Notizen zur Vorlesung
Digitale Informationsverarbeitung
Wintersemester 2001/02

H.-G. Gräbe, Institut für Informatik,
`http://www.informatik.uni-leipzig.de/~graebe`

6. Januar 2002

Inhaltsverzeichnis

1 Einführung

1.1 Was ist Informatik ?

In verschiedenen Nachschlagewerken werden verschiedene Definitionen gegeben. Diese bewegen sich im Spannungsfeld zwischen den Begriffen Information und Computer.

Schneider, Lexikon der Informatik / Duden Informatik :

> Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen – insbesondere der automatischen Verarbeitung mit Hilfe von Digitalrechnern.

Remboldt (etwas vorsichtiger):

> Die Informatik hat es sich u.a. zur Aufgabe gestellt, mittels elektronischer Rechner nach vorgegebenen Regeln Daten zu erfassen, zu verarbeiten und die Ergebnisse zum

> weiteren Gebrauch geordnet darzustellen. Die Verarbeitung wird mit dazu erstellten Algorithmen im zeitlichen Ablauf durchgeführt.

Rechenberg: Was ist Informatik? Als Antwort ein ganzes Buch.

Aus dem Vorwort:

Informatik = Kunstwort aus „Information“ und in Anlehnung an „Mathematik“.

Im anglo-amerikanischen Sprachraum „Computer Science“ und inzwischen auch häufiger „Computing Science“

Zu klären also Verhältnis der vielfältigen hier verwendeten Begriffe, besonders Information, Computer, Algorithmus. Dabei werden an dieser Stelle mehr Fragen offen bleiben als Antworten gegeben.

1.2 Daten und Information

zuück zu Remboldt. Worin besteht der Unterschied zwischen Daten und Information. Gibt es überhaupt einen Unterschied?

Daten = die gegenständliche Form, in der Information daherkommt.

Wo werden Daten erzeugt? Allgegenwart von Datenerzeugungsprozessen.

Materie als Dreieinigkeit ihrer stofflichen, energetischen und informatorischen Seiten.

Riesiger dauernder Datenfluss erfordert Datenauswahl vor der Verarbeitung. Dabei kommt die Bedeutung der einzelnen Daten ins Spiel.

Information = mit Bedeutung aufgeladene und selektierte Daten

Damit Bindeglied zwischen objektiver (Daten) und subjektiver (Bedeutung) Welt.

Biolog. Informationsverarbeitung: 10^{10} bits/s. gehen an den Rezeptoren ein, nur etwa 25 bit/s. werden vom Gehirn verarbeitet.

1.3 Zwecke und Information als dafür bedeutsame Daten

Zweck der Information:

biologisch: um auf Umweltveränderungen zu **reagieren**.

Belebte Natur unterscheidet sich von der unbelebten durch ihre Fähigkeit zur Informationsverarbeitung.

Wie wird Information in biolog. Systemen aufgenommen? (Reize + Daten $\neq$ Information)

Selektion und Bewertung: Neues wird in ein bereits vorhandenes Etwas eingebaut, wodurch sich dieses Etwas zugleich weiterentwickelt.

statt (a) I.-Verarbeitung und I.-Erzeugung $\Rightarrow$ (b) I.-Austausch und I.-Entstehung

(a): technizistisch-deterministisches Verständnis

(b): kommunikativ-selbstorganisierendes Verständnis

sozial: (auch bei Tieren) um koordiniert **handeln** (agieren) zu können.

kommunikativer und aktionistischer Aspekt von Informatik

Mensch: kann im Gegensatz zu Tieren Aktionen nicht nur im Heute koordinieren, sondern vorausdenken und planen. Mit Sprache und Abstraktion kommt zum Informationsbegriff eine deskriptive

Komponente, die Fähigkeit zur (analytischen) Beschreibung hinzu.

Informationbekommt damit auch eine eigene interindividuelle Komponente (es gibt also im Bereich der menschlichen Gemeinschaft so etwas wie einen **Informationsraum**)

Zusammenfassung:

1. Information kann nicht allein auf Daten, also die syntaktische Struktur, reduziert werden.
2. Information ist ein Verhältnis von drei wesentlichen Aspekten, eine Trias von (objektiver) Form (Syntax), (subjektivem) Inhalt (Semantik) und (zwischen beiden Seiten vermittelnder) Wirkung (Pragmatik)

Aspekt	individuell	interindividuell	
dinglich	Daten	Realität	Syntax
deskriptiv	Bedeutung	Wissen	Semantik
kommunikativ	Wirkung	Handeln	Pragmatik

3. Es ist nicht möglich, die Bedeutung (Semantik) von Information syntaktisch vollständig zu speichern. Jeder Informationsaustausch bedarf deshalb eines **Kontexts**, der einen gemeinsamen semantischen Rahmen setzt.

informare (lat.) = „Gestalt geben“ bzw. „jemanden durch Unterweisung bilden“

spiegelt die beiden Seiten (deskriptiv und kommunikativ) wider. Betrachtet das Ganze aber immer noch sehr individuell.

Menschheit: Informationsraum ist nur gesamtgesellschaftlich nutzbare Ressource und seinerseits weiter strukturiert. Er ist die Basis der menschlichen **Vernunft** in ihrer Einheit aus reflexiver Vernunft (Nachdenken über den Gang der Dinge; Kultur, geisteswissenschaften) und instrumenteller Vernunft (Einflussnahme auf den Gang der Dinge; Naturwissenschaften, Technik).

1.4 Noch einmal: Was ist Informatik ?

1.4.1 Informatik und Computer

Informationsverarbeitung durch den Computer? FALSCH. Die subjektive, semantische Seite, erschließt sich dem Computer nicht. Er kann nur die syntaktische Seite der Information, die verschiedenen Bitmuster erfassen.

Richtiger: Informationsverarbeitung (systematische) MIT HILFE des Computers!

1.4.2 Kernbereiche der Informatik

Untersuchungsgegenstand der Informatik umfasst drei wesentliche Komponenten:

- Struktur, Wirkungsweise, Konstruktionsprinzipien von Informations**systemen**
- Struktur, Eigenschaften, Beschreibungsmöglichkeiten von Informations**prozessen**
- **Formalisierung und Modellierung** konkreter Anwendungen

Damit ergeben sich als Kernbereiche der Informatik:

Computer als technisches Gerät	Hardware	**Technische Informatik**	Hardwarekomponenten, Prozessoren, Schaltkreise, Mikroprogrammierung, Rechnerarchitektur, Rechnernetze
Computer als Universalmaschine zur Implementierung von Algorithmen	Software	**Praktische Informatik**	Algorithmen und Datenstrukturen, Programmiermethoden, Programmiersprachen, Übersetzer, Betriebssysteme, Softwaretechnik, verteilte Systeme
Organisation von IuK-Prozessen mit Hilfe von Computersystemen	Orgware	**Wirtschaftsinformatik**	E-Business, Managementsysteme, Software-Engineering, Büroautomatisierung
Entwicklung formaler Methoden, um übergreifende Gesetzmäßigkeiten, die Algorithmen, Datenstrukturen und ihrer Verarbeitung zugrunde liegen, zu erkennen und zu beschreiben.		**Theoretische Informatik**	Automatentheorie, Formale Sprachen, Berechenbarkeitstheorie, Komplexitätstheorie, Programmverifikation, Logik

Aufsetzend auf dem Kompetenzen der Kerninformatik ergeben sich verschiedene Verbindungen in alle Bereiche des gesellschaftlichen Lebens. Diese sind Gegenstand der **Angewandten Informatik**:

- Mechanisierung sowie Automatisierung von Kalkülen aus Wissenschaft, Technik, Verwaltung und Ökonomie
- Erforschung übergreifender Gesetzmäßigkeiten und Systematisierung von Methoden der Organisation von Datenverarbeitung und Informationsprozessen in allen gesellschaftlichen Bereichen
- Einbettung in benachbarte Wissenschaften (Mathematik, Physik, E-Technik)

In Abgrenzung zu den benachbarten Wissenschaften: Informatik erforscht dabei

- grundsätzliche Verfahrensweisen der Informationsverarbeitung
- allgemeine Methoden der Anwendung in verschiedenen Bereichen

unter Abstraktion von

- speziellen technischen Realisierungen
- Besonderheiten spezieller Anwendungen

und stößt dadurch zu den allgemeinen Gesetzmäßigkeiten vor, die der Informationsverarbeitung zugrunde liegen.

1.4.3 Informatik als Querschnittswissenschaft

Informatik im engeren Sinne (als Wissenschaft von der automatischen Verarbeitung von Daten mit dem Computer) ist eher zu den Ingenieurwissenschaften zu rechnen und wird als solche stärker an Technischen Universitäten und Fachhochschulen gelehrt.

Informatik im hier entwickelten weiteren Sinne ist eine Querschnittswissenschaft wie Mathematik oder Philosophie und durchdringt viele Bereiche der Einzelwissenschaften, indem sie diesen methodisches und instrumentelles Handwerkszeug bereitstellt.

Mit der allgegenwärtigen Verfügbarkeit leistungsfähiger Rechentechnik wird die „Verwissenschaftlichung“ gesellschaftlicher Zusammenhänge auf eine qualitativ neue Stufe gehoben. Viele, auch umfangreichere Kalküle können nun mechanisiert oder sogar automatisiert werden und stehen damit für einen breiteren Einsatz zur Verfügung, womit sich zugleich die Reichweite wissenschaftlicher Gedankenführung für einen weiten Kreis von Anwendungen deutlich erhöht. Neben Pflege, Weiterentwicklung und Vermittlung entsprechender Denk-Kalküle, dem traditionellen Gegenstand *mathematischer* Bildung, tritt damit eine weitere Querschnittswissenschaft ins Rampenlicht der Öffentlichkeit. Diese heißt Informatik und

> hat die Erstellung und Pflege, die konkrete und konzeptionelle Weiterentwicklung dieser technikbasierten Hilfsmittel geistiger Arbeit sowie methodische, praktische, ergonomische, physiologische, psychologische und soziale Aspekte des Einsatzes und der gesellschaftlichen Einbettung dieser Instrumente, kurz, eine sich neu herausbildende „technologische Seite des Denkens“ (Buchberger), zum Gegenstand.

Auch wenn das Selbstverständnis der Informatik gewöhnlich enger ausfällt (siehe Definition aus dem Duden Informatik), so ist es doch gerade eine solche Einordnung, die die Informatik als Querschnittswissenschaft charakterisiert und aus der sich auch die Rechtfertigung eines eigenständigen Schulfaches „Informatik“ ableiten lässt.

1.5 Zum Inhalt der Vorlesung DigInf

Im Mittelpunkt stehen Fragen der Algorithmisierung von Prozessen, um diese der automatischen Verarbeitung mit einem (sequentiell arbeitenden) Computer zu erschließen.

- Konzept des (sequetiellen) Rechners
- Entwurf und Bewertung von Algorithmen
- Datenkodierung und Aufbau elementarer Datentypen
- Datenstrukturierungskonzepte
- Implementierung von Algorithmen in einer höheren Programmiersprache (am Beispiel der Sprache C)

Kurzer Überblick über die Literatur.

Literatur

Primärliteratur :

1. M. Dal Cin, J. Lutz, T. Risse: Programmierung in Modula-2. B.G. Teubner, Stuttgart 1989 (4. Auflage). (Standort HB)
   - Teil I: Grundlagen (Elementare Datentypen, Anweisungen und Kontrollstrukturen, Prozeduren und Funktionen, strukturierte Datentypen, dynamische Datentypen)
2. H. Ernst: Grundlagen und Konzepte der Informatik. Vieweg, Braunschweig, 2000 (2. Auflage). (Standort Dat/LS)

   1. Was ist Informatik?
   1.4. Zahlsysteme und binäre Arithmetik
   2. Nachricht, Information, Codierung
   6. Höhere Programmiersprachen
      6.1. Zur Struktur höherer Programmiersprachen
      6.2. Methoden der Syntaxbeschreibung
      6.3. Eine moderne Programmiersprache: C
3. B.W. Kernighan, D.M. Ritchie: Programmieren in C. Hanser, München, und Prentice-Hall, London, 1990 (2. Auflage). (Standort Dat/LS)
4. W. Küchlin, A. Weber: Einführung in die Informatik (mit Java). Springer, Berlin, 2000. (Standort Dat/LS)

   4. Algorithmen und algorithmische Sprachkonzepte
   5. Grundlagen der Rechnerarchitektur und elementare Datentypen
   6. Grundkonzepte von Programmiersprachen
5. P. Rechenberg, G. Pomberger: Informatik-Handbuch. Hanser, München, 1999 (2. Auflage). (Standort Dat/LS)

   B 1. Zeichen- und Zahldarstellungen
   B 2.2. Informationstheorie
   D 2 Programmiersprachen
6. P. Rechenberg : Was ist Informatik ? – Eine allgemeinverständliche Einführung. Hanser, München, 2000 (3. Auflage). (Standort Dat/LS)
7. B. Stroustrup: Die C++-Programmiersprache. Addison-Wesley, 1998 (3. Auflage). (Standort Dat/LS)

Nachschlagewerke :

8. Duden Informatik. Dudenverlag Mannheim 1993 (2. Auflage).
9. H.-J. Schneider (ed.) : Lexikon der Informatik und Datenverarbeitung. Oldenbourg Verlag, München 1986 (2. Auflage).

Ergänzende Literatur :

10. H.-J. Appelrath, J. Ludewig : Skriptum Informatik. B.G. Teubner, Stuttgart 1991.
    - imperative Programmierung am Beispiel Modula-2
11. G. Goos: Vorlesungen über Informatik I. Springer, Berlin, 1997 (2. Auflage).

    1. Grundbegriffe
    B 1.-3. Codierung und Zahldarstellung

12. G. Hotz: Einführung in die Informatik. B.G. Teubner, Stuttgart 1990.
    - Programmiersprachen (Grundlagen, Prozeduren, Datenstrukturen, Unvollständigkeitssatz)
13. D.E. Knuth: The art of computer programming. Addison Wesley, 1991 (3 Bände)
14. T. Ottmann, P. Widmeyer: Algorithmen und Datenstrukturen. BI Wissenschaftsverlag, Mannheim, 1993 (2. Auflage)
15. U. Rembold u.a. : Einführung in die Informatik für Naturwissenschaftler und Ingenieure. Hanser, München 1991.
16. U. Schöning: Algorithmen - kurz gefasst, Reihe Hochschultaschenbuch, Spektrum Verlag Heidelberg, 1997.
17. R. Sedgewick : Algorithmen. Addison Wesley, Bonn 1992.
18. N. Wirth : Systematisches Programmieren. B.G. Teubner, Stuttgart 1983 (6. Auflage).
    - imperatives Programmieren am Beispiel Pascal
19. N. Wirth : Algorithmen und Datenstrukturen. B.G. Teubner, Stuttgart 1986 (4. Auflage).
    - Kap. 1: Fundamentale Datenstrukturen

2 Der Algorithmenbegriff

2.1 Algorithmen und Datenstrukturen – zentrale Begriffe der Informatik

Zum Algorithmenbegriff

- naiv :
  Wie bringe ich die (endlich vielen) zur Verfügung stehenden (elementaren, atomaren) Instrumente zu einem solchen Zusammenspiel, dass damit eine vorgegebene Aufgabe gelöst wird.
- Sedgewick (umgebungsbezogen) :
  Verfahren zur Lösung von Problemen, die für eine Realisierung auf einem Computer *geeignet* sind.
- Rechenberg (aktionistisch) :
  Ein endliches, schrittweises Verfahren zur Berechnung von gesuchten aus gegebenen Größen, in dem jeder Schritt aus einer (endlichen) Anzahl ausführbarer, eindeutiger Operationen und einer Angabe über den nächsten Schritt besteht.
- Schneider (deskriptiv) :
  Allgemeines (eindeutiges) Verfahren zur Lösung einer Klasse gleichartiger Probleme, gegeben durch einen aus elementaren Anweisungen an einen (menschlichen oder maschinellen) Rechner bestehenden (endlichen) Text, der für jeden möglichen Zustand, der bei der Bearbeitung des Problems entstehen kann, eine genaue Anweisung bereithält.

Diskussion der Begriffe, die in diesen vier Definitionen verwendet werden.

Ein Algorithmus ist mehr als die Lösung eines individuellen Problems, denn er verlangt eine ziemlich detaillierte Strukturierung der Lösung.

Algorithmische Lösungen von Problemen sind deshalb nur dann sinnvoll, wenn der analytische Mehraufwand durch den sachlichen Minderaufwand kompensiert wird, z.B. wenn

- dasselbe Problem mit unterschiedlichen Parametern mehrfach zu lösen ist
- das Problem einmalig, aber mit einer großen Eingabemenge zu bearbeiten ist
- das Problem zur Lösung auf andere delegiert werden soll

Letzteres zeigt, dass algorithmisches Problemlösen eine kommunikative Situation ist (Mensch - Mensch). Aber auch die ersten beiden Fälle sind kommunikative Situationen (Mensch - Maschine).

Haben eine **semantische Seite** (Bedeutung), die mit der Algorithmisierung in eine **syntaktische Form** (Beschreibung) überführt werden muss, die auch die Gegenseite versteht (**gemeinsamer semantischer Kontext**), und zwar so, dass der (von mir) anvisierte **pragmatische Zweck** erreicht wird.

Unterscheide damit folgende Aspekte / Fragestellungen bei Algorithmen:

- Beschreibung des Algorithmus selbst (Syntax, ein endlicher Text).
- Mittel, die zum Verständnis / zur Abarbeitung des Algorithmus erforderlich sind (vorausgesetzter gemeinsamer semantischer Kontext)

  Algorithmenbeschreibungen sind damit eigentlich nur als Wenn - Dann - Aussagen möglich.

- Korrektheit des Algorithmus (macht er wirklich, was er soll, Pragmatik).

  Bedarf streng genommen eines exakten mathematischen Beweises. Oft gibt man sich mit einer ausreichenden Anzahl von Tests zufrieden.

Wir wollen immer davon ausgehen, dass Problemlösen im Zusammenhang mit Algorithmen bedeutet, **eine ganze Problemklasse** $P(n)$ mit unendlich vielen Probleminstanzen zu lösen.

Ist etwa $MatMult(n)$ das Problem, zwei $(n \times n)$-Matrizen zu multiplizieren, so ist $MatMult(2)$ die Probleminstanz, zwei (2×2)-Matrizen zu multiplizieren.

Solche Problemklassen $P : E \to A$ heißen in der Mathematik auch **Funktionen**, die Probleminstanzen werden **Funktionswerte** genannt.

2.2 Ein Beispiel: Bestimmen des Maximums einer Menge von Zahlen.

> **Aufgabe**: *Gegeben ist eine Liste l reeller Zahlen. Man bestimme die größte dieser Zahlen.*

Verbales schrittweises Vorgehen.

1. Nimm erstes Listenelement als temporären Wert m des Maximums.
2. Prüfe für alle weiteren Listenelemente x:
   Ist $x > m$? Wenn ja, so setze temporären Wert m des Maximums auf x.
3. Nachdem alle Listenelemente geprüft sind, gib m zurück.

MuPAD-Lösung 1:

```
getmax:=proc(l) local u;
  begin u:=l[1];
  for x in l do
    if x>u then u:=x end_if
  end_for;
  u;
end_proc;

l:=[sin(i*1.0) $ i=1..5 ];

  [0.8414709848, 0.9092974268, 0.141120008, -0.7568024953, -0.9589242747]

getmax(l);

                              0.9092974268
```

Algorithmenbeschreibung wirft sofort die nach der zu verwendenden Sprache auf. Kontextabhängigkeit. (Hier: Muss Listen kennen, Listentraversion mit *in*, spezielle Syntax für verschiedene Arten von Steuerkonstrukten)

Zweite Lösung:

```
getmax:=proc(l) local u,i;
  begin u:=l[1];
  for i from 2 to nops(l) do
```

```
    if l[i]>u then u:=l[i] end_if
  end_for;
  u;
end_proc;

getmax(l);
```

$$0.9092974268$$

Was wird an Sprachmitteln verwendet? Kann man das auf C übertragen? Man braucht Listen !

Effizienz? Unterschied, ob l als Liste oder als Array implementiert ist. $O(n)$ versus $O(n^2)$.

2.3 Einschub: Komplexität von Algorithmen. Die O-Notation.

Generell schwierig, all das exakt auszudrücken. Auch kommt es nicht auf konstante Faktoren an.

Definition **O-Notation**:

Seien $f, g : \mathbf{N} \to \mathbf{N}$ zwei Funktionen. Wir schreiben

$$f \in O(g) \;:\Leftrightarrow\; \exists C > 0, N \,\forall n > N \; f(n) < C \cdot g(n)$$

Äquivalent: $\frac{f(n)}{g(n)}$ ist beschränkt.

Kriterium, das oft angewendet wird: Wenn ein (endlicher) Grenzwert $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existiert.

Bedeutung: Die Wachstumsordnung von $f(n)$ ist nicht größer als die von $g(n)$.

Gilt $f \in O(g)$ und $g \in O(f)$, so haben beide Funktionen gleiche Wachstumsordnung.

Beispiele: Oben Zahl der elementaren Sprünge $\sum_{i=2}^{n} i - 1 = \frac{n(n-1)}{2} \in O(n^2)$

$$\frac{n^2 + 3n + 5}{n + 1} = \frac{n + 3 + \frac{5}{n}}{1 + \frac{1}{n}} \in O(n)$$

Interessante Vergleichsfunktionen sind insbesondere die Klassen $\{n^a \log(n)^b\}$:

$a = 1, b = 0$	lineares Wachstum
$a = 2, b = 0$	quadratisches Wachstum
$a = 0, b = 1$	logarithmisches Wachstum
$a = 0$	polylogarithmisches Wachstum

2.4 Beschreibungen von Algorithmen erstellen

1. Exakte Problemformulierung (auch wieder durch einen endlichen Text)

   Beispiel: Bestimmen Sie die Quersumme $QS(n)$ einer natürlichen Zahl n.

