteil zeigt sich vor allem bei Buchstaben, die im Text, aber nicht im Muster vorkommen. In dem Beispiel $T = (a^{m-1}c)^s$ und $M = aa...a = a^m$ kann man das Muster bei jedem Vergleich ("ist c gleich a?") sogleich um m Stellen weiterschieben, so dass man insgesamt nur $s = n/m$ Vergleiche benötigt, um festzustellen, dass M kein Teilwort von T ist.

Diese Idee liegt dem Boyer-Moore-Verfahren zugrunde, welches ebenfalls ein O(n+m)-Verfahren ist. Wir geben den naiven Algorithmus an und stellen dann fest, dass mit einer ähnlichen Methode wie beim Knuth-Morris-Pratt-Verfahren (aber von hinten betrachtet) die Linearzeit erreicht wird.

## Das Boyer-Moore-Verfahren, einfache Variante.

Vergleichen von rechts. Betrachte eine typische Situation:




Falls $x = T_i \neq M_j = y$ ist, so verschiebe das Muster eine Position nach rechts, d.h., setze R = R+1; i = R; j = m-1;
(Falls x = y ist, i und j erniedrigen usw., bis j = 0 ist.)



Der Algorithmus ist außer im then-Teil der Alternative unverändert:

```
class TeilwortBM1 {
   static String Text = "text.....";   static String Muster = "....";
   public static int substring (String T, String M) {
      int LT = T.length(); int LM = M.length(); int i, j;
      if (LM <= LT) {
         for (int R = LM-1; R < LT; R ++) {
            i = R; j = LM-1;
            while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
            if (j == -1) return (i+1);
      } }
      return -1;
   }
   public static void main (String [] args) {
      String T = Text; String M = Muster; int Position = substring(T,M);
      System.out.print("Das Muster ist im Text ");
      if (Position < 0) System.out.print("nicht");
      else System.out.print("ab Position " + (Position+1));
      System.out.println(" enthalten.");
   }
}
```

(Gleiche Ausgabe wie beim früheren Programm Teilwort.)



Da dieses Verfahren bei einer Ungleichheit R nur um 1 erhöht, arbeitet es im schlechtesten Fall (worst case) ebenfalls nur in O(n·m). Ein Beispiel hierfür sind die Wörter

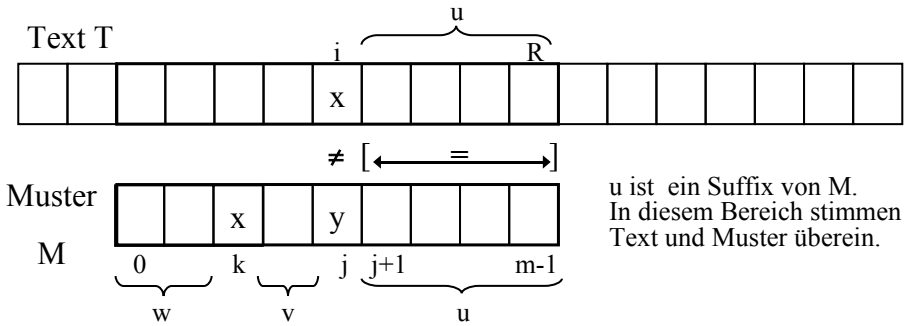
$T = aa...aa = a^n$ und $M = baa...aa = ba^{m-1}$.

Hier stimmen stets m-1 Zeichen überein; erst wenn man auf das 'b' trifft, wird das Muster um eine Stelle nach rechts verschoben. Aber: Es ist klar, dass man in dieser Situation das Muster um m Stellen nach rechts schieben könnte, da b in dem bereits überprüften Teil $a^{m-1}$ nicht vorkommt.

Man erkennt, dass mit der gleichen Idee wie beim KMP-Algorithmus eine deutliche Beschleunigung erzielt werden kann. Wir betrachten dies genauer, wobei erst der zweite Ansatz zum Linearzeitverhalten führen wird.