2. Problemanalyse.

   Satz 1 (Division mit Rest) *Zu zwei natürlichen Zahlen $m, n > 0$ existieren stets zwei eindeutig bestimmte natürliche Zahlen q, r mit $m = q \cdot n + r$ und $0 \le r < n$. q wird als (ganzzahliger) Quotient m div n, r als Rest m mod n bezeichnet.*

C stellt diese beiden Funktionen als elementare Funktionen / und % zur Verfügung.

3. Lösung durch schrittweise Verfeinerung.
   Möglichkeiten der Angabe des Algorithmus: verbal, Ablaufdiagramm, Pseudocode (`bsp2a.c`)
   Implementierung ist eine besonders exakte Form der Beschreibung (`bsp2b.c`)

2.5 Der Euklidsche Algorithmus

Beszimmung des gcd durch fortgesetzte Division mit Rest.

Beispiel:

$$\begin{aligned} 37 &= 5 \cdot 7 + 2 \\ 7 &= 3 \cdot 2 + 1 \\ 2 &= 2 \cdot 1 + 0 \end{aligned}$$

Allgemeine Notation.

$$\begin{aligned} a &= q_0 \cdot b + r_1 \\ r_0 = b &= q_1 \cdot r_1 + r_2 \\ r_1 &= q_2 \cdot r_2 + r_3 \\ \dots & \\ r_{m-3} &= q_{m-2} \cdot r_{m-2} + r_{m-1} \\ r_{m-2} &= q_{m-1} \cdot r_{m-1} + 0 \end{aligned}$$

Demo mit MuPAD:

```
EuklidDemo:=proc(a,b) local r,q;
  begin
  while (b>0) do
    r:=a mod b; q:=a div b;
    print(Unquoted,"".a." = ".q."*".b." + ".r);
    a:=b; b:=r;
  end_while;
  a;
end_proc;

EuklidDemo(37,7);

EuklidDemo(311*217737,311*7961);
```

Bestimmung des größten gemeinsamen Teilers mit Hilfe des Euklidschen Algorithmus.

```
intgcd:=proc(a,b) local r;
  begin
  while (b>0) do
    r:=a mod b; a:=b; b:=r;
  end_while;
  a;
end_proc;

intgcd(3*10^12+1,10^8-21);
```

```
EuklidDemo(3*10^12+1,10^8-21);
```

Beweis der Korrektheit durch Schleifeninvariante: $m = q \cdot n + r \;\Rightarrow\; gcd(m,n) = gcd(n,r)$.

Effizienz: Aufwand hängt von der Zahl m der Divisionen mit Rest ab.

Def. Fibonaccifolge: $F_0 = F_1 = 1,\; F_{m+2} = F_{m+1} + F_m$ für $m > 0$.

MuPAD-Code (alles in `bsp2c.mu`):

```
f:=proc(i) option remember;
  begin
  if i<2 then return(1)
  else f(i-1)+f(i-2) end_if
end_proc;
```

$F_{1\ldots20} = [1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946]$

Wachstum: $2\,F_m < F_{m+2} < 2\,F_{m+1} \;\Rightarrow\; \sqrt{2}^{m-1} < F_m \le 2^{m-1}$ (Beweis mit Induktion)

Binetsche Formel: Es gilt

$$F_i = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{i+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i+1}\right) \sim O(\alpha^i)$$

mit $\alpha = \frac{1+\sqrt{5}}{2} \sim 1.618$.

Satz 2 (Satz von Lamè, 1845) *Seien $a > b > 0$ zwei Zahlen, für die der Euklidsche Algorithmus genau m Schritte benötigt. Dann gilt $a \ge F_{m+1}$.*

Korollar 1 *Ist $l(a)$ die Zahl der Ziffern (im Dezimalsystem) der größeren der beiden Zahlen, so benötigt der Euklidsche Algorithmus zur Berechnung von* `gcd(a,b)` *höchstens $m \le l(a) \cdot \frac{\log(10)}{\log(\alpha)}$ Divisionen mit Rest.*

Beweis: $a \ge F_{m+1} \approx \frac{\alpha^2}{\sqrt{5}}\,\alpha^m \approx 1.17\,\alpha^m > \alpha^m \;\Rightarrow\; l(a) = \log_{10}(a) > m\log_{10}(\alpha) = \frac{\log(a)}{\log(10)}$.

2.6 Schnelles Potenzieren

> **Aufgabe**: *Gegeben sind $x \in \mathbf{R}, n \in \mathbf{N}_+$. Man berechne $x^n \in \mathbf{R}$ mit möglichst geringem Aufwand.*

1. Die klassische Iteration:

```
float power1(float x, int n) {
  float p; int i;
  p=1;
  for(i=0;i<n;i++) p=p*x;
  return p;
}
```

Beispiel per Hand: 2^{100}

2. Schnelles Potenzieren (binäres Potenzieren):

```
float power2(float x, int n) {
  if (n==0) return 1;
  else if (n==1) return x;
  else if (n%2==0) return power2(x*x,n/2);
  else return x*power2(x*x,n/2);
}
```

Vergleich in der Datei `bsp2d.c`

Korrektheit? Termination?

Komplexität:

Die Anzahl der Multiplikationen $C_2(n)$ erfüllt hier die Rekursionsbeziehung $C_2(n) \leq C_2(\frac{n}{2}) + 2$, was $C_2(n) = O(\log(n))$ liefert. Zur Berechnung von 2^{100} benötigen wir nach dem ersten Verfahren 100 Multiplikationen, nach dem zweiten Verfahren nur 8 !

2.7 Zusammenfassung Algorithmeneigenschaften

Grundlage: Definition von Rechenberg

Ein endliches, schrittweises Verfahren zur Berechnung von gesuchten aus gegebenen Größen, in dem jeder Schritt aus einer (endlichen) Anzahl ausführbarer, eindeutiger Operationen und einer Angabe über den nächsten Schritt besteht.

Mit folgenden Ergänzungen

1. Gegeben - Gesucht: Der Zusammenhang zwischen Eingabe und Ausgabe (das Problem) ist durch einen endlichen Text spezifiziert, der eine ganze Problemklasse $P : E \to A$ erfasst.

   Im Idealfall ist eine solche **Spezifikation** eine vollständige formale, von konkreten Gegebenheiten unabhängige Beschreibung des Verhaltens des zu erstellenden Systems.

2. Die Lösung (Algorithmenbeschreibung) ist ebenfalls durch einen endlichen Text spezifiziert.

   Dieser gibt eindeutige Anweisungen, wie für *jede* der (auch unendlich vielen) möglichen Eingaben $e \in E$ die zugehörige Ausgabe $a = P(e) \in A$ erzeugt werden kann.

3. Dieser Text enthält für jede bei der Abarbeitung mögliche Situation eine eindeutige Anweisung zur Fortsetzung des Algorithmus (z.B. auch „Wähle eine Zufallszahl“, deren Ergebnis nicht bei jeder Abarbeitung des Algorithmus gleich ausfallen muss).

   Also: Determiniert, aber nicht unbedingt deterministisch.

   Unterschied zu Kochrezepten

4. Die Detailliertheit der geforderten Beschreibung hängt vom jeweiligen **Kontext** ab, d.h. von den sprachlichen Mitteln, die für ein Verständnis vorausgesetzt werden (können).

   Für Algorithmen, die auf einem Computer implementiert werden sollen, spielen dabei vor allem die **Programmiersprache** und die **Programmierumgebung** eine zentrale Rolle.

5. Der Algorithmus **berechnet** bei Eingabe $e \in E$ die Ausgabe $a \in A$, wenn er die Ausgabe $a = P(e)$ **nach endlich vielen Schritten** an der vereinbarten Stelle in der vereinbarten Form abliefert. Diese **Laufzeit** ist allerdings von der Eingabe e abhängig.

Weitere mit dem Algorithmenbegriff zusammenhängende Begriffe :

- Analyse, Komplexität

  - bzgl. der Laufzeit
  - bzgl. des Speicherplatzverbrauchs
  - bzgl. des Implementierungsaufwands
- Effizienz und Optimierung
- Programmverifikation, Test

3 Berechenbarkeit

Intuitiv: Ein Problem heißt berechenbar, wenn dazu ein Algorithmus existiert.

D.h.: Wenn der Algorithmus zur Problemklasse $f : E \to A$ auf jeder zulässigen Eingabe $e \in E$ die zugehörige Ausgabe $f(e) \in A$ in endlich vielen Schritten (korrekt) berechnet und an der vereinbarten Stelle ablegt.

Algorithmenbegriff recht vage, Kontextabhängigkeit, verfügbare Sprachmittel nicht so ganz klar.

Einfach: Berechenbare Probleme: gib einen Algorithmus an.

Schwierig: Nicht berechenbare Probleme: Beweise, dass es keinen Algorithmus gibt. (Aber vielleicht gibt es solche Probleme auch gar nicht ??)

Dazu muss der Algorithmenbegriff noch etwas weiter formalisiert werden. Eliminiere dazu weitgehend Kontext- und Sprachabhängigkeit.

3.1 Beschränkung auf Funktionen $f : \mathbf{N} \to \mathbf{N}$

Kodierung endlicher Texte durch natürliche Zahlen. Codierung und Decodierung sind offensichtlich berechenbar. Zusammensetzung berechenbarer Funktionen ist berechenbar.

Frage: Welche Funktionen $f : \mathbf{N} \to \mathbf{N}$ sind berechenbar, d.h. für welche kann man einen Algorithmus aufschreiben (z.B. in Form eines C-Programms)?

1. Antwort: garantiert nicht für jede Funktion.

Argument: Die Zahl der Algorithmen ist abzählbar, die Zahl der Funktionen überabzählbar (jeder reellen Zahl $0 < z < 1$ kann eine Funktion f via $f(i) = a_i$ zugeordnet werden, wobei a_i die i-te Nachkommaziffer der Dezimalbruchdarstellung von z ist).

Antwort unbefriedigend, denn es gibt auch nur abzählbar viele Probleme, da auch dahinter Texte sind.

Das Berechenbarkeitsproblem: Gibt es Funktionen $f : \mathbf{N} \to \mathbf{N}$, die sich beschreiben, aber nicht berechnen lassen ?

Noch immer vage. Von hier aus gibt es mehrere Wege. Wir verfolgen nur einen:

3.2 WHILE-Programme

Was heißt berechnen? Wir brauchen Programmiersprache und Bibliotheksfunktionen, z.B. C und die GNU-Bibliotheken. Für unsere Zwecke (zu zeigen, dass es prinzipiell kein Programm gibt, das f berechnet) sollte das Instrumentarium möglichst klein und überschaubar sein.

Wir können auf vieles verzichten, z.B. Ein- und Ausgabe, wenn wir vereinbaren, wo das Programm die Eingaben findet und die Ausgaben ablegen soll.

Wir können uns auch viele Bequemlichkeiten schenken und Werte, die mehrfach benötigt werden, jedesmal neu berechnen. Brauchen also keine Unterprogramme.

Wir können uns auch einige der arithmetischen Funktionen schenken, z.B. `div` und `mod`:

```
int slowmod(int a, int b) {
  while(a>=b) a=a-b;
  return(a);
}
```

Definition WHILE-Programm

- Abzählbare Anzahl von Variablen (von denen natürlich nur endlich viele in einer Algorithmenbeschreibung verwendet werden können):

  Wir wollen grob vereinbaren:

$x_0, x_1, \ldots$	Variablen für die Eingabegrößen
$y_0, y_1, \ldots$	Variablen für die Ausgabegrößen
$z_0, z_1, \ldots$	Hilfsvariablen

- Wertzuweisungen

$x_i := x_j$	x_i den Wert von x_j zuweisen.
$x_i := c$	x_i den Wert der Konstanten $c \in \mathbf{N}$ zuweisen.
$x_i := inc(x_j)$	Funktionswert von $inc : n \mapsto n+1$ zuweisen.
$x_i := dec(x_j)$	Funktionswert von $dec : n \mapsto max(0, n-1)$ zuweisen.

- induktive Definition eines WHILE-Programms:

  - Eine Wertzuweisung ist ein WHILE-Programm
  - Sind P_1 und P_2 WHILE-Programme, so ist auch $P_1; P_2$ ein WHILE-Programm
  - Ist P ein WHILE-Programm und x eine Variable, so ist auch

    $$While\ x > 0\ do\ P\ end$$

    ein WHILE-Programm.

    Beachten Sie, dass dabei alles außer x und P syntaktischer Schnickschnack ist und wir auch kürzer hätten schreiben können $W(x, P)$.
  - Nur die so aufgebauten Programme sind WHILE-Programme.

- Semantik:

  $\mathbf{P_1; P_2}$

  Erst wird P_1 ausgeführt, danach (auf der entstandenen Variablenbelegung) P_2

  While $x > 0$ **do P end**

  Führt P so oft aus, bis x den Wert 0 hat.
  Dabei ist P im Normalfall ein mehrschrittiges Programm, das den Wert von x verändert.
  Hat x gleich zu Anfang den Wert 0, so wird P gar nicht ausgeführt.

Beispiele von WHILE-Programmen

ADD(a,b): Idee: Zähle x_1 herunter und parallel x_0 herauf.

Eingabe: $x_0 \leftarrow a, x_1 \leftarrow b$

```
while x1>0 do x1:=dec(x1); x0:=inc(x0) end;
y0:=x0
```

Ausgabe: $y_0 \rightarrow a + b$

SUB(a,b): Berechnet analog $a \div b := max(a - b, 0)$

GREATER(a,b): Idee: $a > b \Leftrightarrow a \div b \geq 1$

Eingabe: $x_0 \leftarrow a, x_1 \leftarrow b$

```
while x1>0 do x1:=dec(x1); x0:=dec(x0) end;
/* Nun ist x0 = a - b */
y0:=0;
while x0>0 do x0:=0; y0:=1 end;
```

Ausgabe: $y_0 \rightarrow \begin{cases} 1 & \text{wenn } a > b \\ 0 & \text{sonst} \end{cases}$

MULT(a,b): Idee: $a \cdot b = a + a + \ldots + a$ (b mal)

Eingabe: $x_0 \leftarrow a, x_1 \leftarrow b$

```
y0:=0;
while x1>0 do
  y0:=ADD(y0,x0); /* weiter zu verfeinern */
  x1:=dec(x1)
end
```

Ausgabe: $y_0 \rightarrow a \cdot b$

y0:=ADD(y0,x0):

```
z0:=y0; z1:=x0;
while z1>0 do z1:=dec(z1); z0:=inc(z0) end;
y0:=z0
```

Insgesamt also:

MULT(a,b): Idee: $a \cdot b = a + a + \ldots + a$ (b mal)

Eingabe: $x_0 \leftarrow a, x_1 \leftarrow b$

```
y0:=0;
while x1>0 do
    z0:=y0; z1:=x0;
    while z1>0 do z1:=dec(z1); z0:=inc(z0) end;
    y0:=z0;
    x1:=dec(x1)
end
```

Ausgabe: $y_0 \rightarrow a \cdot b$

SQRT(a): Berechnet $n \mapsto [\sqrt{n}]$.

Idee: $x = 0$. Zähle hoch, bis $x^2 > a$. Ergebnis ist dann $x - 1$.

Verwende MULT (zur Berechnung von x^2) und Idee aus GREATER (zum Test, ob $x^2 > a$).

Ergebnis nach mehreren Verfeinerungen des Entwurfs:

Eingabe: $x_0 \leftarrow a$

```
x1:=0; x0:=inc(x0); z0:=1;
while z0>0 do
  x1:=inc(x1); z0:=MULT(x1,x1); z0:=SUB(x0,z0);
end;
y0:=dec(x1);
```

Ausgabe: $y_0 \to [\sqrt{a}]$

Andere Programmkonstrukte kann man simulieren:

FOR-Schleife: **for i from a to b do P**

```
i:=a;
while b+1-i>0 do P; i:=inc(i) end;
```

oder genauer

```
z0:=i; z1:=inc(b); z2:=SUB(z1,z0);
while z2>0 do
  P; z0:=inc(z0); z2:=dec(z2);
end
```

IF-Verzweigung; **if** $x > 0$ **then** P_1 **else** P_2

```
z1:=1; z2:=x;
while z2>0 do P1; z2:=0; z1:=0 end;
while z1>0 do P2; z1:=0 end
```

3.3 Berechenbare und nicht berechenbare Funktionen. Das Halteproblem

Satz 3 *Jede Funktion $f : \mathbf{N} \to \mathbf{N}$, die mit einem C-Programm berechnet werden kann, kann auch mit einem WHILE-Programm berechnet werden.*

Es gibt eine Reihe anderer Berechenbarkeitsmodelle, z.B.

- Berechnung mit Turingmaschinen
- Berechnung mit GOTO-Programmen
- Berechnungen im RAM-Modell
- axiomatischer Zugang über primitiv-rekursive Funktionen

Satz 4 *Jede Funktion $f : \mathbf{N} \to \mathbf{N}$, die im Sinne eines dieser Berechenbarkeitsmodelle berechenbar ist, ist auch im Sinne aller anderen dieser Modelle berechenbar.*

Beweis ist (u.a.) Gegenstand der Vorlesung "Theoretische Informatik".

Churchsche These: *Jede im intuitiven Sinne berechenbare Funktion $f : \mathbf{N} \to \mathbf{N}$ ist WHILE-berechenbar.*

Gibt es beschreibbare, aber nicht berechenbare Funktionen?

Kein Gegenbeispiel: Die konstante Funktion

$$f(x) = \begin{cases} 1 & \text{wenn in der Dezimaldarstellung von } \pi \text{ jede Ziffer gleich häufig vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Zwar weiß man die Antwort nicht, aber natürlich ist die Funktion berechenbar. Eins der beiden Programme $\boxed{\texttt{y0:=0}}$ oder $\boxed{\texttt{y0:=1}}$ leistet das Geforderte.

Kein Gegenbeispiel: Funktionen $f : M \to \mathbf{N}$, die nur aud einer endlichen Teilmenge $M \subset \mathbf{N}$ definiert sind.

Programm: die (endliche) Wertetabelle aufschreiben.

Das Halteproblem

(Berechenbare) Kodierung endlicher Texte durch Zahlen.

Ist folgende Funktion berechenbar?

$WTEST : \mathbf{N}^2 \to \mathbf{N}$ mit

$$WTEST(M,e) = \begin{cases} 1 & \text{wenn das WHILE-Programm mit der Nummer } M \text{ auf der Eingabe} \\ & \text{mit der Nummer } e \text{ (kurz: } M(e)\text{) terminiert} \\ 0 & \text{sonst} \end{cases}$$

Nehmen wir es mal an

WTEST(M,e):

Eingabe: $x_0 \leftarrow M, x_1 \leftarrow e$

...

Ausgabe: $y_0 \to \begin{cases} 1 & M(e) \text{ terminiert} \\ 0 & \text{sonst} \end{cases}$

und betrachten das folgende WHILE-Programm

EVIL(M):

Eingabe: $x_0 \leftarrow M$

```
x1:=x0;
WTEST;
While y0>0 do y0:=y0 end
```

Was passiert bei WTEST(EVIL,EVIL) ?

$y0 = 0$? $\Rightarrow$ EVIL(EVIL) terminiert nicht. Das kann nur an der letzten While-Schleife liegen (denn WTEST terminiert immer, entweder mit $y0 = 0$ oder $y0 = 1$, wenn M und e zulässig sind).
$\Rightarrow$ $y0 > 0$

$y0 = 1$? $\Rightarrow$ EVIL(EVIL) terminiert, bleibt also insbesondere nicht in der letzten Schleife hängen.
$\Rightarrow$ $y0 = 0$

4 Der prinzipielle Aufbau eines Von-Neumann-Rechners

Es geht nur um ein prinzipielles Verständnis, wie ein Rechner aufgebaut ist.

Zweck eines Rechners: Maschinelle Abarbeitung von Algorithmen.

Universalitätsprinzip => Eingabe von Daten und Programm wichtig

Programm = schrittweises Abarbeiten von **Befehlen** mit relevanten **Daten**

Einheitliche Darstellung von Daten und Programmen im Rechner in **Speicherzellen** des **Arbeitsspeichers**.

Von-Neumann-Operationsprinzip kennt deshalb drei Datentypen: Befehl, Datum, Adresse

Dem Inhalt einer Speicherzelle sieht man nicht an, ob sie einen Befehl oder ein Datum enthält => Interpretation erfolgt in einer anderen Einheit, dem **Prozessor** (CPU)

4.1 Struktur eines Von-Neumann-Rechners

(Folie Rechneraufbau)

- Prozessor (CPU)
  interpretiert Befehle und führt sie aus
- Arbeitsspeicher
  enthält das auszuführende Maschinenprogramm und die dazugehörigen Daten
- Ein/Ausgabe-Einheit (E/A-Prozessor(en))
  organisiert die Kommunikation mit der Umwelt (periphere Geräte, Platten, Drucker). Kann bei kleineren Rechnern von der CPU mit übernommen werden (Interrupts)
- Verbindungswege (Busse)
  Austauschmedium für Adressen, Daten, Steuer- und Statusinformationen.

Prozessor besteht aus zwei Teilen:

- Steuerwerk (Folie Steuerwerk)
  Aufgaben:
  - Befehl holen
  - Befehlszähler inkrementieren
  - Befehl dekodieren und Adresse der zugehörigen Daten ermitteln

  Ein Befehl besteht aus einem Operationsteil und einem Adressteil

  Struktur: Enthält
  - Befehlsregister
  - Befehlszähler
  - Befehlsdecodierer
- Rechenwerk (Folie Rechenwerk)
  Aufgaben:

- Daten zum Befehl in die Register laden
- Daten zum Befehl verarbeiten
- Daten in den Speicher zurückschreiben

Struktur: Enthält

- Arbeitsregister zur lokalen Speicherung von Daten,
- Verknüpfungslogik (ALU = Arihtmetic Logic Unit) für
  - Transferoperationen (Laden, Speichern)
  - Boolesche Operationen
  - Arithmetische Verknüpfungen
  - Registermanipulationen
- Statusregister (Zero, Sign Carry, Overflow)

4.2 Ein-Adress-Rechner

Einfachstes Beispiel eines solchen Rechners mit folgender Spezifikation:

- Arbeitsspeicher enthält Code und Daten in Speicherzellen, die mit natürlichen Zahlen adressiert sind. Die Adresse 0 steht zwar zur Verfügung, wird aber nicht verwendet (außer als /dev/null).

  Unterscheide für Arbeitsregister

`AR`	Adresse des Registers
`<AR>`	Im Register AR gespeicherter Inhalt
`<<AR>>`	Im Register `<AR>` gespeicherter Inhalt

- In einem Befehl ist nur eine Adresse kodiert. Aufbau eines Befehls:

Operand	Adresse

  In der Simulation `simneu` erfolgt die Kodierung als $AT \cdot 2^4 + OT$.

- Prozessor verfügt nur über folgende Register:

Kürzel	Bezeichnung	Kommentar
BR	Befehlsregister	Befehl laden heißt ` =<<BZ>>`
RR	Resultatsregister	
ZR	Zustandsregister	`<RR>>0 <=> <ZR>=1`
BZ	Befehlszähler	

- Folgende Befehle stehen zur Verfügung:

Befehl	Code	Semantik	Kommentar
`H`	0	Halt	Auführung beenden
`+ - * / %`	1...5	`<RR>=<RR> o <<AT>>`	Arithmetik-Operationen wie in C
`=>`	6	`<<AT>>=<RR>, <RR>=0`	Speichern mit Löschen
`S`	7	`<BZ>=<AT>`	unbedingter Sprung
`S>`	8	`<BZ>=<AT>` if `<ZR>=1`	bedingter Sprung

4.3 Beispielprogramme

Im Kopf jedes dieser Programme steht als Kommentar die **Spezifikation des Programms**, d.h. die Funktion, die damit berechnet wird, sowie die Adressen der Ein- und Ausgabegrößen.

1. Berechnen von gcd(a,b) mit dem Euklidschen Algorithmus (`gcd.in`)

```
.START 101      ; Einsprungadresse
; Berechnung von gcd(a,b) mit dem Euklidschen Algorithmus
; Input  : a in Adresse 1, b in Adresse 2
; Output : g = gcd(a,b) in Adresse 10
;-----------------------
1       21      ; a
2       15      ; b
10       0      ; g
;---- Hilfsvariablen -----
20       1      ; r
21       1      ; Kopie a
22       1      ; Kopie b
; --- Programmcode -----
101  =>  0      ; loesche RR
102  +  1 ;
103  => 21 ; Kopie a
104  +  2 ;
105  => 22 ; Kopie b
106  +  22 ; lade b
107  S> 120 ; b > 0 ? weiter : a ausgeben
108  +  21 ; a laden, rr ist Null !
109  => 10 ; auf g ablegen
110  H  0 ; Ende
; ---- Schleife beginnt --
120  =>  0 ; rr=0
121  +  21 ; lade a
122  %  22 ; a mod b
123  => 20 ; speichern in r
124  +  22 ; lade b
125  => 21 ; speichern in a
126  +  20 ; lade r
127  => 22 ; speichern in b
128  S  106 ; Sprung zum Laden von b und Test
```

2. Bestimmen von $s(n) = \sum_{i=1}^{n} i$ (Summe der ersten n natürlichen Zahlen, `sum1.in`**)**

```
.START 1001        ; Einsprungadresse
; Berechnung von sum(i, i=1..n)
; Input  : n in Adresse 1
; Output : Summe in Adresse 10
;-----------------------
1      5           ; n
10     0           ; sum
;---- Hilfsvariablen -----
21     1           ; eine Eins
; --- Programmcode -----
1001   =>   0      ; rr initialisieren
1002   =>   10     ; sum:=0
1003   +    1      ; n laden
1004   S>   1020   ; n>0 ? weiter : halt
1005   H    0      ; Halt
```

```
; ---- Schleife beginnt --
1020   =>   1      ; n speichern
1021   +    1      ; n laden
1022   +    10     ; sum zu rr=n addieren
1023   =>   10     ; sum speichern
1024   +    1      ; n laden
1025   -    21     ; n-1
1026   S    1004   ; springe zurueck
```

3. Berechnen von $s = \sum_{i=1}^{n} a_i$. (sum2.in)

Problem: Die Maschine kennt keine indirekte Adressierung.

Der Befehl '+ a[i]' zum Laden des aktuellen Summanden muss also *während der Abarbeitung des Programms* dynamisch erzeugt werden.

```
.START 1001
; Berechnung von sum(a[i], i=1..n)
; Input  : n in Adresse 1, Startadresse von a in Adresse 2
; Output : Summe in Adresse 10
;------------------------
1      7           ; n
2     11           ; Startadresse von a
10     0           ; sum
;------------------------
; Datenfeld a[1..7]
11     1           ; a1
12     5           ; a2
13     6           ; a3
14     8           ; a4
15     7           ; a5
16     3           ; a6
17     2           ; a7
;------------------------
; Hilfsvariablen
21     1           ; eine Eins
22    16           ; Shift zur Kodierung von { code , addresse }
23     1           ; Code von '+'
24     0           ; Adresse des aktuellen a[i]
; --- Programmcode -----
1001   =>   0      ; rr initialisieren
1002   =>   10     ; sum:=0
1003   +    2      ; Startadresse von a laden
1004   =>   24     ; als aktuelle Adresse speichern
1005   +    1      ; n laden
1006   S>   1020   ; n>0 ? weiter : halt
1007   H    0      ; Halt
; ---- Schleife beginnt --
1020   =>   1      ; n speichern
1021   S    1200   ; aktuelles a[i] laden
1022   +    10     ; sum dazu addieren
1023   =>   10     ; als sum speichern
1024   +    1      ; n laden
```

```
1025   -    21     ; n-1
1026   S    1006   ; springe zurueck
;------------------------
; "aktuelles a[i] laden"
; Befehlszeile '+ a[i]' muss erst erzeugt werden
;------------------------
1200   =>   0      ;
1201   +    24     ; Adresse AT von a[i] laden
1202   *    22     ; berechne 16*AT
1203   +    23     ; berechne <BR>=16*AT+OT
1204   =>   1300   ; Codezeile speichern
1205   +    24     ; Adresse AT von a[i] laden
1206   +    21     ; Adresse des nächsten Summanden berechnen
1207   =>   24     ; und speichern
1208   S    1300   ; erzeugte Codezeile ausführen
;--- erzeugte Codezeile --
1300   H    0      ; wird von 1204 überschrieben
1301   S    1022   ; Springe zurück zu 1022
```

Dieses Beispiel zeigt, dass von-Neumann-Rechner in der Lage sind, während der Programmabarbeitung nicht nur Daten zu verändern, sondern auch neuen Code dynamisch zu erzeugen und danach abzuarbeiten.

Die hier fehlende Funktionalität der **indirekten Adressierung** (Zugriff auf `<<<AT>>>`) ist in gängigen Prozessoren Teil des Befehlssatzes.

4.4 Drei-Adress-Rechner

Spezifikation: Gegenüber dem Ein-Adress-Rechner sind in jeder Befehlszeile drei Adressen gespeichert. Zwei geben die Adressen der Argumente an, die dritte die Stelle, wohin das Ergebnis gespeichert werden soll. Das macht ein spezielles Zustandsregister überflüssig.

- Prozessor verfügt nur über folgende Register: Befehlsregister, Resultatsregister, Befehlszähler
- Aufbau eines Befehls:
- Folgende Befehle stehen zur Verfügung:

  arithmetische Operationen: `<<AT3>> = <<RR>> = <<AT1>>` $\circ$ `<<AT2>>`

  Sprung: `<BZ> = <AT1>`, wenn `<<RR>>` $\leq$ `0`, sonst `<BZ> = <AT2>`

  Damit können bedingte und unbedingte (`<AT1> = <AT2>`) Sprünge realisiert werden.

4.5 Beispielprogramm

Bestimmen von $s(n) = \sum_{i=1}^{n} i$

```
.START 1001
; Berechnung der Summe der ersten n Zahlen
; Input  : n in Adresse 1
; Output : Summe in Adresse 10
1      3            ; n
10     0            ; sum
```

```
; --- Hilfsvariablen ---
20     1           ; eine Eins
21     0           ; eine Null
; --- Programmcode ---
1001  -  1   1  10  ; sum:=0
1002  +  1  21   1  ; n laden
1003  S 1010 1020 0 ; n<=0 ? halt : contine
; ---- Zweig n <= 0 --
1010  H  0
; ---- Zweig n > 0 ---
1020  +  1  10  10  ; sum:=n+sum
1021  -  1  20   1  ; n:=n-1
1022  S 1003 1003 0 ; da n bereits geladen
```

4.6 Zusammenfassung Von-Neumann-Prinzipien

- Der Rechner besteht aus **fünf Funktionseinheiten**: Steuerwerk, Rechenwerk, Arbeitsspeicher, Eingabewerk, Ausgabewerk
- Die Struktur des Rechners ist unabhängig vom zu bearbeiteten Problem. Der Rechner ist ein **Universalrechner**: Zur Lösung eines Problems muss das Programm im Speicher abgelegt werden.
- Programme, Daten und Ergebnisse werden im selben Speicher abgelegt. Einer Speicherzelle sieht man nicht an, was für Informationen sie enthält.

  Teile des Programmcodes können sogar dynamisch generiert werden.
- Der Speicher ist in fortlaufend nummerierte Zellen unterteilt. Über die Adresse eine Speicherzelle kann deren Inhalt abgerufen werden.
- Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt. Durch Sprungbefehle kann von dieser Reihenfolge abgewichen werden.
- Es gibt zumindest
  - arithmetische Befehle
  - logische Befehle
  - Transportbefehle
  - bedingte Sprünge
- Alle Daten (Befehle, Adressen usw.) werden binär kodiert.

5 Binäre Repräsentation elementarer Datentypen

Bits und Bytes

Speicher im von-Neumann-Rechner enthält in Speicherzellen Bitmuster, denen man nicht ansieht, was sie darstellen sollen (keine Semantik!)

Kleinste Einheit: **1 Bit**, repräsentiert die kleinste Speichereinheit, die zwei Zustände (0 und 1) speichern kann.

Speicherzelle = Kleinste *adressierbare* Speichereinheit. Im Wesentlichen die Anzahl von Bits, die gleichzeitig über den Bus übertragen werden kann. Früher 8-Bit-Bus, heute 32- oder 64-Bit-Bus (bzw. noch ein paar mehr Leitungen, die die Konsistenz der Übertragung sichern)

Aus historischen und Kompatibilitätsgründen: kleinste *adressierbare* Speichereinheit **1 Byte** = 8 Bit

Heute Speichergrößen in KB, MB, GB. Beachte aber, dass die Umrechnung sich am Zweiersystem orientiert:

$1\ KB = 2^{10}\ Byte, 1\ MB = 2^{10}\ KB, 1\ GB = 2^{10}\ MB$ (Abweichung: +2.4%)

5.1 Elementare Datentypen

Gleiche Bitmuster können unterschiedliche Daten repräsentieren. Die Interpretation wird von den Programmen vorgenommen bzw. sind im Betriebssystem und der Prozessorarchitektur eingebrannt.

Eine vorgegebene Art der Interpretation eines Bitmusters bezeichnet man als **Datentyp**. Eine solche Interpretation zeichnet sich aus durch zwei Komponenten:

- die Größe des belegten Speicherbereichs (genauer: der **Wertevorrat** an Daten, der dort abgelegt werden kann)

  Beispiel: Wenn Integer 4 Byte (also 4*8=32 Bit) einnehmen, dann geht der Bereich von -2^{31} bis $2^{31} - 1$.

- die **Operationen**, die auf Instanzen dieses Datentyps möglich sind.

Liegt der Schwerpunkt auf den Eigenschaften, die die Operationen und Wertebereiche besitzen, dann spricht man von einem **abstrakten Datentyp**.

Steht die konkrete Darstellung in einer Programmiersprache im Vordergrund, dann spricht man von einem **konkreten Datentyp**.

Aufbau komplexerer Datentypen erfolgt rekursiv aus kleinsten Bausteinen. Diese kleinsten Bausteine heißen **elementare Datentypen**.

Elementare Datentypen in C

In C gibt es nur wenige elementare Datentypen:

`char`	ein Byte, kann ein Zeichen aus dem Zeichensatz der Maschine aufnehmen
`int`	vorzeichenbehafteter Integerwert
`short int`	vorzeichenbehafteter Integerwert
`long int`	vorzeichenbehafteter Integerwert
`float`	vorzeichenbehafteter Gleitkommawert
`double`	vorzeichenbehafteter Gleitkommawert
`long double`	vorzeichenbehafteter Gleitkommawert

Die Speichergröße dieser Typen ist implementationsabhängig und kann mit `sizeof` ermittelt werden. Programm `sizeof.c`

Schlüsselwort `int` kann in Zusammensetzungen weggelassen werden: `short` = `short int`

Für Integer-Speichergrößen gilt $\texttt{short} \leq \texttt{int} \leq \texttt{long}$ und $\texttt{short}, \texttt{int} \geq 2\ Byte$. Analoge Ungleichung für Gleitkommazahlen.

Integertypen: `char`, `short`, `int`, `long`

Wertebereich eines Integertyps, der n Bits belegt: $-2^{n-1} \ldots 2^{n-1} - 1$

Können mit `signed` und `unsigned` gekoppelt werden. Das bewirkt jedoch keine andere interne Darstellung, sondern nur eine andere Interpretation desselben Bitmusters (Wertebereich dann $0 \ldots 2^n - 1$).

5.2 Der Datentyp `char`

Nicht definiert, ob intern als `signed char` oder `unsigned char` interpretiert.

Kann 2^8 Zeichen darstellen, historisch wurde das 8. Bit jedoch teilweise für Steuerzwecke benutzt. Internationaler Standard ISO 7 Bit (ASCII=American Standard Code for Information Interchange) deshalb nur im Bereich der Zeichen $0 \ldots 2^7 - 1 = 127$ normiert. (Folie)

Bereich $128 \ldots 255$ unterschiedlich normiert (etwa 8 Bit ISO-Latin, verschiedene ISO-Standards für östliche Sprachen). Deshalb Probleme mit deutschen Sonderzeichen zwischen verschiedenen Computerplattformen.

Unicode ist eine normierte Erweiterung der 7 Bit und 8 Bit ISO Codes und kann alle international praktisch bedeutsamen Schriftzeichen darstellen. Diese Darstellung für Zeichen wird in Java verwendet. C verwendet den ASCII-Standard.

Daneben existieren weitere (ältere) Standards, etwa EBCDIC der Firma IBM (erweiterter BCD-Code für Buchstaben und Zeichen)

Achtung: Die Zahl 0 (`int`) und das Zeichen '0' (`char`) sind intern unterschiedlich repräsentiert.

Operationen: Ein- und Ausgabe-Operationen und Arithmetikoperationen (denn `char` ist ein Integertyp)

Wertebereich also $-2^7 \ldots 2^7 - 1$ (`signed char`) bzw. $0 \ldots 2^8 - 1$ (`unsigned char`)

Beispiel: (`chartest.c`)

```
char c,d;
c='F'; d=2*' ';
printf("F als Zeichen:     %c\n",c);
printf("F als Zahl:        %d\n",c);
printf("2*' ' als Zeichen: %c\n",d);
printf("2*' ' als Zahl:    %d\n",d);
```

```
  printf("'F'-'A' als Zahl:  %d\n",'F'-'A');
```

Ausgabe:

```
F als Zeichen:     F
F als Zahl:        70
2*' ' als Zeichen: @
2*' ' als Zahl:    64
'F'-'A' als Zahl:  5
```

Darauf aufbauend zusammengesetzter Typ **Zeichenkette**: Ist ein zusammenhängender Speicherbereich von char's, der durch seine erste Adresse referenziert wird: `char *s`. Ende einer Zeichenkette wird durch das Nullzeichen '\0' markiert. Keine eigene Datentypsbezeichnung.

Achtung: Der Speicher, den Zeichenketten belegen, muss in C explizit organisiert werden.

5.3 Binäre Repräsentation ganzer Zahlen

Verschiedene Arten der Darstellung möglich: Stockkerben, römische Zahlzeichen (additive Zahlsysteme). Uns interessieren nur:

Positionssysteme

Beispiel: $125 = 1 * 10^2 + 2 * 10^1 + 5 * 10^0$

Was steht links und was rechts? Datentyp String vs. Datentyp Zahl mit besonderem Augenmerk auf die verschiedenen Operationen auf den Datentypen.

Allgemein:

$$\pm[z_n z_{n-1} \dots z_0]_\beta = \pm \sum_{k=0}^{n} z_k \beta^k =: z$$

bezeichnet man als **Positionsdarstellung der Zahl z zur Basis** β, wenn $0 \leq z_i < \beta$. Ist darüber hinaus $z_n \neq 0$, so heißt die Darstellung n-stellig. Die einzelnen z_i heißen **Ziffern**.

Ist $\beta > 10$, so werden gewöhnlich für die Ziffern mit einem Zahlenwert > 9 die Buchstaben A,B,... verwendet, wobei kein Unterschied zwischen Klein- und Großbuchstaben gemacht wird.

Also genauer: $[125]_{10} = 1 * 10^2 + 2 * 10^1 + 5 * 10^0$

String in Zahl verwandeln

Beispiel: $[1628]_9 = 1 * 9^3 + 6 * 9^2 + 2 * 9 + 8 = ((1 * 9 + 6) * 9 + 2) * 9 + 8 = (15 * 9 + 2) * 9 + 8 = 137 * 9 + 8 = 1241$

Allgemeines Verfahren funktioniert analog (Horner-Schema).

Effizienz: Pro Ziffer eine Addition und eine Multiplikation, also $O(n)$, wenn n die Länge der Eingabe (Zahl der Ziffern von z) ist.

Genauere Analyse des gesamten Konversionsvorgangs: String $\Leftrightarrow$ Zeichen $\Leftrightarrow$ Ziffer $\Leftrightarrow$ Zahl mit den jeweiligen Operationen

Operationen `char2digit` und `digit2char`

Allgemeines Verfahren in MuPAD (Funktioniert bis Basis $a = 20$);

```
char2digit:=proc(c) begin text2int(c,20) end_proc;
digit2char:=proc(d) begin int2text(d,20) end_proc;

int_eingabe:=proc(a,s)
  local z;
  begin
  if a>20 then error("Basis zu groß") end_if;
  z:=0;
  for i from 0 to length(s)-1 do
    z:=a*z+char2digit(s[i]);
  end_for;
  z;
end_proc;
```

Beispiel:

```
int_eingabe(9,"1628");
                                   1241
int_eingabe(16,"BADECAFE");
                                3135163134
```

Zahl in String verwandeln

Beispiel: 2658 im Dreiersystem ausgeben.

Idee: $\sum_{i=0}^{n} z_i 3^i = z_0 + 3\left(\sum_{i=1}^{n} z_i 3^{i-1}\right)$

Letzte Ziffer kann also abgespalten werden, indem Quotient und Rest bei Division durch 3 bestimmt werden:

```
   2658 : 3 = 886 Rest 0
    886 : 3 = 295 Rest 1
    295 : 3 =  98 Rest 1
     98 : 3 =  32 Rest 2
     32 : 3 =  10 Rest 2
     10 : 3 =   3 Rest 1
      3 : 3 =   1 Rest 0
```

Also $2658 = [10122110]_3$. Probe:

```
int_eingabe(3,"10122110");
                                   2658
```

Allgemeines Verfahren in MuPAD:

```
int_ausgabe:=proc(a,z)
  local s;
  begin
  if a>20 then error("Basis zu groß") end_if;
  s:="";
  while (z>0) do
    s:=digit2char(z mod a).s; z:=z div a;
```

```
  end_while;
  s;
end_proc;

int_ausgabe(3,2658);
                                "10122110"
int_ausgabe(19,2658);
                                  "76H"
```

Rechnen in Positionssystemen

Beispiel $a = 5$:

Addition: $[134]_5 + [423]_5 = [1112]_5$

Übertrag kann immer nur 1 sein. Man braucht also bei der Addition nur ein Übertragsbit. Subtraktion geht analog.

Probe:

```
int_ausgabe(5,int_eingabe(5,"134")+int_eingabe(5,"423"));

                                  "1112"
```

Multiplikation: $[134]_5 * [423]_5 = [124342]_5$ wie beim üblichen schriftlichen Multiplizieren.

Dazu ist Multiplikationstabelle zu memorieren:

	1	2	3	4
1	1	2	3	4
2	2	4	11	13
3	3	11	14	22
4	4	13	22	31

```
int_ausgabe(5,int_eingabe(5,"134")*int_eingabe(5,"423"));

                                "124342"
```

Folgerung: Multiplikation in Zahlsystemen zur Basis a wird auf elementare Multiplikationen mit der Signatur $Digit \times Digit \to DoubleDigit$ zurückgeführt.