Ansatz 1: Im Suffix u stimmen T und M überein, und es sei $x \neq y$.




Wenn x links von der Verschiedenheit ("missmatch") nicht mehr im Muster vorkommt, dann verschiebe man das Muster um j+1 Stellen nach rechts. Kommt x dort aber vor, so betrachte das letzte Vorkommen von x, d. h., es sei M = wxvyu und x kommt in v nicht vor. Dann kann man das Muster um j-k Stellen verschieben!

Hat man also $x = T_i \neq M_j = y$ festgestellt, so gehe man ab dieser Position j nach links, bis man x oder das erste Zeichen von M erreicht hat, und erhöhe dann R entsprechend, Ersetze also

```
for (int R = LM-1; R < LT; R ++) {
   i = R; j = LM-1;
   while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
   if (j == -1) return (i+1);
   }
```

durch

```
int R = LM-1;
while ( R < LT)  {
   i = R; j = LM-1;
   while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
   if (j == -1) return (i+1);
   else  { k = j;                          // nun die Position k berechnen
      while ((k >= 0) && (T.charAt(i) != M.charAt(k))) k--;
      R = R + (j - k);                     // dies ist auch für k=-1 richtig.
   }
}
```



So nützlich diese Beschleunigung in der Praxis sein mag, so wenig hilft sie aber beim Groß-O-Verhalten. Ein Beispiel hierfür sind wiederum der Text und das Muster

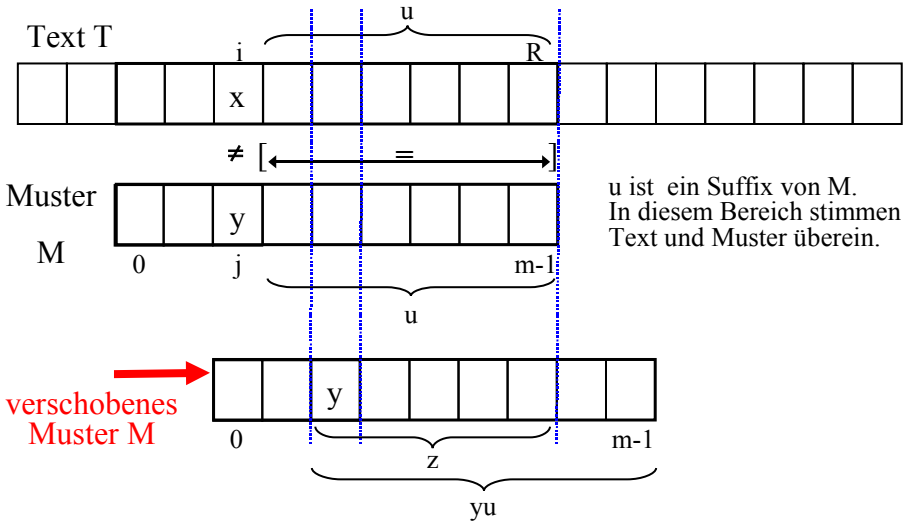
$T = aa...aa = a^n$ und $M = baa...aa = ba^{m-1}$.

Die Verschiedenheit wird erst festgestellt, wenn das 'b' des Musters mit dem Text verglichen wird. Dann wird aber nur um eine Stelle nach rechts verschoben, da b das erste Zeichen des Musters ist.

Wir müssen also die Information ausnützen, dass das Suffix u bereits auf Gleichheit mit dem aktuellen Text überprüft wurde und nun so verschieben, dass der verbleibende Anfang von u wieder ein Suffix des Musters ist. Wir schauen uns dies in der folgenden Skizze an.



Ansatz 2: Im Suffix u stimmen T und M überein, und es sei $x \neq y$.




Man erkennt als Bedingung für das Verschieben um |yu|-|z| Stellen:
z muss das längste Suffix von yu sein, das zugleich Anfang von yu ist.