Multiplikation von Binärzahlen durch Addition und Shiftoperationen:

$13 * 5 = [1101]_2 * [101]_2 = [101000]_2 + [10100]_2 + [101]_2 = [1000001]_2 = 65$

Darstellung von Integertypen im Rechner

Breite n Bits.

Dann `unsigned` mit Wertebereich $0 \ldots 2^n - 1$. Darstellung durch Binärzahlen klar.

Beispiel (`unsigned char`, $n = 8$): $37 = [00100101]$

Addition und Multiplikation zweier solcher Zahlen kann zu Überlauf führen. Lässt man das Überlaufbit unberücksichtigt, so werden Vielfache von 2^n vernachlässigt, d.h. die Rechnungen sind $(mod\ 2^n)$ noch immer korrekt.

`signed` mit Wertebereich $-2^{n-1} \ldots 2^{n-1} - 1$.

Vorzeichenbit? Schlechte Lösung, weil 0 dann doppelte Darstellung hat.

Idee: Stelle $-a$ für $a \leq 2^{n-1}$ als $2^n - a$ dar. Dabei ist $2^n - a \geq 2^n - 2^{n-1} = 2^{n-1}$, also Vorzeichen auch an einem Bit ablesbar.

Beispiel (`signed char`, $n = 8$): $-37 \mapsto 256 - 37 = [11011011]$

Berechnung: $256 - 37 = 255 - 37 + 1 = [11111111] - [00100101] + 1$

Erste Operation kann durch bitweises Komplement ausgeführt werden (Einerkomplement), zweite Operation fügt zum Ergebnis Eins an (Zweierkomplement)

Rechnungen sind wieder ($mod\ 2^n$) korrekt. Wir können sogar auf eine spezielle Implementierung der Subtraktion verzichten, weil $a - b \equiv a + (2^n - b)\ (mod\ 2^n)$ einheitlich als Addition und Zweierkomplement (Vorzeichenwechsel) realisiert werden kann.

Zusammenfassung:

> Alle Rechnungen (Addition, Subtraktion, Multiplikation) auf Integertypen mit n Bits werden im Restklassenring ($mod\ 2^n$) exakt ausgeführt. Tritt dabei kein Bereichsüberlauf ein, so sind die Rechnungen auch in $\mathbf{Z}$ exakt.
>
> Der Typ `unsigned` interpretiert die Bitmuster im Dualsystem als minimales nichtnegatives Restesystem, der Typ `signed` als symmetrisches Restesystem.

Die Operationen `/` und `%` sind für $a, b > 0$ als Quotient und Rest nach dem Satz über Division mit Rest eindeutig definiert. Auf negative Zahlen werden sie nach folgenden Regeln erweitert:

$$(-a)/b = -(a/b) \quad (-a)\%b = -(a\%b) \quad a/(-b) = -(a/b) \quad a\%(-b) = a\%b$$

In allen Fällen gilt die Beziehung $a = (a/b) * b + a\%b$.

5.4 Binäre Repräsentation rationaler Zahlen

Beschränken uns auf rationale Zahlen $\frac{z}{n}$, für die $z, n > 0$ und $gcd(z, n) = 1$ gilt (also reduzierte gebrochene Zahlen).

1. Darstellung: Als Paare (z, n) natürlicher Zahlen mit den üblichen Rechenregeln, wobei am Ende der Rechnungen der Ergebnisbruch jeweils zu reduzieren ist.

Datentyp `Rational` mit den entsprechenden Operationen + - * /, die sich auf das Rechnen mit ganzen Zahlen zurückführen lassen.

2. Darstellung: Endliche Festkommadarstellung zur Basis a (im Weiteren immer $a \in \mathbf{N}$ und $a > 1$).

$$\frac{z}{n} = [z_k \ldots z_0.z_{-1} \ldots z_{-l}]_a = \sum_{i=-l}^{k} z_i\, a^i = [z_k \ldots z_0]_a + [z_{-1} \ldots z_{-l}]_a \cdot a^{-l}$$

Mit der letzteren Beziehung kann die rationale Zahl $\frac{z}{n}$ berechnet werden, die durch die Festkommadarstellung repräsentiert wird. Verwendet nur Operationen auf und bereits bekannte Darstellungen von ganzen Zahlen.

Beispiel:

$[10101.101]_2 = [10101]_2 + [101]_2 \cdot 2^{-3} = 21 + \frac{5}{8} = \frac{173}{8}$

```
text2int("10101",2)+text2int("101",2)*2^(-3)
```

173/8

```
float(%);
```

21.625

Satz 5 *Ein Bruch $\frac{z}{n}$ mit $gcd(z,n) = 1$ besitzt genau dann eine endliche Festkommadarstellung zur Basis a, wenn n ein Teiler einer Potenz von a ist.*

Beweis: Übungsserie.

Die Umrechnung in die umgekehrte Richtung kann wie folgt vorgenommen werden:

Ist $z = q \cdot n + r$ die Zerlegung nach dem Satz von der Division mit Rest, so ergibt sich der Vorkommaanteil als $q = [z_k \ldots z_0]_a$ und der Nachkommaanteil als Darstellung von $\frac{r}{n}$.

Sei also $z < n$. Dann gilt

$$a \cdot \frac{z}{n} = [z_{-1} \,.\, z_{-2} \ldots z_{-l}]_a$$

Die nächste Ziffer z_{-1} ergibt sich also aus Division mit Rest von $a\,z : n$. Daraus kann ein algorithmisches Verfahren abgeleitet werden, das eine vorgegebene endliche Zahl von Nachkommaziffern der Festkommadarstellung von $\frac{z}{n}$ zur Basis a berechnet. (Zusatzaufgabe, Übungsserie)

Analogie zur schriftlichen Division im Zehnersystem: $2 : 3 = [0.666\ldots]_{10}$.

3. Darstellung: Als Festkommadarstellung zur Basis a mit Periode.

Beispiele (mit MuPAD, `float` und DIGITS:=50): $\frac{2}{3} = [0.666\ldots]_{10}, \frac{7}{15} = [0.4666\ldots]_{10}, \frac{1}{7} = [0.\overline{142857}]_{10}$

Allgemein

$$\frac{z}{n} = [z_k \ldots z_0 . z_{-1} \ldots z_{-l}\overline{z_{-l-1} \ldots z_{-l-m}}]_a$$

Vorkomma- oder ganzzahliger Anteil: $g = [z_k \ldots z_0]_a$, Vorperiode (der Länge l): $v = [z_{-1} \ldots z_{-l}]_a$, Periode (der Länge m): $p = [z_{-l-1} \ldots z_{-l-m}]_a$, wobei wir die Periode $[\overline{(a-1)}]_a$ ausschließen. (Grund: $[0.\overline{(a-1)}]_a = 1$, Beispiel: 9-er Periode im Dezimalsystem).

Enthält keine Pünktchen mehr, ist also beschreibungsendlich! Es gilt

$$\frac{z}{n} = g + v \cdot a^{-l} + p \cdot (a^{-l-m} + a^{-l-2m} + a^{-l-3m} + \ldots) = g + v \cdot a^{-l} + p \cdot a^{-l} \cdot \frac{1}{a^m - 1}$$

Beispiel: $[0.4\overline{6}]_{10} = 0 + 4 \cdot 10^{-1} + 6 \cdot 10^{-1} \cdot \frac{1}{9} = \frac{42}{90} = \frac{7}{15}$

Oder anders: $x = [0.4\overline{6}]_{10}, 10 \cdot x = [4.6\overline{6}]_{10}, 10 \cdot x - x = 9 \cdot x = [4.2]_{10}, x = \frac{4.2}{9} = \frac{7}{15}$

Ambiguität der Darstellung:

$[g \,.\, v\overline{p}]_a = [g \,.\, vp\overline{p}]_a = [g \,.\, v\overline{pp}]_a$

Im ersten Fall ist die Vorperiode zu lang. Verkürzung möglich gdw. letzte Ziffer der Vorperiode = letzte Ziffer der Periode.

Im zweiten Fall ist die Periode zu lang. Verkürzung möglich gdw. "Periode periodisch".

Satz 6 *Jede gebrochene Zahl $\frac{z}{n}$ besitzt bzgl. jeder Basis a ($a \in \mathbf{N}$ und $a > 1$) eine Festkommadarstellung mit Periode. Diese Darstellung ist eindeutig, wenn Vorperiode und Periode jeweils minimale Länge haben.*

Beweis : OBdA $z < n$, denn wir können wie beschrieben den ganzzahligen Anteil abspalten.

Sei zunächst $gcd(a,n) = 1$. Dann besitzt $\frac{z}{n}$ eine rein periodische Festkommadarstellung $[0.\overline{p}]_a$ und die minimale Periodenlänge ergibt sich als

$$m = \min(M) \quad \text{mit} \quad M = \{r > 0 \,:\, a^r \equiv 1 \; (mod\; n)\}$$

Beispiel: $\frac{1}{7} = [0.\overline{142857}]_{10}$

> Ist $\frac{z}{n} = [0.\overline{p}]_a$ eine rein periodische Darstellung mit Periodenlänge m, so gilt $\frac{z}{n} = \frac{p}{a^m-1}$, also $n \,|\, z(a^m - 1)$ und wegen $gcd(z,n) = 1$ auch $n \,|\, a^m - 1$, d.h. $m \in M$.
> Beispiel: $\frac{1}{7} = \frac{142857}{999999}$ und somit $10^6 \equiv 1 \; (mod\; 7)$
> Ist $m \in M$, so gilt $a^m - 1 = n \cdot q$ und
>
> $$\frac{z}{n} = \frac{z \cdot q}{n \cdot q} = (z \cdot q) \cdot \frac{1}{a^m - 1} = [0.\overline{[z \cdot q]_a}]_a,$$
>
> hat also eine rein periodische Festkommadarstellung mit (nicht unbedingt minimaler) Periodenlänge m.
>
> Beispiel: Ist $10^6 \equiv 1 \; (mod\; 7)$, so nimm $q = \frac{10^6-1}{7} = 142857$. Hier ist $z = 1$.
>
> Die Menge M ist nicht leer, denn es gilt nach dem Satz von Euler $a^{\phi(n)} \equiv 1 \; (mod\; n)$, wobei $\phi(n)$ die Eulersche ϕ-Funktion ist, die die primen Restklassen modulo n zählt.

Allgemeiner Fall: Beispiel $\frac{7}{15} = [0.4666\ldots]_{10}$

Zerlege $n = n_1 \cdot n_2$, so dass $n_1 | a^l$ für ein $l > 0$ und $gcd(n_2, a) = 1$. Diese Zerlegung ist eindeutig.

Beispiel: $15 = 3 \cdot 5, n_1 = 5, n_2 = 3$

Dann gilt $a^l = n_1 \cdot q'$ und

$$a^l \frac{z}{n} = \frac{a^l\, z}{n_1\, n_2} = \frac{q' z}{n_2} = [g.\overline{p}]_a$$

nach dem ersten Teil des Satzes. Die gesuchte Festkommadarstellung von $\frac{z}{n}$ ergibt sich daraus durch Kommashift um l Positionen nach links.

Beispiel: $\frac{7}{15} = \frac{70}{15} \cdot 10^{-1} = \frac{14}{3} \cdot 10^{-1} = [4.\overline{6}]_{10} \cdot 10^{-1} = [0.4\overline{6}]_{10}$ □

4. Darstellung: Näherungsweise Darstellung in endlicher Festkommadarstellung zur Basis a.

Beispiele: $\frac{2}{3} \sim [0.66666]_{10}, \frac{7}{15} \sim [0.46666]_{10}$

Allgemein:

$$q := [z_k \ldots z_0.z_{-1} \ldots z_{-l}]_a$$

heißt *abgeschnittene endliche Festkommadarstellung* mit l Nachkommastellen zur Basis a von $\frac{z}{n}$, wenn

$$0 \leq \frac{z}{n} - q < a^{-l}$$

gilt.

Satz 7 *Jede gebrochene Zahl $\frac{z}{n}$ besitzt eine eindeutig bestimmte abgeschnittene endliche Festkommadarstellung zur Basis a mit l Nachkommastellen.*

Beweis : $0 \leq \frac{z}{n} - q < a^{-l}$ gdw. $0 \leq a^l \cdot z - a^l \cdot q \cdot n < n$. $a^l \cdot q = [z_k \ldots z_0 z_{-1} \ldots z_{-l}]_a$ ist also der (nach dem Satz von der Division mit Rest eindeutig bestimmte) Quotient $a^l \cdot z \; div \; n$. □

Beispiel: $\frac{2}{3}$ soll im 7-er-System auf 5 Stellen nach dem Komma genau berechnet werden:

$7^5 \cdot 2 \; div \; 3 = 33614 \; div \; 3 = 11204 = [44444]_7$. Also gilt $\frac{2}{3} \sim [0.44444]_7$ (und sogar $\frac{2}{3} = [0.\overline{4}]_7 = \frac{4}{6}$).

```
7^5*2 div 3;
                                  11204
int2text(%,7);
                                 "44444"
```

5.5 Gleitkommazahlen

Rationale Zahlen kann man (theoretisch) exakt im Computer darstellen (wir hatten die Dst. z/n sowie mit Vorperiode und Periode betrachtet. Andere Darstellung: Z.B. als endlicher Kettenbruch. Beispiel 109/257=[0;2,2,1,3,1,7])

Bei der Darstellung reeller Zahlen im Computer sind Näherungen auch theoretisch unvermeidlich (Überabzählbarkeit im Intervall [0,1]). Praktisch spielen sie auch bei rationalen Zahlen schon eine Rolle.

Näherungsdarstellungen orientieren sich an der näherungsweisen endlichen Festkommadarstellung zur Basis a. Generelle Situation: $\overline{z}$ Näherungswert für z.

Absolute und relative Genauigkeit:

$|\overline{z} - z| < a^{-p}$ absolute Genauigkeit (p Stellen nach dem Komma im Positionssystem zur Basis a). Tritt bei Festkommadarstellung auf.

$|\frac{\overline{z}-z}{z}| < a^{-p}$ relative Genauigkeit (p gültige Stellen im Positionssystem zur Basis a)

Tritt bei Gleitkommadarstellung auf:

$$\pm[z_0 \,.\, z_1 z_2 \ldots z_{p-1}]_a \times a^k$$

Bezeichnungen: Mantisse (oder Signifikant), Exponent, p Stellenzahl, normalisierte Darstellung

Beispiele:
$a = 10, p = 3 : 0.1 = 1.00 \times 10^{-1}$
$a = 2, p = 24 : [0.1]_{10} = [1.10011001100110011001100]_2 \times 2^{-4}$

Berechnung mit MuPAD:

```
2^30 div 10;
                              107374182
int2text(%,2);
                  "110011001100110011001100110"
length(%);
                                 27
```

Bei fixiertem Exponenten k gibt es $(a-1) \cdot a^{p-1}$ verschiedene Mantissen, die normalisierten Zahlen x entsprechen, die im Intervall $a^k \leq x < a^{k+1}$ gleichverteilt sind. Der Abstand benachbarter Mantissen beträgt $m_k = a^{k-p+1}$. Beispiel: 1.00 und 1.01 ($k = 0, p = 3$, Abstand ist 10^{-2}).

Bezeichnung: **ulps** (units in the last place). Rundet man Zahlen zur nächstgelegenen Mantisse, so beträgt der (absolute) Fehler also maximal 0.5 ulps. Die Größe a^{-p} bezeichnet man auch als **Maschinen-Epsilon** m_ε. Für $k = 0$ gilt $1\,ulps = a \cdot m_\varepsilon$. Das ist der Abstand zwischen 1 und der nächstgrößeren darstellbaren Zahl.

Für unterschiedliche k sind die Abstände benachbarter Mantissen verschieden. Es gilt $m_{k+1} = a \cdot m_k$. ulps ist also ein relatives Maß für den absoluten Fehler!

Interessant ist auch der relative Fehler. Wegen $a^k \leq x < a^{k+1}$ kann dieser trotz Gleichverteilung der Mantissen um den Faktor a schwanken. (Bezeichnung: Wobble)

Beispiel:
$x = 1.00$, relat. Fehler $0.005/x = 0.005$
$x = 9.99$, relat. Fehler $0.005/x = 0.0005$

Aus beiden Gründen ist $a = 2$ besonders angenehm.

In praktischen Anwendungen beschränkt man auch die Zahl der Bits für den Exponenten $e_{min} \leq k \leq e_{max}$. Damit wird der Bereich der darstellbaren (normalisierten) Gleitkommazahlen eingeschränkt. Insbesondere liegt die 0 außerhalb des darstellbaren Bereichs und muss deshalb besonders behandelt werden.

Bei Rechnungen sind deshalb neben **Rundungsfehlern** auch Bereichsüberschreitungen denkbar, und zwar sowohl als **Überlauf** (Overflow – Ergebnis x mit $|x| \geq a^{e_{max}+1}$) als auch als **Unterlauf** (Underflow – Ergebnis x mit $0 < |x| < a^{e_{min}}$).

Bei Unterlauf könnte man generell zur 0 runden, aber das bringt Probleme mit sich, da das "Loch bei der Null" (Zero Threshold) ziemlich groß ist. Sei $k = e_{min}$. Bild in der Nähe der Null.

5.5.1 Rechnen mit Gleitkommazahlen

Alle Beispiele mit $a = 2, p = 6$.

Addition und **Subtraktion**: $m_1 \times a^{e_1} \pm m_2 \times a^{e_2}$

Vorgehen:

- Normieren der Mantissen auf $e = \max(e_1, e_2)$ durch Shift.
- Ausführen der (ganzzahligen) Operation auf den Mantissen.
- Evtl. Renormierung.

Beispiel: $1/7 + 1/13$

```
mantisse:=proc(u) begin int2text(floor(2^30*u),2) end_proc;
zahl:=proc(u,e) begin float(text2int(u,2)*2^(e-5)) end_proc;

mantisse(1/7);
                          "1001001001001001001001001001001"
mantisse(1/13);
                           "1001110110001001110110001001110"
```

$[1.00100] \times 2^{-3} + [1.00111] \times 2^{-4} = ([1.00100] + [0.10011]) \times 2^{-3} = [1.10111] \times 2^{-3}$

Vergleich des Ergebnisses:

```
zahl("110111",-3);
                                      0.21484375
float(1/7+1/13);
                                     0.2197802198
float(2^-8);
                                      0.00390625
```

Fehler also in der Größenordnung von 1 ulps.

Beispiel: $3/4 - 5/7$

```
mantisse(3/4);
```

```
                     "110000000000000000000000000000"
mantisse(5/7);
                     "101101101101101101101101101101"
```

$[1.10000] \times 2^{-1} - [1.01101] \times 2^{-1} = [0.00011] \times 2^{-1} = [1.10000] \times 2^{-5}$

```
zahl("110000",-5);
                                0.046875
float(3/4-5/7);
                                0.03571428571
float(2^-10);
                                0.0009765625
```

Fehler in der Größenordnung von 11 ulps.

Bemerkung: Bei Addition/Subtraktion können drastische Fehler durch Stellenauslöschung auftreten.

Multiplikation: $m_1 \times a^{e_1} * m_2 \times a^{e_2}$

Vorgehen:

- Exponenten addieren $e = e_1 + e_2$.
- Mantissen multiplizieren $m = m_1 \cdot m_2$.
- Ergebnis evtl. renormieren.

Bei der Multiplikation treten keine gravierenden Fehler auf, da für die Mantissen $1 \leq m_1, m_2 < a$ gilt und folglich $1 \leq m < a^2$, also höchstens ein Shift um -1 erfolgt.

Beispiel: $1/7 * 1/13$

$[1.00100] \times 2^{-3} * [1.00111] \times 2^{-4} = ([1.00100] * [1.00111]) \times 2^{-7} = [1.01011] \times 2^{-7}$ wenn Rundung durch einfaches Abschneiden erfolgt. Sonst $= [1.01100] \times 2^{-7}$

```
zahl("101100",-7);
                                0.0107421875
zahl("101011",-7);
                                0.01049804688
float(1/7*1/13);
                                0.01098901099
float(2^-12);
                                0.000244140625
```

Fehler also 1 bzw. 2 ulps.

Division vollkommen analog (Differenz der Exponenten und Quotient der Mantissen). Auch hier wird die Ergebnismantisse um höchstens eine Stelle zu renormieren sein.

Pathologien:

Operationen auf Gleikommazahlen erfüllen nicht immer die üblichen Gesetze der Arithmetik. Z.B. gilt das Assoziativgesetz nicht immer:

$a = 10, p = 3, x = 10^9, y = -10^9, z = 1 : (x + y) + z = 1 \neq x + (y + z) = 0$

5.5.2 Der IEEE-Standard 754

Der IEEE-Standard 754 für binäre Arithmetik ($a = 2$) spezifiziert 4 Gleitkommaformate: single, single extended, double und double extended. Die meisten PC (auf x87-Basis) unterstützen die Formate single (`float`), double und double extended (`long double`).

p-stellige Mantisse, q-stelliger Exponent. Damit etwa $-e_{min} = e_{max} = 2^{q-1}$ und maximal darstellbare Zahlen in der Größenordnung $2^{2^{q-1}}$.

Format	Bytes	p	m_ε	q	Bereich	Bitsumme
Single	4	24	$2^{-24} \sim 10^{-7}$	8	$10^{\pm 38}$	32
Double	8	53	$2^{-53} \sim 10^{-16}$	11	$10^{\pm 308}$	64
Double extended	10	64	$2^{-64} \sim 10^{-19}$	15	$10^{\pm 4932}$	79

Details: **hidden bit** (damit Mantisse nur $p-1$ Stellen und Platz für das Vorzeichenbit), außer bei double extended.

Exponentendarstellung durch Verschiebung: Exponenten $e \in [e_{min}, e_{max}]$ werden dargestellt als $e + k \in [0, 2^q - 1]$. Da 2^q gerade ist, kann man $-e_{min} = e_{max}$ nicht erreichen. Man wählt k so, dass $-e_{min} = e_{max} - 1$ gilt, womit das Reziproke $1/z$ der kleinsten darstellbaren Zahl $z = 2^{e_{min}}$ keinen Überlauf erzeugt. Also $k = 2^{q-1} - 1$. Dann erzeugt das Reziproke des größten Zahl zwar einen Unterlauf, aber der ist gewöhnlich nicht so schwer wiegend.

Verschiebung rechnet sich bei Mantissenshifts für Addition und Subtraktion wieder raus. Vorzeichen des Exponenten kann am ersten Bit erkannt werden: $1 \Rightarrow e > 0, 0 \Rightarrow e \leq 0$

Die beiden extremen Werte für den Exponenten (also 0 und $2^q - 1$) werden für spezielle Zwecke genutzt, so dass gilt:

Format	e_{min}	e_{max}
Single	-126	127
Double	-1022	1023
Double extended	-16382	16383

Rechnungen mit Gleitkommazahlen können zu Überlauf führen oder andere Fehler auslösen (Division durch 0, negativer Radikand, Logarithmus aus negativer Zahl usw.). Bei all diesen Ereignissen werden Statusbits gesetzt, die Rechnungen aber nicht abgebrochen. Grund: Das Tabellierungsproblem. Beim Berechnen von einer Wertetabelle einer Funktion kann es durch Rundungsfehler zu Überlauf oder den anderen Fehlern kommen. In diesem Fall werden "Nichtzahlen" generiert (und auch von `printf` ausgegeben!).

Nichtzahlen: Der IEEE-Standard spezifiziert die folgenden Nichtzahlen: $\pm\infty$, `NaN` (not a number). Sie sind durch den Exponenten $e_{max} + 1$ (alle Exponentenbits 1) dargestellt. Im ersten Fall ist die Mantisse $m = 0$, im zweiten Fall $m \neq 0$. Die Mantisse kann die Art des Fehlers codiert enthalten, was aber nicht standardisiert ist.

Für `infinity` und `NaN` gelten die offensichtlichen Rechenregeln, also etwa

$$
\begin{aligned}
&a + \infty = \infty \\
&a \cdot \infty = \infty \quad \text{wenn } a > 0 \\
&a \cdot \infty = -\infty \quad \text{wenn } a < 0 \\
&\sqrt{\infty} = \infty \\
&0 \cdot \infty = NaN \\
&0/0 = \infty/\infty = NaN
\end{aligned}
$$

Denormalisierte Zahlen: Zahlen mit dem Exponenten $e = e_{min} - 1$, also Exponentenbits alle 0, werden nicht als normalisierte Zahlen interpretiert.

Beispiel, dass sonst $x \neq y$, aber $x - y = 0$ sein kann. Somit kann folgende C-Code-Zeile zum Abbruch führen:

```
if (x!=y) z=1/(x-y);
```

signed zero: Zwei Darstellungen von $x = 0.0$. Es gilt $+0.0 = -0.0$, die beiden Darstellungen können aber über die Berechnung von $1/x = \pm\infty$ unterschieden werden.

Zusammenfassung:

Exponent	Bruchteil	stellt dar	Name
$e = e_{min} - 1$	$f = 0$	± 0	vorzeichenbehaftete Null
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$	denormalisierte Zahlen
$e_{min} \leq e \leq e_{max}$		$1.f \times 2^{e}$	normalisierte Zahlen
$e = e_{max} + 1$	$f = 0$	$\pm\infty$	Infinity
$e = e_{max} + 1$	$f \neq 0$	NaN	echte Nichtzahlen

6 Höhere Programmiersprachen

6.1 Vom Algorithmus zum Computerprogramm

Problem: geringe Konkretheit, implizite Darstellung

Algorithmus: Aufarbeitung des Problems bis zu einer konstruktiven Lösung. Stärker formal, aber noch nicht unbedingt formalisiert.

Implementierung: Umsetzung eines Algorithmus in ein Computerprogramm.

Selten werden Algorithmen, insbesondere wenn sie kompliziertere Struktur haben, direkt in Computerprogramme umgesetzt. Selbst für die Simulation einer Ein-Adress-Maschine hatten wir nicht direkt mit bitcodierten Befehlsfolgen gearbeitet, sondern uns einer etwas verständlicheren Zwischensprache bedient, die dann vom Werkzeug `simneu` in die zugehörige bitcodierte Befehlsfolge übertragen (Teil 1 der Ausgabe) und abgearbeitet (Teil 2 der Ausgabe) wurde.

(Aus [Ernst]) Das Programmieren auch in anderen maschinennahen Sprachen (Assembler) ist, trotz der dort verfügbaren ausgefeilteren Sprachelemente wie mnemonischer Bezeichnungen für die Funktionen, frei wählbarer Namen, Makros und Unterprogrammen sehr mühsam und Zeit raubend. Die resultierenden Programme sind meist lang, unübersichtlich und schwer zu durchschauen. Das liegt daran, dass die grundlegenden Befehle solcher Sprachen maschinenspezifisch sind, aber gerade mit dem zu bearbeitenden Problem nichts zu tun haben und insofern vom Programmierer früher oder später als Ballast empfunden werden.

Man hat daher schon früh nach Wegen gesucht, eine Zwischenebene zwischen Programmierer und Computer einzubauen, die ersteren von diesen maschinenspezifischen Details abschirmt und und eine *problemorientierte* Programmentwicklung ermöglicht. (Konzept der **virtuellen Maschine**)

Programme organisieren das **Zusammenspiel von Daten und Algorithmen**. Programmierer müssen deshalb für eine solche Abstraktion Beschreibungsmöglichkeiten für beide Komponenten haben, die überdies von entsprechenden Werkzeugen automatisch in Bitmuster für Daten und Befehle übertragbar sein müssen, um sie auf einem Computer abzuarbeiten. Das erfordert eine starke **Formalisierung dieser Beschreibungssprachen**.

Trotz ihrer Formalisierung sind die heute verfügbaren Sprachen der menschlichen Ausdrucksweise angepasst, etwa durch die enge Anlehnung an die Schreibweise mathematischer Formeln. Je nachdem, auf welchem Abstraktionsniveau diese Zwischenebene angesiedelt ist, spricht man von *höheren* oder *niederen* oder *maschinennahen* Sprachen. Mittlerweile hat sich eine ganze Reihe wohl durchdachter Sprachkonzepte für die verschiedensten Anwendungen etabliert.

Quellprogramm und **Maschinenprogramm**

Damit ein in einer höheren Programmiersprache geschriebenes Programm auf einem Rechner zur Ausführung kommen kann, muss es zunächst in die dem Rechner direkt verständliche Maschinensprache übertragen werden.

Maschinenprogramm und **Ausführung**, **Übersetzungszeit** und **Laufzeit**.

6.1.1 Programmiersprachen

Programmiersprachen sind Sprachen zur Formulierung von Rechenvorschriften, d.h. Datenstrukturen und Algorithmen, die von einem Computer ausgeführt werden können. Sie bilden damit die wichtigste Schnittstelle zwischen Benutzern und Computern. Jeder Benutzer muss die Bearbeitung von Problemen, die einem Computer übergeben werden sollen, in einer Programmiersprache formulieren.

Programmiersprachen müssen **formalisiert** sein, um Mehrdeutigkeiten zu vermeiden und die weitere automatische Verarbeitung durch Interpreter bzw. Compiler zu ermöglichen. Dazu muss spezifiziert werden, welche Zeichenfolgen als Programme zugelassen sind (**Syntax**) und was diese Zeichenfolgen auf dem Rechner bewirken (**Semantik**). Die Syntax der meisten Programmiersprachen wird durch kontextfreie Grammatiken mit umgangssprachlichen Zusätzen beschrieben und ist stärker formalisiert, um die Übersetzung in dem Computer verständliche Bitfolgen zu ermöglichen. Die Semantik wird häufig stärker umgangssprachlich beschrieben, kann jedoch auch bis zu einem gewissen Grad formalisiert werden.

Je nach dem Grad, mit dem die Hardware bei der Programmierung beachtet werden muss, **klassifiziert** man Programmiersprachen in

- Maschinensprachen (direkte Programmierung der Hardware),
- niedere, maschinenorientierte oder Assemblersprachen (Programmierung der Hardware, aber mit symbolischen Namen)
- und höhere oder problemorientierte Sprachen (von der Hardware weitgehend unabhängig und an den zu bearbeitenden Problemfeldern orientiert).

Jeder Programmiersprache liegt ein bestimmtes Konzept oder Denkschema zu Grunde. Auf der Basis dieser Konzepte unterteilt man die höheren Programmiersprachen in folgende Kategorien:

- iterative Programmiersprachen (Programm als Folge von Befehlen, Daten gespeichert in einer Menge von Variablen)
- funktionale Programmiersprachen (Programme als Folge von Funktionsaufrufen, in reiner Form treten Variable nur als formale Parameter in Funktionsdeklarationen auf)
- prädikative oder logische Programmiersprachen (Programme als Menge von Fakten und Regeln, in der die virtuelle Maschine selbstständig und autonom navigieren kann)
- objektorientierte Programmiersprachen (deutlichere Strukturierung der Daten als Objekte und der Datentypen als Klassen)

6.1.2 Der Programmbegriff imperativer Programmiersprachen

Das klassische Konzept imperativer Programmiersprachen geht von der Vorstellung eines Programms als schrittweise Transformation einer Menge von **Eingabedaten** in eine Menge von **Ausgabedaten** nach einem vorgegebenen **Algorithmus** aus.

Diese Transformation erfolgt durch *Abarbeiten einzelner Programmschritte*, in denen die Daten entsprechend den angegebenen Instruktionen verändert werden. Den Zustand der Gesamtheit der durch das Programm manipulierten Daten bezeichnet man auch als den *Programmstatus*.

Die Programmschritte werden gewöhnlich in *Anweisungen und Deklarationen* unterteilt, wobei Anweisungen den Programmstatus ändern, Deklarationen dagegen nicht, sondern lediglich Bedeutungen von Bezeichnern festlegen.

Anweisungen setzen sich aus elementaren Anweisungen zusammen, die durch entsprechende *Kontrollstrukturen* miteinander verbunden sind. Elementare Anweisungen können *Zuweisungen* oder *Funktionsaufrufe* sein. Durch erstere wird direkt der Wert einer Variablen geändert, durch zweitere erfolgt die Änderung des Programmstatus als Seiteneffekt.

Bei **Deklarationen** unterscheidet man gewöhnlich zwischen Deklarationen von *Datentypen*, *Variablen*, *Funktionen* und *Prozeduren*. Jede solche Deklaration verbindet mit einem **Bezeichner** eine gewisse Bedeutung. Neben Variablenbezeichnern gibt es (z.B. in Pascal oder C) noch Bezeichner für Funktionen, Typen, Marken und Konstanten.

Zur praktischen Arbeit mit Programmen benötigt man neben Kenntnissen der Programmiersprache auch entsprechende **Programmierwerkzeuge**, mit denen sich Programme erstellen, testen, warten, optimieren und ausführen lassen. Dazu gehören

- Editoren
- Übersetzer (Compiler) oder Interpreter
- Debugger
- Profiler

6.1.3 Compiler

Unter einem **Übersetzer** versteht man ein Programm, das die Anweisungen eines in der Programmiersprache A, der *Quellsprache*, geschriebenen Programms in Anweisungen einer anderen Programmiersprache B, der *Zielsprache*, überträgt. Das dabei in der Zielsprache erzeugte Programm muss zum Ausgangsprogramm semantisch äquivalent sein. Von einem **Compiler** spricht man, wenn die Quellsprache A eine höhere Programmiersprache als die Sprache B ist.

Die Compilierung eines Quellprogramms erfolgt im Wesentlichen in vier Schritten:

1. **Lexikalische Analyse**: Das Quellprogramm $a \in A$ wird mit Hilfe eines *Scanners* in eine Folge von Token (grundlegende Symbole der Quellsprache) umgewandelt, d.h. in einzelne Sinneinheiten zerlegt.

   Dabei werden die verschiedenen Objekte der Sprache (Kommentare, Operatoren, Schlüsselworte, Bezeichner) als solche identifiziert.

2. **Syntaktische Analyse**: Im zweiten Schritt wird daraus mit einem *Parser* ein Ableitungsbaum entsprechend der Syntax der Quellsprache erzeugt, d.h. die Sinneinheiten entsprechend ihrer Bedeutung und gegenseitigen Beziehung gruppiert.

   In diesem Schritt werden auch syntaktische Fehler erkannt.

3. **Semantische Analyse**: Im dritten Schritt wird der Ableitungsbaum von $a \in A$ analysiert und in das Programm $b \in B$ übertragen.

   Dabei werden sematische Inhalte überprüft, etwa ob alle verwendeten Variablen auch deklariert und typgerecht verwendet wurden. Es können jedoch nur solche Fehler entdeckt werden, die sich bereits bei der Übersetzung bemerkbar machen. Laufzeitfehler, etwa Division durch 0, können an dieser Stelle nicht entdeckt werden.

   Dabei wird eine **Symboltabelle** angelegt, in der jeder Bezeichner genau einmal eingetragen wird. Unter diesem Eintrag werden alle Eigenschaften dieses Bezeichners, die durch verschiedene Deklarationen vereinbart werden, gespeichert.

4. **Code-Generierung** und **Code-Optimierung**: Im letzten Schritt wird der Code in der Sprache B zu einem Programm zusammengesetzt und evtl. weiter optimiert (Umbau von Schleifen und Funktionsaufrufen, Auflösung von Endrekursionen usw.).

Das Ergebnis der Übersetzung eines solchen Programmtexts ist, insbesondere bei C, noch kein lauffähiges Programm, sondern ein **Objektcode**, der erst durch den **Binder** (Linker) in ein lauffähiges Computerprogramm übertragen wird.

Der Grund dafür ist, dass in einem Programm so gut wie nie alle dort aufgeführten Funktionen auch als Code enthalten sind, sondern Implementierungen von Funktionen auch in anderen Moduln oder Bibliotheken enthalten sein können, die bereits (oder nur) als Objektcode vorliegen. Bei der Übersetzung bleiben dann die Adressen dieser externen Funktionen zunächst offen. Die Aufgabe des Binders ist es, die Objektcodes aller benötigten Module und Bibliotheken zu einem lauffähigen

Programm zusammenzufügen und die fehlenden Adressen externer Funktionen entsprechend zu ergänzen.

Bei einem Standardaufruf des C-Compilers

```
gcc myprog.c -o myprog
```

werden alle diese Schritte automatisch ausgeführt und am Ende liegt ein Programm `myprog` in Maschinenform vor.

Dieses kann nun von einem **Lader** in den Hauptspeicher gebracht werden, wobei vor allem geeigneter Platz im Hauptspeicher reserviert werden muss und die Adressen aller Variablen und Funktionen (diese Adressen sind im Maschinencode nur relativ zu Basisadressen fixiert) entsprechend anzupassen sind.

Das geladene Programm kann mit unterschiedlichen Datensätzen abgearbeitet werden, wobei sich ebenfalls noch Fehler, auch datensatzabhängig, bemerkbar machen können.

6.1.4 Interpreter versus Compiler

Neben Compilern spielen für einzelne Programmiersprachen auch Interpreter noch eine wichtige Rolle. Beides sind Werkzeuge, die dazu dienen, die in einer Hochsprache fixierte Implementierung eines Programms in ein Maschinenprogramm zu übertragen und auf dem Rechner auszuführen.

Beide Arten durchlaufen dabei die Phasen *lexikalische Analyse, syntaktische Analyse* und *semantische Analyse* des Programms.

Ein **Interpreter** führt die Analyse für *jeden einzelnen Befehl* extra aus und bringt ihn nach dieser Prüfung sofort zur Ausführung.

Ein **Compiler** führt in einer ersten Phase, der **Übersetzungszeit**, die Analyse des *vollständigen Programms* aus, übersetzt dieses in eine maschinennahe Sprache und speichert es ab. Erst in einer zweiten Phase, der **Laufzeit**, wird das übersetzte Programm zur Abarbeitung in den Hauptspeicher geladen.

Ein Compiler betreibt also einen größeren Aufwand bei der Analyse des Programms, der sich in der Abarbeitungsphase rentieren soll. Das ist nur sinnvoll, wenn dasselbe Programm mit mehreren Datensätzen ausgeführt wird. Das zentrale Paradigma des Compilers ist also das des *Programmflusses*, längs dessen Daten geschleust werden.

> Compiler werden deshalb vorwiegend dann eingesetzt, wenn ein und dasselbe Programm mit einer Vielzahl unterschiedlicher Datensätze abgearbeitet werden soll und die (einmaligen) Übersetzungszeitnachteile durch die (mehrmaligen) Laufzeitvorteile aufgewogen werden.
>
> Compiler sind typische Arbeitsmittel einer stapelorientierten Betriebsweise.

Ein **Interpreter** dagegen zeichnet sich durch eine höhere Flexibilität aus, da auch noch während des Ablaufs in das Geschehen eingegriffen werden kann. Werte von (globalen) Variablen sind während der Programmausführung abfrag- und änderbar und einzelne Anweisungen oder Deklarationen im Quellprogramm können geändert werden. Ein Interpreter ist also dort von Vorteil, wo man verschiedene Daten auf unterschiedliche Weise kombinieren und modifizieren möchte. Das zentrale Paradigma des Interpreters ist also das der *Datenlandschaft*, die durch Anwendung verschiedener Konstruktionselemente erstellt und modifiziert wird.

Als entscheidender Nachteil schlagen längere Rechenzeiten für einzelne Operationen zu Buche, da z.B. die Adressen der verwendeten Variablen mit Hilfe der Bezeichner ständig neu gesucht werden müssen. Ebenso ist Code-Optimierung nur beschränkt möglich.

Interpreter werden deshalb vor allem in Anwendungen eingesetzt, wo diese Nachteile zugunsten einer größeren Flexibilität des Programms, der Möglichkeit einer interaktiven Ablaufsteuerung und der Inspektion von Zwischenergebnissen in Kauf genommen werden.

Interpreter sind typische Arbeitsmittel einer dialogorientierten Betriebsweise.

Die meisten Interpreter stellen aber heute Mittel zur Verfügung, um einzelne Teile (Funktionen, Module) auch zu compilieren, wenn für diese das erste Paradigma zutrifft.

6.2 Methoden der Syntaxbeschreibung. Formale Sprachen.

Die Syntax einer Progammiersprache ist, wie bereits ausgeführt, streng formalisiert und in einer (endlichen) Anzahl von Regeln niedergelegt. Die natürliche Sprache ist nicht gut geeignet, diese Regeln kurz, übersichtlich, unmissverständlich und vor allem für eine maschinelle Verarbeitung geeignet zu beschreiben.

Man bedient sich deshalb einer formalisierten **Metasprache**, in der man Aussagen über andere Sprachen machen kann. Der Vorteil ist, dass man in einer solchen Metasprache meist mit wenigen Sprachkonstrukten auskommt, deren Beschreibung leicht in natürlicher Sprache erfolgen kann.

6.2.1 Bezeichnungen

Ein Programm ist ein endlicher Text, der aus einzelnen Buchstaben eines Alphabets Σ besteht.

Bezeichnung Σ^* für Worthalbgruppe über Σ, ε als leeres Wort, $\Sigma^+ = \Sigma^* - \{\varepsilon\}$, $|w|$ für Wortlänge von $w \in \Sigma^*$. Notation vw für $v, w \in \Sigma^*$.

Eine Sprache legt fest, welche dieser Texte "sinnvoll" sind. Eine Sprache L ist also eine Teilmenge $L \subset \Sigma^*$.

6.2.2 Ein Beispiel

Folie Ableitungsbaum, gelesen von unten nach oben. Vorher lexikalisch auseinandernehmen.

Folie Grammatik.

Anschreibung als BNF und EBNF.

6.2.3 Formale Sprachen und Grammatiken

Eine formale Sprache $L = L(G)$ kann durch eine **Grammatik** G beschrieben werden.

Dazu ist zunächst ein **Vokabular** $V = T \cup N$ festzulegen, bestehend aus einer endlichen Menge T von **Terminalsymbolen** (dies sind die kleinsten Einheiten, die in Sätzen der Sprache L wirklich auftreten können) und einer endlichen Menge von **Nichtterminalsymbolen** (NTS) N (dies sind Bezeichner für größere Sinneinheiten in einem Text aus L).

In der Menge N wird ein **Satzsymbol** S (oder Startsymbol) ausgezeichnet (dieses steht für die größte Sinneinheit, also bei Programmiersprachen für `<Programm>`).

Schließlich ist noch eine endliche Menge von **Produktionsregeln** $P = \{a_i \mapsto b_i\}$ anzugeben mit $a_i, b_i \in V^*$, die beschreiben, wie die Bedeutung von Nichtterminalsymbolen Schritt für Schritt aufgelöst werden kann.