Die Verschiebungstabelle hängt also ausschließlich vom Muster M ab. Sie lautet:

Wenn $y = M_j$ das j-te Zeichen (die Nummerierung beginne ab 0) im Muster M ist, so beträgt die Verschiebung V[j] (falls hier die erste Verschiedenheit zum Text vorliegt, von rechts aus betrachtet) genau k Positionen mit

$V[j] = k = Min\{t \mid t \geq 1 \text{ und } M_jM_{j+1}...M_{m-t-1} = M_{j+t}M_{j+t+1}...M_{m-1}\}$
bzw. V[j] = m-j, falls ein solches t nicht existiert.

Wenn die Verschiebungen V bekannt sind, so erhöhen wir bei einer Verschiedenheit an der Position j die rechte Grenze R also nicht um 1, sondern um V[j].

Somit erhalten wir aus der alten Methode substring mit dieser Modifizierung die neue Methode substringBM:

```
public static int substringBM (String T, String M) {
   int LT = T.length(); int LM = M.length(); int i, j;
   if (LM <= LT) {
      int [] V = new int [LM];
      < hier fehlt noch die Berechnung des Feldes V >
      int R = LM-1;
      while (R < LT) {
         i = R; j = LM-1;
         while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
         if (j == -1) return (i+1);
         else R = R + V[j];
      }
   }
   return -1;
}
```

Wie berechnet man das Feld V?
Beispiel: Muster M = "abbabab". m = 7.

Für j = m-1 = 6 muss man prüfen, ob das Suffix "b" ein echtes Anfangswort ≠ ε besitzt, welches Suffix von M ist. Dies ist nicht möglich, da "b" die Länge 1 besitzt, d. h. V[6] = 7 - 6 = 1.

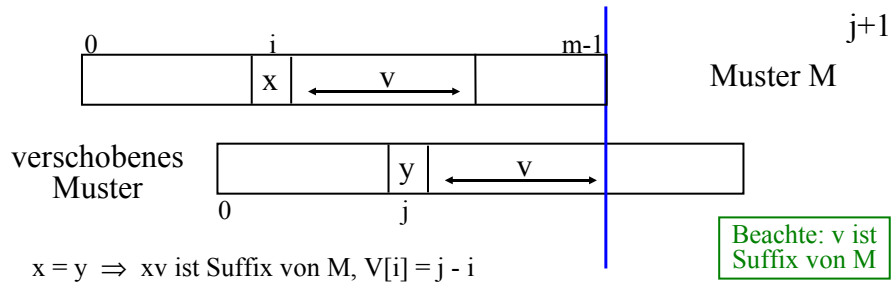
Für j = 5 muss man prüfen, ob das Suffix "ab" ein echtes Anfangswort ≠ ε besitzt, welches Suffix von M ist. Dies ist nicht möglich, d. h. V[5] = 7 - 5 = 2.

Für j = 4 muss man prüfen, ob das Suffix "bab" ein echtes Anfangswort ≠ ε besitzt, welches Suffix von M ist. Dies trifft für "b" mit k = 2 zu, d. h. V[4] = 2. Usw.

So erhält man das Feld V = (5,5,2,2,2,2,1).

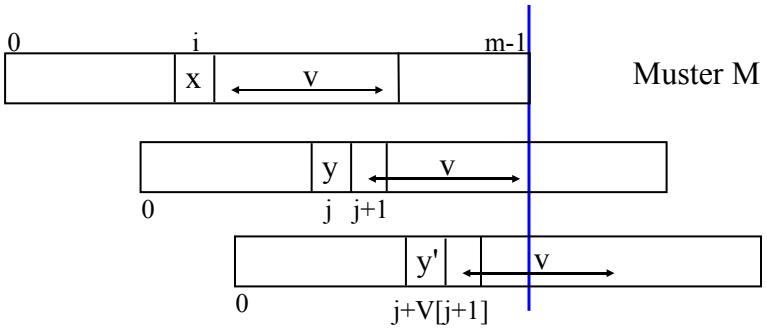
Das Feld V lässt sich rasch mit folgender Überlegung berechnen:






x = y ⇒ xv ist Suffix von M, V[i] = j - i

x ≠ y ⇒ Teste, ob gilt: x = y' = Zeichen links von "verschiebe v" bzgl. j




Dies führt zu folgendem Algorithmus, um das Feld V zu berechnen.

```
int [] V = new int [LM+1]; int i = LM - 1; int j = LM;
V[LM] = 1;                          // beachte, es gilt in der Schleife stets  j > i
while (i >= 0) {
   if ((j = LM) || (T.charAt(i) == M.charAt(j))) {V[i] = j - i; i--; j--; }
   else  j = j+V[j+1];
}
```

Man kann sich überlegen, dass dieses Programmstück nur O(m) Schritte benötigt. (Warum? Es sieht doch nach $O(m^2)$ aus?!)

Es entsteht schließlich folgendes Programm (wobei wir in Java keine gleichen Namen in Unterblöcken deklarieren dürfen, also i und j im Inneren durch i1 und j1 ersetzen):

```
class TeilwortBM2 {
   static String Text = "text.....";   static String Muster = "....";
   public static int substringBM2 (String T, String M) {
      int LT = T.length(); int LM = M.length(); int i, j;
      if (LM <= LT) {
         int [] V = new int [LM+1]; int i1 = LM - 1; int j1 = LM;
         V[LM] = 1;                  // beachte, es gilt in der Schleife stets  j1 > i1
         while (i1 >= 0) {
            if ((j1 == LM) || (T.charAt(i1) == M.charAt(j1)))
               { V[i1] = j1 - i1;  i1--;  j1--; }
            else  j1 = j1+V[j1+1];
         }
         int R = LM-1;
         while (R < LT) {
            i = R; j = LM-1;
            while ((j >= 0) && (T.charAt(i) == M.charAt(j))) {i--; j--;}
            if (j == -1) return (i+1);
            else R = R + V[j];
         }
      }
      return -1;
   }
```



```
   public static void main (String [] args) {
      String T = Text; String M = Muster; int Position = substringBM2(T,M);
      System.out.print("Das Muster ist im Text ");
      if (Position < 0) System.out.print("nicht");
      else System.out.print("ab Position " + (Position+1));
      System.out.println(" enthalten.");
   }
}
```

Konnten Sie dieser Herleitung gut folgen? Dann wird Sie Folgendes überraschen. Gibt man den Text
T = "aabaabbbabaabaaba" und das Muster M = "abaaba" ein, so erhält man die korrekte Ausgabe:

Das Muster ist im Text ab Position 9 enthalten.

Nimmt man hingegen den im Programm angegebenen Text und Muster, so erhält man die falsche Ausgabe:

Das Muster ist im Text nicht enthalten.

Da muss also irgendwo ein logischer Fehler bei der Berechnung des Feldes V sein. Können Sie ihn finden und das Programm korrigieren?



## 4.3 Konzepte und Objektorientierte Sprachen

Die wichtigen Prinzipien der objektorientierten Programmierung finden Sie zu Beginn dieses Kapitels 4. Um sie zu erreichen, haben wir folgende Konzepte kennen gelernt:

- Klassen, Vererbung und die Hierarchie der Klassen (ausgehend von "Object")
- Dynamische Bindung (late binding)
- Polymorphie
- Ausnahmen (exceptions)

Oft fügt man noch "Spezifikationen" (abstrakte Klassen und interfaces), getrennte Übersetzung, Nutzung von Bibliotheken (auch Pakete) und die Verwendung von Mustern hinzu.