Als **Produktion** bezeichnet man die Relation auf V^2, die aus allen Paaren (u, v) (Notation $u \Rightarrow v$) besteht mit folgender Eigenschaft:

$$\exists\, x, y \in V^*,\ \ (a_i \mapsto b_i) \in P : u = x a_i y,\ v = x b_i y$$

$u \Rightarrow^* v$ ist die transitive Hülle dieser Relation. Eine Buchstabenkette $u \in V^*$ heißt **Satzform**, wenn sie durch endlich viele Produktionen aus dem Satzsymbol hergeleitet werden kann, d.h. wenn $S_0 \Rightarrow^* u$ gilt. Als (korrekt geformten) **Satz** bezeichnet man eine Satzform aus T^*.

Die durch $G = (N, T, P, S)$ erzeugte Sprache $L(G) \subset T^*$ besteht dann aus genau den korrekt geformten Sätzen, d.h. aus den $w \in T^*$, die sich durch endlich viele Produktionen aus dem Satzsymbol S_0 herleiten lassen:

$$L(G) = \{u \in T^* \ :\ S \Rightarrow^* u\}$$

Beispiel: Hund-Katze

Folie Grammatik. Folie "Eine Ableitung". Folie Syntaxdiagramm

Beispiel: Arithmetische Ausdrücke

$G = (N = \{\texttt{<E>}, \texttt{<T>}, \texttt{<F>}\}, T = \{+, -, *, /, v, c, (,)\}, P, S = \texttt{<E>})$

(mnemonische Bedeutung: E = expression, T = term, F = factor, v = variable, c = constant)

Produktionsregeln:

$$\begin{array}{lcl}
\texttt{<E>} & \mapsto & \texttt{<T>} | + \texttt{<T>} | - \texttt{<T>} | \texttt{<E>} + \texttt{<T>} | \texttt{<E>} - \texttt{<T>} \\
\texttt{<T>} & \mapsto & \texttt{<F>} | \texttt{<T>} * \texttt{<F>} | \texttt{<T>} / \texttt{<F>} \\
\texttt{<F>} & \mapsto & v | c | (\texttt{<E>})
\end{array}$$

Darstellung als Syntaxdiagramm

Beispiel: Bezeichner

In C können Objekte durch vom Programmierer wählbare Namen, also **Bezeichner**, angesprochen werden. Ein Name muss dabei eine Zeichenkette aus Buchstaben, dem Unterstrich _ und Dezimalziffern sein, wobei als erstes Zeichen nur ein Buchstabe oder ein Unterstrich zugelassen ist.

$G = (N = \{\texttt{<B>}, \texttt{<D>}, \texttt{<Z>}, \texttt{<N>}\}, T = \{0..9, A..Z, a..z, \_\}, P, S = \texttt{<N>})$

(mnemonische Bedeutung: B = Buchstabe, D = Digit, Z = Zeichenkette, N = Name)

Produktionsregeln:

$$\begin{array}{lcl}
\texttt{<N>} & \mapsto & \texttt{<B>} | \texttt{<B><Z>} \\
\texttt{<Z>} & \mapsto & \texttt{<B>} | \texttt{<B><Z>} | \texttt{<D>} | \texttt{<D><Z>} \\
\texttt{<B>} & \mapsto & a|b|..|z|A|..|Z|_ \\
\texttt{<D>} & \mapsto & 0|1|..|9
\end{array}$$

Darstellung als Syntaxdiagramm. Ableitung alternativer Darstellung:

Produktionsregeln:

$$\begin{array}{lcl}
\texttt{<N>} & \mapsto & \texttt{<B><Z'>} \\
\texttt{<Z'>} & \mapsto & \varepsilon | \texttt{<B><Z'>} | \texttt{<D><Z'>} \\
\texttt{<B>} & \mapsto & \ldots \\
\texttt{<D>} & \mapsto & \ldots
\end{array}$$

Darstellung in erweiterter BNF spart ein NTS ein:

$$\begin{array}{lcl}
\texttt{<N>} & \mapsto & \texttt{<B>}[\{(\texttt{<B>}|\texttt{<D>})\}] \\
\texttt{<B>} & \mapsto & \ldots \\
\texttt{<D>} & \mapsto & \ldots
\end{array}$$

6.2.4 Syntaxdiagramme und BNF-Notation

Formalisierung der bisherigen Notation

Backus-Naur-Form (BNF)

Verwendet folgende **Metasymbole** (von denen angenommen wird, dass sie nicht in V enthalten sind: `::=`, |

Für jedes NTS gibt es genau eine (zusammenfassende) Produktionsregel. $A_1, A_2, ..., A_n$ bezeichnet Worte aus V^*.

Sprachelement	**Bedeutung**
<A>	Nichtterminales Symbol
<A>::=...	Produktionsregel
$A_1 A_2 ... A_n$	Sequenz
$A_1 \vert A_2 \vert ... \vert A_n$	Alternative

Syntaxdiagramme

siehe [Ernst], S. 247

6.2.5 Die Chomsky-Hierarchie

Für praktische Zwecke dürfen die Produktionsregeln nicht "zu wild" sein. Es gibt einen engen Zusammenhang zwischen verschiedenen Sprachtypen und verschiedenen Typen von Automaten, die solche Sprachen analysieren können. Sprachanalyse kann in dem Sinne also "unterschiedlich schwierig" sein.

Als **Chomsky-0-Grammatik** bezeichnet man eine Grammatik, in der jede Produktionsregel mindestens ein NTS auf der linken Seite enthält. Auf Worte aus T^* können dann keine Produktionen mehr angewendet werden, sie sind also *terminal*. Solche Sprachen bezeichnet man auch als **rekursiv aufzählbare Sprachen**, weil man einen Algorithmus angeben kann, der alle gültigen Worte nacheinander erzeugt.

Produktionen: $xsy \mapsto u$ mit $x, y, u \in V^*$, $s \in N$

Als **Chomsky-1-Grammatik** oder auch **kontextabhängige Grammatik** bezeichnet man eine Chomsky-0-Grammatik mit Produktionsregeln der Art

$u \mapsto v$ mit $u, v \in V^*$, $|u| \leq |v|$

Meist haben die Regeln die Form $xsy \mapsto xuy$. $x..y$ ist also der Kontext, in dem die Ersetzung $s \mapsto u$ vorgenommen werden darf. Solche Sprachen können durch linear beschränkte Automaten analysiert werden.

Als **Chomsky-2-Grammatik** oder **kontextfreie Grammatik** bezeichnet man eine Grammatik, in deren Produktionsregeln links nur einfache NTS stehen. Die Regeln haben also die einfache Form

$A \mapsto u$ mit $A \in N, u \in V^*$

Solche Sprachen spielen bei der Definition der Syntax von Programmiersprachen eine zentrale Rolle. Sie können durch Kellerautomaten analysiert werden.

Als **Chomsky-3-Grammatik** oder **(rechts)reguläre Grammatik** bezeichnet man eine kontextfreie Grammatik, wo auch auf der rechten Seite nur ein NTS vorkommt, und zwar in der Form

$A \mapsto uB$ mit $A, B \in N, u \in T^*$

Solche Sprachen spielen bei der Definition der Lexik von Programmiersprachen eine zentrale Rolle. Sie sind besonders einfach und können durch endliche Automaten analysiert werden.

6.2.6 Reguläre Sprachen und reguläre Ausdrücke

Beispiel Bezeichner: Zweite Variante kann in eine reguläre Grammatik aufgelöst werden. Durch Einsetzen können aus dem Syntaxdiagramm alle NTS entfernt werden. Aufschreiben des zugehörigen Syntaxdiagramms als **regulärer Ausdruck**:

Metasymbole sind $*$, $|$, runde Klammern, eckige Klammern

Reguläre Ausdrücke sind (wenn $A_1, A_2, ..., A_n$ für bereits definierte reguläre Ausdrücke steht):

Worte aus T^*	Diese stehen für sich selbst
$A_1 A_2 ... A_n$	Sequenz
$A_1 \| A_2 \| ... \| A_n$	Alternative
$A*$ oder $(A)*$	Wiederholung (0,1,... Mal)

Runde Klammern werden gesetzt, um die Ausdrücke eindeutig zu machen.

Weitere Abkürzungen: [...] für Alternativen von einzelnen Zeichen. $A+$ für $AA*$.

Beispiele:

`[a-zA-Z_][a-zA-Z_0-9]*`	Bezeichner
`(+|-|)[0-9]+`	ganze Zahl
`1*0(1*01*0)*1*`	Zahlen aus 0 und 1 mit ungerader Anzahl von Nullen

6.2.7 Beispiel: Die Syntax der WHILE-Programme

Analyse eines konkreten Programms (**ADD(a,b)**)

```
while x1>0 do x1:=dec(x1); x0:=inc(x0) end;
y0:=x0
```

1) Lexikalische Analyse:

Finde alle **Token**, also Schlüsselwörter (`while`, `>0`, `do`, `inc`, `dec`), Operator-, Trenn-, Kommentarsymbole (`( ) ; /* */`) sowie die Variablen und Konstanten (Literale).

```
<Variable>   ::=  [xyz][0-9]*
<Konstante>  ::=  [0-9]+
```

2) Syntaktische Analyse:

Untersuche, ob die Token in einer sinnvollen Weise angeordnet sind.

```
<Programm>   ::=  <Zuweisung>
                  |<Programm> ; <Programm>
                  |while <Test> do <Programm> end
<Zuweisung>  ::=  <Variable>:=<Variable>
                  |<Variable>:=inc(<Variable>)
                  |<Variable>:=dec(<Variable>)
                  |<Variable>:=<Konstante>
<Test>       ::=  <Variable> > 0
```

Notationsvereinbarung (erweiterte BNF):

- Terminalsymbole (z.B. die Schlüsselworte) in `Schreibmaschinenschrift`
- Als Nichtterminalsymbole werden Wörter verwendet, die Bedeutung suggerieren

- Nichtterminalsymbole nicht mehr in spitzen Klammern, sondern kursiv
- Statt <A>::= steht *A:* auf einer Zeile, rechte Seite der Produktionsregel darunter eingerückt
- Optionale Bestandteile, also $(\varepsilon|A)$, durch [*A*] gekennzeichnet.
- Sich wiederholende Bestandteile, also $(A|AA|AAA|...)$, durch { *A* } gekennzeichnet. (Braucht sonst ein eigenes NTS *A-list:* (*A* |*A-list A*))
- Optional mehrfache Bestandteile also [{ *A* }]
- Absatz trennt Regeln voneinander.

Programm:

    Zuweisung
    |*Programm* `;` *Programm*
    |`while` *Test* `do` *Programm* `end`

Zuweisung:

    Variable `:=` *Variable*
    |*Variable* `:=` `inc(` *Variable* `)`
    |*Variable* `:=` `dec(` *Variable* `)`
    |*Variable* `:=` *Konstante*

Test:

    Variable `> 0`

Variable: `[xyz][0-9]*`

Konstante: `[0-9]+`

6.3 Der grundlegende Aufbau von C-Programmen

6.3.1 Deklarationen und Definitionen

Ein C-Programm besteht aus mehreren **Übersetzungseinheiten**, die in verschiedenen Dateien (source files) gespeichert sind.

Jede solche Übersetzungseinheit enthält eine Folge von Präprozessordirektiven, Deklarationen und Funktionsdefinitionen.

Beispiel:

```
#include <stdlib.h>

long gcd(long a,long b) {
  long r;
  while(b) {r=a%b; a=b; b=r;}
  return a;
}

int main() {
  long a,b;

  a=333; b=481;
  printf("gcd(%i,%i)=%i\n",a,b,gcd(a,b));
  a=4859; b=9589;
  printf("gcd(%i,%i)=%i\n",a,b,gcd(a,b));
  a=10001*a; b=10001*b;
  printf("gcd(%i,%i)=%i\n",a,b,gcd(a,b));
```

```
  a=485901; b=958902;
  printf("gcd(%i,%i)=%i\n",a,b,gcd(a,b));
}
```

Deklaration (oder Vereinbarung)	Bezeichner wird mit Bedeutung verbunden (Festlegung von Eigenschaften)
Definition	Bezeichner wird mit Bedeutung und Inhalt verbunden

Beispiele:

Variablendeklaration	`int x;`
Variablendefinition	`int x=2;`
Funktionsdeklaration	`extern float sqrt(float x);`
Funktionsdefinition	`float sqrt(float x) { ... }`
Typdefinition	`typedef struct { int z,n; } bruch;`

Vom Compiler wird jede Übersetzungseinheit separat in Objektcode übertragen, der vom Linker zu einem ausführbaren Programm zusammengebunden wird. Dieses Maschinenprogramm enthält nur noch die Laufzeitinformationen und besteht aus einer Sammlung von Funktionsdefinitionen in Maschinencode. Im Gegensatz etwa zu Pascal gibt es kein (syntaktisch herausgehobenes) Hauptprogramm. Statt dessen muss es in dieser Sammlung von Funktionsdefinitionen genau eine mit dem Bezeichner `main` geben, der beim Abarbeiten des Programms die Steuerung vom Betriebssystem übergeben wird.

In jeder Übersetzungseinheit müssen alle Bezeichner vor ihrer ersten Verwendung **deklariert** sein. Bezeichner (insbesondere Funktionsbezeichner), die in mehreren solchen Einheiten verwendet werden, müssen in **jeder** von ihnen deklariert sein. Dazu verwendet man gemeinsame **Header-Dateien**, die alle relevanten Deklarationen enthalten und vom Präprozessor in den Quellcode der jeweiligen Übersetzungseinheit eingefügt werden. Dieser Mechanismus wird auch vom C-Compiler selbst verwendet.

Unterschied

```
#include <stdio.h>
#include "zahlen.h"
```

Variablen- und Funktionsbezeichner haben auch über die Compilezeit hinaus Bedeutung, wozu für sie entsprechender Speicherplatz zu reservieren ist. Benannte Speicherbereiche nennt man auch **Objekte**, die Zuordnung von Speicherplatz zu einem solchen Objekt heißt **Initialisierung**. Jedes Objekt darf innerhalb der verschiedenen Übersetzungseinheiten, die zu einem Programm gehören, nur einmal initialisiert werden. Diese Initialisierung wird normalerweise bei der Deklaration des entsprechenden Bezeichners vorgenommen.

Da Headerdateien gewöhnlich in mehrere Quelldateien desselben Programms eingebunden sind, wird auch die Deklaration mehrfach ausgeführt. Mit dem Schlüsselwort `extern` kann die Initialisierung auf einen späteren Zeitpunkt verschoben und nur die Compileinformation vereinbart werden. Deshalb sind Variablen- und Funktionsdeklarationen in Header-Dateien gewöhnlich als `extern` gekennzeichnet.

Typischer Aufbau eine Header-Datei (am Beispiel `zahlen.h`)

```
#ifndef _zahlen_h
#define _zahlen_h
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct { int z,n; } bruch ;

extern bruch b_int(int n);
extern bruch b_plus(bruch a, bruch b);
extern bruch b_minus(bruch a, bruch b);
extern bruch b_mal(bruch a, bruch b);
extern int b_iszero(bruch a);
extern bruch b_inv(bruch a);
extern bruch b_durch(bruch a, bruch b);
extern bruch b_hoch(bruch a, int b);
extern void b_print(bruch a);

extern bruch b_scan();

#endif
```

Die Initialisierung erfolgt dann in der Übersetzungseinheit, die die zugehörigen Definitionen enthält. Die Zuordnung des dabei reservierten Speicherplatzes zum Objektcode anderer Übersetzungseinheiten, die auf denselben Bezeichner Bezug nehmen, erfolgt durch den Linker.

Das hat den Vorteil, dass es ausreicht, von Funktionen die Deklaration (den Prototyp) offenzulegen, die Implementierung kann jedoch als Objektcode weitergegeben werden.

Der C-Compiler verwendet denselben Mechanismus für die "eingebauten Funktionen", die auf verschiedene Bibliotheken (und die Prototypen auf verschiedene Header-Dateien) verteilt sind. Damit kann spezieller Code, der nur für gewisse Aufgabenstellungen benötigt wird (z.B. Grafikroutinen), in den Bibliotheken zur Verfügung stehen, aber selektiv in Programme eingebunden werden. Der Linker ist überdies so clever, dass er nur diejenigen Funktionsdefinitionen aus dem Objektcode in den Maschinencode übernimmt, die wirklich benötigt werden.

6.3.2 Lexikalische Analyse

Vor der lexikalischen Analyse werden zunächst alle Präprozessordirektiven ausgeführt (insbesondere die Header-Dateien eingefügt und alle Makros expandiert). Danach wird der Text in eine Folge von Eingabesymbolen (Token) zerlegt.

Leerzeichen, Tabulatoren, Zeilentrenner usw. (white spaces) sowie Kommentare werden bei dieser Analyse ignoriert (außer dass sie Eingabesymbole voneinander trennen). Als nächstes Eingabesymbol wird immer die längstmögliche sinnvolle Zeichenkette genommen.

In C gibt es 6 Klassen von Eingabesymbolen:

Symbolart	**englisch**	**Syntax**
Bezeichner	identifier	`[a-zA-Z_][a-zA-Z0-9_]*`
Schlüsselworte	keywords	siehe unten
Konstanten	literals	*int-c* | *char-c* | *float-c* | *enum-c* | *string-c*
Operatoren	operators	`+ - * / = == *=` usw.
Trennzeichen	punctuators	`, ;` usw.

Schlüsselworte von C:

```
auto break case char const continue default do double else enum extern float for
goto if int long register return short signed sizeof static struct switch
typedef union unsigned void volatile while
```

Konstanten (oder **Literale**):

NTS	Bezeichnung	Beispiel
int-c	ganzzahlige Konstante	`-31827`
float-c	Gleitkommakonstante	`1.33e3`
char-c	Zeichenkonstante	`'A'`
string-c	konstante Zeichenkette	`"abc"`
enum-c	Aufzählungskonstante	—

Als Zeichen sind dabei neben den ASCII-Zeichen auch entsprechende **Escape-Sequenzen** (`\n` `\t` `\b` usw.) zugelassen, die verschiedene Metazeichen maskieren.

6.4 Variablen, Referenzen, Zuweisungen

Allgemein gesprochen ist eine Variable wie jedes Objekt als benannter Speicherplatz ein Paar (Referenz, Wert).

Die **Referenz** ist ein Verweis auf den Ort, wo der **Wert** steht. **L-Wert** (lvalue) und **R-Wert** (rvalue). Referenz einer Variablen ist fest, der Wert kann sich im Laufe des Programms ändern. Bild von Box mit Aufschrift und Inhalt.

Beispiel: `int x=12; ... x=13;` Bild dazu.

Der **Variablenname** ist ein symbolischer Bezeichner für die Referenz, über den auf den Inhalt (Wert) der Variablen zugegriffen wird. Die Zuordnung Bezeichner - Speicherplatz übernimmt der Compiler.

Bestandteile einer Variablen:

zur Compilezeit	**zur Laufzeit**	Bemerkung
Bezeichner	–	symbolische Bezeichnung für die Referenz
Referenz	Speicheradresse	fixiert (L-Wert)
Wert	Speicherinhalt	veränderbar (R-Wert)
Datentyp	Speichergröße	fixiert

Variablen, deren Wert nur einmal während der Initialisierung gesetzt werden kann, bezeichnet man als **Konstantenvariable** (manchmal auch nur kurz als Konstanten). Schlüsselwort `const`.

Beispiel: `const double E=2.7182818284590452353602 8;`

Für Konstantenvariable muss der Compiler nicht unbedingt Speicherplatz reservieren, sondern kann bereits zur Compilezeit jedes Vorkommen durch den (ja bekannten) Wert ersetzen. Das ist aber compilerabhängig.

Der Wert einer Variablen wird durch eine **Zuweisung** gesetzt oder verändert. `x=y;` bedeutet folgendes: Packe eine Kopie des Inhalt der Büchse mit Aufschrift `y` in die Büchse mit der Aufschrift `x`. Etwas formaler: Finde den Wert von `y` und speichere eine Kopie davon an dem von der Referenz von `x` bezeichneten Speicherplatz. Von `x` ist in diesem Fall der L-Wert, von `y` der R-Wert interessant.

Variablen müssen vor ihrer ersten Verwendung **deklariert** werden. Der Compiler erkennt am Datentyp, wieviel Speicherplatz zu reservieren und wie ein dort gespeichertes Bitmuster zu interpretieren ist.

Daten, deren Wert unmittelbar in einer Variablen an der angegebenen Stelle gespeichert ist, heißen auch **statische Daten**. Die Größe des benötigten Speicherplatzes muss dazu zur Compilezeit bekannt sein.

Es kann auch Datenstrukturen geben, deren (genauer) Speicherbedarf erst zur Laufzeit bekannt ist. Beispiele: Listen, Bäume, Strings. Solche Daten können mit dem angegebenen Mechanismus nicht duch Variablen dargestellt werden. Statt dessen kann aber eine Variable angelegt werden, die als Wert **die Referenz** auf diese erst zur Laufzeit zu erzeugende Datenstruktur aufnimmt. Derartige Daten heißen **dynamische Daten**. Der für die Daten selbst erforderliche Speicherplatz kann nicht vom Compiler reserviert werden. Er muss vom Programm zur Laufzeit explizit abgefordert werden.

Solche Variablen, deren Wert also wieder eine Referenz ist, heißen **Zeigervariablen**. Der spezielle Wert `NULL` (definiert in `<stdio.h>`) symbolisiert die leere Referenz, die auf keine gültige Speicherstelle verweist. Sie wird intern durch das Bitmuster `0..0` dargestellt. Referenzen werden in C u.a. für die Darstellung von Strings und Feldern verwendet.

Beispiel: `char *s="Das ist ein String";` Bild dazu.

Referenzen können beliebig tief geschachtelt sein.

Beispiel: `int m[][]={{1,1,1},{1,2,3}}`. Bild dazu.

In komplexeren objektorientierten Sprachen (etwa Java) sind fast alle Variablen (genauer: alle Klassenvariablen) dynamisch angelegt. Speicherplatzanforderung erfolgt dort zur Laufzeit während der Initialisierung automatisch. Dazu muss aber ein Teil des Typsystems in die Laufzeitumgebung übertragen werden.

6.5 Datentypen und Datenstrukturen

Eigenschaften von Variablen sind in ihrem **Datentyp** fixiert.

Ein **Datentyp** `T` (oder **abstrakter Datentyp**) besteht aus folgenden Informationen:

Grundbereich	Wertebereich, aus dem Daten als Bitmuster in Variablen entsprechenden Typs dargestellt werden können. **Bezeichnung**: $M(T)$
Signatur	Operationen und Funktionen mit genauer Angabe von Stelligkeit, Definitionsbereich und Wertevorrat, die für Daten dieses Typs erklärt sind.

Als **Datenstruktur** (oder **konkreten Datentyp**) bezeichnen wir einen abstrakten Datentyp, für den zusätzlich Informationen über dessen Syntax und Implementierung gegeben sind:

Realisierung	insbesondere Zurückführung auf andere Datentypen durch geeignete **Konstruktoren**
Literale	Konstanten dieses Datentyps, und die Art, wie diese, etwa bei der Initialisierung, anzuschreiben sind.

Eine Variable eines gegebenen (abstrakten oder konkreten) Datentyps `T` bezeichnet man auch als **Instanz** dieses Datentyps, den Inhalt einer solchen Variablen (also ein Element aus dem

Grundbereich) als **Datensatz** vom Typ `T`.

Beispiel:

Name	**Integertypen**
Grundbereich	$\{x \in \mathbf{N} : -2^{q-1} \leq x \leq 2^{q-1} - 1\}$
Signatur	Arithmetikoperatoren, Vergleichsoperatoren, ...
Realisierung in C	`int`, `short`, `long`, `char`, jeweils kombinierbar mit `signed` und `unsigned`
Literale	`(|+|-)[0-9]+`

Beispiel Signaturen:

$+$:	`int` $\times$ `int` $\rightarrow$ `int`	Addition zweier ganzer Zahlen
$-$:	`int` $\times$ `int` $\rightarrow$ `int`	Subtraktion zweier ganzer Zahlen
$-$:	`int` $\rightarrow$ `int`	Vorzeichenwechsel
$==$:	`int` $\times$ `int` $\rightarrow$ `boolean`	Vergleich zweier ganzer Zahlen

In einem Grundbereich können spezielle Elemente ausgezeichnet sein, etwa $0, 1 \in \mathbf{N}$. Solche speziellen Konstanten werden als nullstellige Funktionen in die Signatur aufgenommen:

`null` :	$()$ $\rightarrow$ `int`	Aufruf `null()` liefert 0
`eins` :	$()$ $\rightarrow$ `int`	Aufruf `eins()` liefert 1

Realisierung in C etwa `int null() { return 0; }`

6.5.1 Elementare Datentypen

Bereits behandelt: Integertypen, Realtypen, Zeichen (`char`)

Weitere elementare Datentypen:

Name	**Boolean**
Grundbereich	$\{true, false\}$
Signatur	logische Operatoren
Realisierung in C	Wird abgebildet auf `int`, Bitmuster 0..0 (alles Null) werden als *false* interpretiert, alle anderen Bitmuster als *true*

Beispiel: Summe der ersten n natürlichen Zahlen bestimmen.

```
int sum(int n) { int u=0; while(n) u+=n--; return u; }
```

Name	**Leerer Typ**
Grundbereich	leere Menge
Signatur	—
Realisierung in C	Als elementarer Typ `void`

Beispiel: Deklaration einer Funktion zur Ausgabe von Strings

```
void print(char *s)
```

Da es keine Objekte vom Typ `void` gibt, können auch keine Variablen dieses Typs vereinbart werden. `void` wird verwendet als Resultattyp für Funktionen, die keine Werte erzeugen, sowie als `void*` für generische Zeigervariablen.

Name	**Aufzählungstyp**
Grundbereich	selbstdefinierte endliche Menge
Signatur	Vergleichsoperatoren
Realisierung in C	`enum` [*enum-name*] { *enum-list* }

Wird auf geeigneten Integertyp abgebildet (ohne Bereichsprüfung), beginnend mit 0 (Standard). Literale eines solchen Typs sind selbstgewählte Bezeichner. Normalerweise verwendet man dafür Bezeichner aus `[A-Z]+`.

Beispiel:

```
enum boolean {NO, YES};

enum months {JAN=1, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC};
```

Name	**Unterbereichstyp**
Grundbereich	endliche Teilmenge eines Integertyps
Signatur	wie die des Grundtyps
Realisierung in C	nicht unterstützt

Unterbereichstypen werden verwendet, wenn Variablen eines Integertyps nur Werte aus einem bestimmten Bereich annehmen dürfen.

Beispiel (Pascal):

```
type jahr = 1800 .. 2000;
...
var geburtsjahr:jahr;
```

Unterbereichstypen haben als Grundbereich meist ein zusammenhängendes Intervall des Grundtyps. Die Signatur stimmt mit der des Grundtyps überein mit der Einschränkung, dass Argumente und Ergebniswerte immer im Unterbereich liegen müssen.

Der Compiler kann Bereichsüberschreitungen von Literalen zur Übersetzungszeit bemerken sowie für Zuweisungen an Variablen vom Unterbereichstyp eine entsprechende Bereichsüberprüfung in das Maschinenprogramm integrieren. In C müssen diese Effekte ggf. softwareseitig realisiert werden.

Unterbereichstypen und Aufzählungstypen gehören zu den **endlichen Typen**, denn ihre Grundbereiche sind endliche (geordnete wg. Grundtyp!) Mengen.

6.5.2 Feld und Verbund

Feld und Verbund sind **Konstruktoren**, mit denen aus einfacheren Datentypen zusammengesetzte Datenstrukturen aufgebaut werden können.

Zusammengesetzte Datenstrukturen dienen dazu, mehrere Daten in einer Einheit zusammenzufassen. Unterscheide heterogene Einheiten (**Verbund**) und homogene Einheiten (**Feld**).

Zusammengesetzte Datentypen können mit eigenen Namen belegt werden:

`typedef` *Typ-Definition Typ-Bezeichner* `;`

Verbund: Gegeben Datentypen $\mathtt{T}_1$,...,$\mathtt{T}_n$. Als Verbund `T` bezeichnet man einen Datentyp mit dem

> Grundbereich $M(T_1) \times ... \times M(T_n)$ (kartesisches Produkt) und
>
> Zugriffsfunktionen (Selektoren) $e_i : \mathtt{T} \rightarrow \mathtt{T}_i,\ i = 1, ..., n$

Realisierung in C:

Typdefinition :	`typedef struct {` $\mathtt{T}_1$ *name*$_1$`;` ...`;` $\mathtt{T}_n$ *name*$_n$`; } T;`
Variablendeklaration :	`T var;`
Selektor :	$\mathtt{var.name}_i$
Literale :	$\mathtt{T\ var=\{l_1,...,l_n\};}$

Dabei bezeichnen $\mathtt{l}_1$,...,$\mathtt{l}_n$ Literale der Datentypen $\mathtt{T}_1$,...,$\mathtt{T}_n$.

Beispiel:

```
typedef struct { int z; int n; } bruch ;
```

Feld: Gegeben Datentyp `T` (Wertebereich) und ein endlicher Datentyp `B` (Indexbereich). Als B-indiziertes Feld `F` über `T` bezeichnet man einen Datentyp mit dem

> Grundbereich $M(F) = M(T)^{M(B)}$ (kartesische Potenz, auch Interpretation als Menge der Funktionen $\mathtt{B} \rightarrow \mathtt{T}$) und
>
> einer Zugriffsfunktion e mit der Signatur $\mathtt{F} \times \mathtt{B} \rightarrow \mathtt{T}$, die aus einem Feld vom Typ `F` das Element mit dem entsprechenden Index selektiert.

In C wird als endlicher Datentyp `B` immer das Intervall $0..(n-1)$ genommen und durch n symbolisiert, in anderen Sprachen (etwa Pascal) sind spezielle Unterbereichstypen mit anderen Intervallgrenzen möglich.

Realisierung in C:

Variablendeklaration :	`T var[n];`
Selektor :	`var[i]`
Literale :	$\mathtt{T\ var[n]=\{l_1,...,l_n\};}$

Dabei bezeichnen $\mathtt{l}_1$,...,$\mathtt{l}_n$ Literale vom Typ `T`.

Die Zugriffsfunktion wird als Operator in Klammernotation angeschrieben. Gewöhnungsbedürftig ist die Variablendeklaration. Statt der logischen Notation `T[n] var` wird `T var[n]` verwendet, also das Feld über die Deklaration des Elementtyps deklariert.

Je nachdem, ob n zur Compilezeit oder erst zur Laufzeit bekannt ist, können Felder statisch oder dynamisch angelegt sein. Die folgenden beiden Definitionen vereinbaren Typen für Zeichen-Felder. Der Typ `dyn_chararray` ist dynamisch (Länge zur Compilezeit unbekannt), der Typ `stat_chararray` ist statisch (Feld mit 10 Elementen).

Typdefinitionen für dynamische Datentypen erfolgen in C stets als Typdefinition für zugeordnete Objekte:

```
falsch:
typedef char[] dyn_chararray;
typedef char[10] stat_chararray;

richtig:
typedef char dyn_chararray[];
typedef char stat_chararray[10];
```

Vom Compiler zu initialisierende Objekte sind immer statisch. Deshalb können Variablen vom Typ `stat_chararray`, nicht aber vom Typ `dyn_chararray` deklariert werden. Variablen vom Typ `dyn_chararray` können aber in Funktionsdefinitionen verwendet werden.

Beispiel:

```
int getinteger(dyn_chararray s) {
  int i,n;
  n=0;
  for(i=0; s[i]>='0' && s[i]<='9'; i++)
    n=10*n+(s[i]-'0');
  return n;
}

int main() {
  stat_chararray c;
  scanf("%s",&c);
  printf("%i\n",getinteger(c));
}
```

Die Feldvariable c (bzw. s) stellt eine Referenz auf die Basisadresse dar, ab welcher die Feldelemente hintereinander im Speicher liegen. Deshalb können Feldvariable in C wie Zeigervariable verwendet werden.

Auch bei statischen Feldvariablen findet keine Bereichsprüfung statt, ob die referenzierten Feldelemente noch innerhalb des bei der Initialisierung reservierten Speicherbereichs liegen.

Flexible Felder: Dynamische Felder lassen sich beschränkt auch statisch implementieren, wenn die Anwendung nicht speicherplatzkritisch ist.

Beispiel:

```
#define MAX 100
typedef struct {int F[MAX]; int len;} int_feld;

void print_feld(int_feld f) {
  int i;
  printf("[");
  for (i=0;i<f.len-1;i++) printf("%i, ",f.F[i]);
  printf("%i]\n",f.F[f.len-1]);
}

int main() {
  int_feld f={{1,2,3},3};
```

```
  print_feld(f);
}
```

Ein solcher Verbund aus statischem Feld maximaler Länge und Integer-Variable für die aktuelle Länge heißt *flexibles Feld.*

Strings als Zeichen-Felder: Strings werden in C als dynamische Zeichenfelder implementiert. Deshalb muss, entweder statisch oder dynamisch, für Strings ausreichend Speicherplatz allokiert werden. Das Stringende wird am Stringendezeichen `'\0'` erkannt. Strings sind also immer ein Zeichen länger als die Anzahl der (druckbaren) Zeichen angibt.

Beispiel:

```
void print(char s[]) {
  int i;
  for(i=0;s[i];i++) printf("%c",s[i]);
}

int main() {
  char buffer[20]="Ein Testbeispiel";
  print(buffer); printf("\n");
}
```

6.5.3 Klassifikation der Konstruktoren

Konstruktoren zum Aufbau zusammengesetzter Datenstrukturen unterteilt man gewöhnlich in die folgenden Klassen:

Aggregation: Feld oder Verbund

Generalisation
    Grundmenge: $M(T_1) \cup ... \cup M(T_n)$
    C-Realisierung: `union`

Signatur: Durchschnitt der Signaturen der Teilbereiche (d.h. diejenigen Funktionen, die auf allen Teilbereichen und damit der Vereinigung definiert sind)

Generalisationen bezeichnet man auch als **varianten Verbund**. Hier ist derselbe Speicherbereich unterschiedlich zu interpretieren. Realisation in C mit Fallvariable.

Meist wird die Generalisation aber in der umgekehrten Richtung eingesetzt als **Obertyp** `T` der Typen $\mathtt{T}_1, \ldots, \mathtt{T}_n$, die vom Obertyp dessen Signatur (und teilweise sogar die Implementierung derselben) **erben**. Objektorientierter Ansatz, der von C nur in der Erweiterung C++ unterstützt wird.

Sequenz
    Grundmenge: $M(T)*$
    C-Realisierung: (`FILE`)

Signatur: `put, get, next, eof`

Sequenzen werden in C nur als Bitstrom unterstützt. Syntax stärker an der Realisierung als Datei im Dateisystem orientiert als an der Struktur des Grundtyps `T`.

dynamische Datenstrukturen
    C-Realisierung: Zeiger

Funktionstypen
    Grundmenge: Fkt. $M(T_1) \rightarrow M(T_2)$
    C-Realisierung: nicht unterstützt

6.5.4 Funktionstypen

Wertebereich sind die Abbildungen einer Menge in eine andere. Realisierung relativ kompliziert, wird aber von einigen Sprachen (Scheme, Lisp) unterstützt. In anderen Sprachen stehen nur einzelne Funktionstypen explizit zur Verfügung.

Beispiele:

1) **Feld als Funktionstyp:** Die Menge $M(T)^{M(B)}$ kann verstanden werden als Menge der Funktionen $M(B) \to M(T)$.

2) **Potenzmengentyp:** konstruiert aus einem gegebenen endlichen Typ `T` einen Typ `Set of T`, der als Grundbereich die Menge $P(M)$ der Teilmengen von $M = M(T)$ hat. Die Signatur umfasst die geläufigen Mengenoperationen. Die Potenzmenge $P(M)$ kann man auch als Menge der Funktionen `T` $\to$ `boolean` (Schreibweise auch 2^M) verstehen, die jeder Teilmenge ihre charakteristische Funktion zuordnet.

Diese Darstellung liegt der Realisierung von Potenzmengentypen zu Grunde: Realisierung erfolgt gewöhnlich durch Bitfelder, wo das Bit i kodiert, ob das Element $e_i \in M$ zur Teilmenge gehört oder nicht. Statische Realisierung durch Bitfelder (etwa in Pascal) bedeutet, dass die Kardinalität $|M|$ durch die Implementierung beschränkt werden muss.

3) **Signaturdeklaration als Instanz eines Funktionstyps:**

Die Zuordnung von Funktionen zur Signatur einzelner Datentypen ist schwierig, wenn Definitions- und Wertebereich mehrere Datentypen verwenden.

Beispiel:

$$* : \texttt{Matrix} \times \texttt{Vektor} \longrightarrow \texttt{Vektor}$$

Eine konsequente Auflösung dieses Dilemmas ist nur mit Funktionstypen möglich, die

$$\texttt{Matrix} \times \texttt{Vektor} \longrightarrow \texttt{Vektor}$$

als neuen (Funktions)typ verstehen und

$$* : \texttt{Matrix} \times \texttt{Vektor} \longrightarrow \texttt{Vektor}$$

oder in der Schreibweise

```
* : ( Matrix, Vektor ) -> Vektor
```

als Typangabe für die Instanz $*$ interpretieren.

6.5.5 Dynamische Datenstrukturen und Zeigertypen

Als dynamische Datenstrukturen bezeichnet man Datenstrukturen, deren Größe erst zur Laufzeit bekannt ist, so dass der Compiler für Instanzen solcher Typen den erforderlichen Speicherplatz nicht reservieren kann.

Der Speicherplatz muss zur Laufzeit dynamisch bereitgestellt und initialisiert bzw. wieder freigegeben werden. Dazu sind entsprechende Funktionen erforderlich. Bezeichnung: **Konstruktor** und **Destruktor**.

In C müssen diese Funktionen softwareseitig implementiert werden. Grundfunktionen `malloc`, `calloc`, `realloc` und `free`. In anderen Sprachen werden solche Konstruktoren (meist `new`) und Destruktoren (meist `dispose`) automatisch vom Compiler generiert.

Der zur Laufzeit angeforderte Speicherplatz wird aus einem speziellen Speicherbereich, der **Halde**, bereitgestellt. Der *Wert* einer solchen dynamischen Variablen ist also die Adresse (die *Referenz*) dieses Speicherbereichs.

Für dynamische Variablen ist damit zu unterscheiden zwischen der (vom Compiler, also statisch verwalteten) Referenz der Variablen, der Referenz des zugehörigen Objekts und dem eigentlichen dynamisch angelegten Objekt (dem Inhalt) selbst. Die Objektreferenz ist der Wert der Variablenreferenz.

In C besteht direkter Zugriff auf die Adressen von Bezeichnern. Zu jedem Datentyp `T` existiert deshalb der Datentyp `T*` der Referenzen auf Objekte vom Typ `T`.

Name	**Zeigertyp `T*`**
Grundbereich	Referenzen auf Objekte vom Typ `T`
Signatur	$*$: `T*` $\to$ `T` (Inhaltsoperator)
	& : `T` $\to$ `T*` (Adressoperator)
	Zeigerarithmetik
Bemerkung	Die Operatoren können nur auf Objekte im Hauptspeicher angewendet werden.

Deklaration erfolgt in C als `T *var`, d.h. Zeiger werden über den zugehörigen Objekttyp deklariert.

Beispiel:

```
void print(char *s) {
  while(*s) printf("%c",*s++);
}

int main() {
  char buffer[20]="Ein Testbeispiel";
  print(buffer); printf("\n");
}
```

Bemerkung 1:
Für Zeigervariablen `T *a,*b` kopiert die Zuweisung `a=b` die in b gespeicherte Referenz, nicht das zugehörige Objekt `*b`!

Bemerkung 2:
Mit dem direkten Zugriff auf die Arithmetik von Speicheradressen ist zwar sehr effiziente low-level Programmierung möglich, aber damit kann man auch jedes Typsystem aushebeln. Auch ist eine automatische Speicherverwaltung nicht möglich. Deshalb erlauben andere Programmiersprachen (Java, Lisp) einen solchen direkten Zugriff nicht, sondern haben spezielle Datentypkonstruktoren für dynamische Datentypen. Einer davon, der auch in C bekannt ist, ist der Feldkonstruktor: `int s[];` definiert eine dynamische Variable s für ein (statisch oder dynamisch zu erzeugendes) Feld ganzer Zahlen. s enthält die Referenz des Feldes, d.h. dessen Startadresse, `s[0]` (oder `*s`) hat als Wert den Inhalt des ersten Feldelements.

Bemerkung 3:
Für dynamische Daten ist zu unterscheiden, ob mehrere Referenzen auf dasselbe Objekt angelegt werden sollen oder mit Kopien des Objekts gearbeitet wird. Konstruktoren müssen deshalb meist explizit aufgerufen werden. Destruktoren können automatisch aufgerufen werden, wenn das Laufzeitsystem zu jedem Objekt einen Referenzenzähler führt.

Verkettete dynamische Datenstrukturen

Beispiel: Verkettete Listen ganzer Zahlen

```
/* intlist.h */
typedef struct uhu { int eintrag; struct uhu *next; } node ;
```

```
typedef node *intlist;

extern intlist new();
extern intlist l_random(int n);
extern void l_print(intlist l);
extern int l_laenge(intlist l);
/* Ende intlist.h */

intlist new() {
  node *u;
  if (u=(node *) malloc(sizeof(node))) return u;
  else {
    perror("\nNicht ausreichend Speicherplatz vorhanden");
    exit(0);
  }
}

void l_print(intlist l) {
  while(l) {
    printf("%d ",l->eintrag); l=l->next;
  }
  printf("\n");
}

int l_laenge(intlist l) {
  int i=0;
  while(l) { i++; l=l->next; }
  return i;
}

intlist l_random(int n) {
  node *a=NULL; /* Listenanfang */
  node *e;      /* Listenende */
  node *u;      /* Zeigervariable */
  while (n>0) {

    /* neue Zelle anlegen und fuellen */

    u=new();
    u->eintrag=rand()%10;
    u->next=NULL;

    /* neue Zelle einbauen */

    if (a!=NULL) { e->next=u; e=u;}
    else { a=e=u; }
    n--;
  }
  return a;
}
```

Fortführung in der Übungsserie 6.

6.6 Operatoren und Ausdrücke

In Analogie zu mathematischen Termen kennen Programmiersprachen **Ausdrücke** zur Repräsentation von Werten. Ausdrücke bezeichnen also Elemente eines Datentyps.

Ausdrücke setzen sich zusammen aus Literalen, Variablen-, Funktions- und Operatorsymbolen sowie Klammern als Hilfszeichen.