Weiterhin gibt man Ziele der ooP an, z. B.:

- Wiederverwendbarkeit
- Adaptionsfähigkeit, Anpassbarkeit
- Portabilität
- Modularität, Kapselung

Sodann gibt man Hinweise zum "Programmieren im Großen". Gerade die objektorientierte Herangehensweise ermöglicht es, kaum überschaubare Probleme in Angriff nehmen und erfolgreich implementieren, warten, einsetzbar machen und meist auch beherrschbar halten zu können.

Mehr hierzu in der Grundlagenvorlesung über Software Engineering und in Veranstaltungen der Praktischen und der Angewandten Informatik.

Objektorientierte Sprachen:

Smalltalk (1980)

C++ (1986), Java (ab 1995), C# (ab 2000)

Eiffel (1988)

Object Pascal, Delphi, Oberon, Oberon-2 (1986-1991)

Hinzuzählen sind weiterhin:
Simula 67, CLOS, Modula-3, Ada 95, Ada 2005, Beta, Visual Objects und etwa 30 weitere weniger verbreitete Sprachen.



# 5. Zusammenfassung der Vorlesung



Hier nur Stichwörter.



## 5.1 Was wurde vermittelt?

Zentrales Thema: "Programmierparadigmen":

Maschinennahes Vorgehen.

Funktionales Vorgehen.

Objektorientierung.

Nebenläufigkeit.

Im Vordergrund stehen die Modelle und abstrakten Begriffe, auch wenn konkret Scheme und Java mitgelehrt wurden und viel Zeit kosteten. Die Konzepte wurden präsentiert, aber die tatsächlichen Möglichkeiten können nur durch zusätzliche Vertiefungen vermittelt werden. Sinnvoll wäre daher ein "vielsprachiges" Software-Praktikum, welches sich in der vorlesungsfreien Zeit anschließen sollte, sowie eine Theorie-Veranstaltung zu Algorithmen und Kalkülen.



## 5.2 Was fehlte noch?

Weitere Paradigmen:
- Prädikatives Programmieren
- Gleichzeitigkeits-Programmierung (systolische Arrays, zelluläre Ansätze, Schaltkreise)
- Spezifizieren, Modellieren (z. B. mit UML)
- Heuristiken (z. B. evolutionäre Algorithmen)
- Grundkonzepte der Betriebssysteme und der Datenbanken

Standardbeispiele, zum Beispiel
- Anagramme
- Komplette Analyse des Teilworterkennungs-Problems
- Flüsse in Graphen
- Zuordnungsprobleme, Scheduling
- Heuristiken für TSP (Problem des Handlungsreisenden)
- Basisalgorithmen für data mining

## 5.3 Was baut später auf dieser Vorlesung auf?

Die Vorlesung "Einführung in die Informatik III" soll die "Grundlagen"-Vorlesungen ab dem 5. Semester vorbereiten. Die Hörer(innen) haben einen Einblick in die Algorithmik und die gängigen Sprachen erhalten, in den Aufbau von Informatiksystemen und in die prinzipiellen Herangehensweisen.

Wichtig ist die Erfahrung, dass man Probleme mit recht unterschiedlichen Ansätzen in Angriff nehmen und dann mit unterschiedlicher Qualität lösen kann.

Weiterhin wurde auch Wert auf ein das Handwerkliche gelegt: In kommenden Vorlesungen sind stets (kleine) Programme zu schreiben - und von Informatiker(innen) darf man durchaus eine gewisse "beherrschte Vielsprachigkeit" erwarten.



## Ende der Vorlesung „Einführung in die Informatik III“

Wintersemester 2006/2007, Universität Stuttgart

Fakultät 5, Fachbereich Informatik

Institut für Formale Methoden der Informatik

*Kontakt bzgl. der Vorlesung und der Übungen:*

Volker Claus, Botond Draskoczy, Dietmar Lippold

Universitätsstraße 38, 70569 Stuttgart

Tel.: 0711 - 7816 - 300, - 225, - 435 oder - 328; Fax: - 310