6.6.1 Operatoren und Schreibweisen

Operatoren bezeichnen in die Programmiersprache "eingebaute" Funktionen in spezieller Syntax (z.B. `1+2` statt `+(1,2)`). Folgende Operatorschreibweisen sind gebräuchlich:

Bezeichnung	**Stellung**	**Arität**	**Beispiel**
Präfix	vorangestellt	einstellig	unäres Minus `-1`
Postfix	nachgestellt	einstellig	Postinkrement `x++`
Infix	zwischen den Operanden	zweistellig	Summe `a+b`
Roundfix	um den Operanden	einstellig	Betrag |`x`|
Mixfix	gemischt	mehrstellig	Feldelement `a[i]`

6.6.2 Syntax von Ausdrücken

Ausdrücke müssen zunächst korrekt geschrieben sein (vgl. Beispiel Grammatik der arithmetischen Ausdrücke), d.h. sie müssen syntaktisch korrekt sein. Man spricht in dem Zusammenhang auch von **wohlgeformten Ausdrücken**.

Induktive Definition für Ausdrücke vom Typ `D`:

1. Jedes Literal k vom Typ `D` ist ein Ausdruck vom Typ `D`.
2. Jede Variable v vom Typ `D` ist ein Ausdruck vom Typ `D`.
3. Sind $t_1, ..., t_n$ Ausdrücke der Typen $\mathtt{D}_1, ..., \mathtt{D}_n$ und $f : \mathtt{D}_1 \times ... \times \mathtt{D}_n \to \mathtt{D}$ eine (präfixnotierte) Funktion der angegebenen Signatur, so ist $f(t_1, ..., t_n)$ ein Ausdruck vom Typ `D`.
4. t_1, t_2 Ausdrücke der Typen $\mathtt{D}_1, \mathtt{D}_2$ und $\circ : \mathtt{D}_1 \times \mathtt{D}_2 \to \mathtt{D}$ ein infixnotierter Operator der angegebenen Signatur, so ist $t_1 \circ t_2$ ein Ausdruck vom Typ `D`.

   Entsprechendes gilt für Operatoren anderer Stelligkeit und Schreibweise.

6.6.3 Rangfolge von Operatoren

Unterschiedliche Möglichkeit der Interpretation von `1+2*3`.

Bindungskraft (oder **Präzedenz**) von Operatoren.

Auflistung von Operatorenklassen nach fallender Bindungskraft:

Postfixoperatoren	`() [] -> .`
einstellige Operatoren	`! ++ -- + - * &`
multiplikative Operatoren	`* / %`
additive Operatoren	`+ -`
Schiebeoperatoren	`<< >>`
Relationale Operatoren	`< <= > >=`
Gleichheitsoperatoren	`== !=`
bitweise Operatoren	`& ^ |`
logische Operatoren	`&& ||`
Zuweisungsoperatoren	`= += -= *=` usw.
Kommaoperator	`,`

Beispiele:

`a && b == b && c` wir interpretiert als `a && (b == b) && c`

Der Wert von Ausdrücken, in denen mehrere Operatoren gleicher Bindungskraft Vorkommen, wird durch Angabe der **Assoziativität** der Operatoren eindeutig.

Alle binären Operatoren sind linksassoziativ außer den Zuweisungsoperatoren, die rechtsassoziativ sind. Also `1+2+3` steht für `(1+2)+3`, aber `x=y=1` steht für `x=(y=1)`.

Nach Auflösung dieser (lexikalisch möglichen) Diversitäten können Operatoren wie Funktionen betrachtet werden (und werden vom Parser in der Regel auch in die äquivlente Präfixnotation überführt). Im Weiteren werden wir Operatoren und Funktionen deshalb nicht unterscheiden.

6.6.4 Der Wert von Ausdrücken

Jeder Ausdruck symbolisiert semantisch eine Berechnungsvorschrift und das entsprechende Ergebnis, den **Wert** des Ausdrucks.

Die semantische Abbildung $\mu : A \to W$ der Menge der korrekten Ausdrücke A in die Menge der Werte W ergibt sich aus der (unter Berücksichtigung der Bindungskraft eindeutig auflösbaren) rekursiven Definition der Syntax von Ausdrücken wie folgt:

1. Ein Literal k vom Typ `D` hat als Wert das Element $\mu(k) \in M = M(D)$ aus dem Grundbereich von `D`, für das es steht.

2. Eine Variable v vom Typ `D` hat als Wert $\mu(v) \in M$ das unter der Referenz v gespeicherte Element aus M.

3. Für Funktionen (und Operatoren) $f : \mathtt{D}_1 \times ... \times \mathtt{D}_n \to \mathtt{D}$ muss genauer hingeschaut werden:

   Es muss noch unterschieden werden zwischen dem Funktions**symbol** f und der zugehörigen **Implementierung** $\mu(f)$ der entsprechenden Funktion. Diese kann als Unterprogramm, aber auch nur als Bibliotheksfunktion vorliegen.

   Dieser Unterschied ist analog dem Unterschied zwischen Referenz und Wert von Variablen. An ein Funktionssymbol (Referenz) können zur Laufzeit unterschiedliche Implementierungen angebunden sein (Wert). In C kann das über Funktionszeiger realisiert werden.

   Wir sagen auch, das Funktionssymbol f hat den **Typ** $\mathtt{D}_1 \times ... \times \mathtt{D}_n \to \mathtt{D}$.

   Sind also $M_i = M(D_i)$ bzw. $M = M(D)$ die zu den Typen gehörenden Grundbereiche, so ist $\mu(f)$ eine Funktion mit Definitionsbereich $M_1 \times ... \times M_n$ und Wertebereich M.

   Für Ausdrücke t_i vom Typ $\mathtt{D}_i$ ergibt sich der Wert $\mu(f(t_1, ..., t_n))$ des Ausdrucks $f(t_1, ..., t_n)$ dann als $\mu(f)(\mu(t_1), ..., \mu(t_n))$.

4. Die Zuweisungsoperatoren spielen eine besondere Rolle (exemplarisch für `=`). $\mu(x = a)$ ist nicht $\mu(=)(\mu(x), \mu(a))$, sondern $\mu(=)(\mu_L(x), \mu(a))$, wobei $\mu_L(x)$ für den L-Wert, also die **Referenz** von x steht. Unter dieser wird der berechnete Wert als **Seiteneffekt** gespeichert.

Bemerkung 1:
Auswertungen mit μ bezeichnet man als **call by value**, Auswertungen mit μ_L als **call by reference**. μ kann auf beliebige Ausdrücke, μ_L nur auf Bezeichner (allgemeiner: Ausdrücke mit L-Wert, also z.B. auch auf Elemente eines Felds oder Verbunds) angewendet werden. Manche Sprachen (Pascal) unterstützen beide Auswertungsmechanismen, C dagegen verwendet (bis auf den Zuweisungsoperator) nur 'call by value'. Der Effekt eines 'call by reference' kann in C über Zeigeroperatoren realisiert werden.

Beispiel: Einlesen einer ganzen Zahl `int i; scanf("%d",&i);`

Bemerkung 2:
Operatorsymbole können mehrere unterschiedliche Typen haben (d.h. **überladen** sein). Beispiel: Addition.

$\mu(f)$ hängt in diesem Fall nicht nur vom Funktionssymbol ab, sondern vom Typ der Aufruf- und Rückgabewerte. Ein Funktionssymbol bezeichnet in diesem Fall mehrere Funktionen. In objektorientierten Sprachen kann man auch eigene Funktionen überladen.

Bemerkung 3:
Regel 3 besagt, dass Funktionen von innen nach außen ausgewertet werden, d.h. zunächst werden die Argumente berechnet, danach der Funktionswert.

Die Reihenfolge, in der die Auswertung der Argumente einer Funktion erfolgt, ist (bis auf wenige Ausnahmen – logische Operatoren) nicht festgelegt und kann insbesondere durch Compileroptimierungen von der im Text angegebenen abweichen.

Solchen Code sollte man also nicht schreiben:

```
printf("%d %d\n", ++n, power(2,n));
a[i] = i++;
```

6.7 Anweisungen, Kontrollstrukturen und Blöcke

Als **Anweisungen** hatten wir solche Programmschritte bezeichnet, die den Programmstatus ändern, d.h. einen Laufzeiteffekt haben. Anweisungen sind also Vorschriften zur Verarbeitung von Daten.

Anweisungen setzen sich aus **elementaren Anweisungen** zusammen, die durch entsprechende **Kontrollstrukturen** zu **komplexeren Anweisungen** verbunden sein können.

Elementare Anweisungen in C können sein:

Name	Beispiel
Zuweisungsausdrücke	`x=5; x+=y-6;`
Initialisierungssausdrücke	`int i=5;`
Funktionsaufrufe	`abs(x-7);`
einige weitere Ausdrücke	`n++; i=0,j=1,c=2;`

Elementare Anweisungen werden durch ';' abgeschlossen (; ist also ein **Terminationszeichen**).

Als **Kontrollstruktur** bezeichnet man ein Schema, das die Reihenfolge der Abarbeitung von Anweisungen festlegt. Kontrollstrukturen sind also Mittel zur Algorithmenbeschreibung im Programmtext und werden durch den Compiler zur Übersetzungszeit ausgewertet und in Maschinencode umgesetzt, der nur Sequenzen und Sprünge kennt.

Davon zu unterscheiden ist der **Kontrollfluss**, also die Reihenfolge, in der die Anweisungen zur Laufzeit mit realen Daten wirklich abgearbeitet werden.

Die Kontrollstrukturen eines Programms geben ein Schema für alle möglichen Pfade, die der Kontrollfluss zur Laufzeit nehmen kann. Allerdings kann es auch Pfade geben, die mit keinem zulässigen realen Datensatz realisiert werden.

Kontrollstrukturen werden grafisch auch durch **Struktogramme** dargestellt, wenn man die möglichen Pfade des Kontrollflusses besonders hervorheben möchte.

Die einfachste Kontrollstruktur ist eine Folge von nacheinander abzuarbeitenden Anweisungen, die **Anweisungs-Sequenz**. In EBNF-Notation:

Anweisungs-Sequenz:
    { *Anweisung* }

Wir hatten gesehen (WHILE-Programme), dass es für eine vollwertige Programmiersprache ausreicht, über eine einzige weitere Kontrollstruktur zu verfügen, eine while-Schleife. Das ist natürlich für praktische Anwendungen etwas dürftig.

Höhere Programmiersprachen verfügen gewöhnlich über folgende Arten von Kontrollstrukturen:

Name	**Beispiel in C**
Sequenz	`... ; ... ; ...`
Block	`{ ... }`
Verzweigung	`if, switch`
Iteration	`while, do .. while, for`
Sprung	`goto, break, continue, return`

Die Sequenz ist der natürliche Kontrollfluss zwischen komplexen Anweisungen. C unterscheidet deshalb zwischen Anweisungen und Anweisungssequenzen. In EBNF-Notation:

Anweisung:
    elementare-Anweisung
    | *Block-Anweisung*
    | *Verzweigungs-Anweisung*
    | *Iterations-Anweisung*
    | *Sprung-Anweisung*
    | *markierte-Anweisung*

Die einzelnen Anweisungsarten werden im Folgenden besprochen. Auf markierte Anweisungen gehen wir nicht speziell ein. Auf Sprung-Anweisungen gehen wir im Zusammenhang mit anderen Kontrollstrukturen ein. Sie sind hier der Vollständigkeit halber aufgezählt:

Sprung-Anweisung:
    `goto` *Bezeichner* `;`
    | `continue` `;`
    | `break` `;`
    | `return` [*Ausdruck*] `;`

Auf `goto`-Sprünge gehe ich ebenfalls nicht ein. Sie sollten generell vermieden werden, weil damit der Kontrollfluss sehr unübersichtlich wird (Spaghetti-Code). Einige Programmiersprachen sehen sie deshalb gar nicht vor, andere erlauben nur Sprünge an wohldefinierte Stellen (z.B. nicht in Iterations-Ausdrücke hinein).

6.7.1 Blöcke, Gültigkeit, Lebensdauer, Sichtbarkeit

Ein **Block** ist eine Programmeinheit, die aus einer Folge von Deklarationen, Definitionen und Anweisungen besteht. Er dient vor allem der Zusammenfassung einer Anweisungssequenz zu einer logischen Einheit, einer **Blockanweisung**. Blöcke sind in C (sowie C++ und Java) durch ein Paar geschweifter Klammern `{ ... }` und den eingeschlossenen Programmtext gegeben. Andere Sprachen (Pascal) verwenden die Notation `begin ... end`.

Struktogramm eines Blocks.

Blöcke können ineinander geschachtelt sein (wir sprechen von **äußeren** und **inneren** Blöcken).

Struktogramm geschachtelter Blöcke.

Blöcke können eigene Deklarationen und Definitionen enthalten. In C müssen diese vor den Anweisungen stehen (C++ und Java sind da freizügiger). In C sind nur lokale Variablendefinitionen möglich, Funktionsdefinitionen können nur im Programmrumpf vorgenommen werden. In EBNF-Notation:

Block-Anweisung:
$\underline{\{}$ [{ *Deklaration* | *Definition* }] [*Anweisungs-Sequenz*] $\underline{\}}$

Moderne Programmiersprachen (so auch C) beschränken außerdem die Gültigkeit eines Bezeichners auf den Block, in dem dieser definiert ist. Dies hat vor allem den Zweck der Kapselung von Information. Damit können verschiedene Blöcke jeweils ihre eigene separate Variable `i` oder `x` haben, ohne sich gegenseitig abstimmen zu müssen.

Beispiel:

```
if (n>0) {
  int i; /* neues "i" */
  for(i=0; i<n; i++) {
    ...
  }
} /* Ende des Gültigkeitsbereichs von "i" */
```

Kollisionen zwischen Bezeichnern können auftreten, wenn Blöcke ineinander geschachtelt sind und im inneren Block ein Bezeichner gleichen Namens definiert wird. In diesem Fall gilt die neue Bedeutung bis zum Ende des inneren Blocks, wonach die alte Bedeutung wieder auflebt. Der lokale Bezeichner des inneren Blocks „verdeckt" den Bezeichner des äußeren Blocks (und das zugehörige Objekt). In einigen Programmiersprachen kann auf das verdeckte Objekt durch qualifizierte Namen zugegriffen werden.

Beispiel:

```
{  int x=1; int y;
   {
      int x=2;
      y=x; /* y=2 */
   }
   y=x;    /* y=1 */
}
```

Gültigkeitsbereich (scope) eines zu Beginn eines Blocks definierten Bezeichners nennt man denjenigen Bereich des Programmtexts, in dem der Bezeichner in seiner vereinbarten Bedeutung verwendet werden kann. Er reicht also bis zum Ende des aktuellen Blocks (oder der Datei), abzüglich innerer Blöcke, in denen ein gleichnamiger Bezeichner definiert ist (Scope-Regel). Gültigkeit ist ein Übersetzungszeit-Begriff.

Die **Lebensdauer** (lifetime) des mit dem Bezeichner assoziierten Objekts reicht dagegen vom Eintreten des Kontrollflusses in den Block bis zum Austreten aus dem Block, einschließlich des Verweilens in inneren Blöcken. Lebensdauer ist also ein Laufzeit-Begriff.

Dazu wird zur Laufzeit bei jedem Eintreten des Kontrollflusses in den Block ein Speicherbereich (**Rahmen**, frame) auf dem **Laufzeitstapel** (run time stack) erzeugt, wo die Werte der lokal deklarierten Objekte abgelegt sind. Er verbleibt dort so lange, bis der Kontrollfluss den Block verlässt, worauf der Rahmen wieder entfernt wird.

Beispiel: Bild des Stapelspeichers für obiges Programm

Tritt der Kontrollfluss in ein und denselben Block mehrfach ein, so werden auch mehrere Instanzen des Blockrahmens und damit mehrere Instanzen der lokalen Variablen erzeugt. Solche Variablen werden deshalb auch als **dynamisch** bezeichnet.

Beachten Sie, dass diese Rahmen automatisch vom Laufzeitsystem erzeugt und wieder entfernt werden. Der Laufzeitstapel hat also nichts mit der Halde zu tun, auf der die durch Zeiger referenzierten dynamischen Objekte „leben". Der Haldenspeicher pulsiert nicht. Objekte auf der Halde leben weiter, auch wenn die entsprechende Referenzvariable inzwischen vom Stapel verschwunden ist. Gibt es auf das Objekt keine gültige Referenz mehr, so kann es vom *garbage collector* (Speicherverwaltung mit automatischem Referenzenzähler, in C nur softwareseitig zu implementieren) gelegentlich recycled werden.

Der Stapelspeicher pulsiert dagegen im Takt des Abarbeitens von Blöcken. Werte, die in (dynamischen) Variablen eines inneren Block gespeichert sind, gehen nach Ende des Blocks verloren; sie können gerettet werden, wenn sie noch im inneren Block an eine Variable eines äußeren Blocks zugewiesen werden.

Abweichend von diesem Mechanismus kann eine lokale Variable auch bereits vom Compiler, also **statisch** initialisiert werden. Dabei wird dieser Variablen ein Objekt (Speicherbereich) im Datensegment zugeordnet, so dass der Bezeichner während der gesamten Laufzeit auf dasselbe Objekt verweist. Von einer solchen Variablen gibt es also im Laufe des Programms genau eine Instanz. Der aktuelle Wert der Variablen steht damit der nächsten Blockinstanz zur Verfügung, wird also „gerettet".

Solche Variablen müssen in C mit dem Attribut `static` deklariert werden. Wenn nichts Derartiges vereinbart ist, werden lokale C-Variablen dynamisch angelegt.

Beispiel: Pseudozufallszahlen nach der Kongruenzmethode bestimmen. Dabei ergibt sich die je nächste Zahl z_{n+1} aus der vorhergehenden Zahl z_n und der Rechenvorschrift

$$z_{n+1} = a \cdot z_n + b \ (mod\ m).$$

Für gute Wahlen von a, b, m werden dabei alle Restklassen $(mod\ m)$ in sehr zufälliger Reihenfolge durchlaufen. Realisierung in C:

```
#define MODULUS 999999999989L
#define MULTIPLIER 419669081L

long zufallszahl() {
  static unsigned long u = MULTIPLIER;
  return u = (MULTIPLIER * u) % MODULUS;
}

int main() {
  int i;
```

```
  for(i=1;i<10;i++) printf("%10u\n", zufallszahl());
}

/* Ausgabe:

2778212081  145419721 3994840801 1150136377 1003603921 4033043369
3443401153 1719427097 3759697073

*/
```

Statische Variablen können nur mit Literalen initialisiert werden, da dies vom Compiler geschieht. Dynamische Variablen können dagegen mit beliebigen (typgerechten) Ausdrücken initialisiert werden, da dies zur Laufzeit geschieht.

6.7.2 Verzweigungen

Als **Verzweigung** bezeichnet man eine Programmstelle, von der aus der Kontrollfluss mindestens zwei andere Programmstellen erreichen kann. In maschinennahen Sprachen handelt es sich dabei immer um einen bedingten Sprungbefehl. In höheren Programmiersprachen werden Verzweigungen durch **bedingte Anweisungen** realisiert.

C kennt zwei Arten, If-Anweisungen und Case-Anweisungen. In EBNF-Notation:

Verzweigungs-Anweisung:
    If-Anweisung | *Case-Anweisung*
If-Anweisung:
    `if` `(` *Ausdruck* `)` *Anweisung* [`else` *Anweisung*]
Case-Anweisung:
    `switch` `(` *Ausdruck* `)` `{`
        { `case` *const-Ausdruck* `:` *Anweisungs-Sequenz* }
        [`default` `:` *Anweisungs-Sequenz*]
    `}`

Zugehörige Struktogramme.

Bemerkung 1:
Bei geschachtelten If-Anweisungen bezieht sich `else` immer auf das **letzte** `if`. Notfalls müssen Block-Anweisungen verwendet werden.

Beispiel:

```
if ( n>0 ) /* okay */
  if ( a>b ) z=a;
  else z=b;

if ( n>0 ) /* falsch */
  if ( a>b ) z=a;
else z=b;

if ( n>0 ) { /* okay */
  if ( a>b ) z=a;
  }
else z=b;
```

Bemerkung 2:
Die Auswertung einer Case-Anweisung in C ist etwas knifflig. Deshalb sollten die Anweisungs-Sequenzen jeder Alternative mit `break;` abgeschlossen werden. Damit wird an der angegebenen Stelle der Block der Case-Anweisung verlassen. Mehr dazu im [Kernighan/Ritchie].

Bemerkung 3:
Eine spezielle Verzweigung ist in C mit dem **Bedingungsoperator** `(A ? B : C)` möglich. Es handelt sich dabei allerdings nicht um eine Anweisung, sondern einen Ausdruck. EBNF-Notation:

Bedingungsoperator:
Ausdruck ? *Ausdruck* : *Ausdruck*

Beispiel: `( x>0 ? x : -x)`

6.7.3 Iterationen

Als **Iteration** bezeichnet man das wiederholte Durchlaufen von Anweisungen oder Anweisungssequenzen. Eine Iteration kann nach einer zu Iterationsbeginn bekannten Zahl von Durchläufen beendet werden (**Zählschleife**) oder bei Erfüllung einer Bedingung abgebrochen werden (**bedingte Schleife**).

Der Effekt von Iterationen ergibt sich meist daraus, dass dieselben Anweisungen auf unterschiedlichen Daten ausgeführt werden, die in entsprechenden Schleifenvariablen gespeichert sind. Diese Daten müssen anfangs in den Schleifenvariablen abgelegt und nach jedem Durchlauf aktualisiert werden. Schleifendefinitionen enthalten deshalb stets die folgenden vier Komponenten: **Initialisierung**, **Schleifenkörper**, **Reinitialisierung** und **Test**.

Zählschleifen werden normalerweise als `for`-Konstrukte realisiert, in denen Initialisierung und Reinitialisierung der Laufvariablen sowie der Test automatisch ausgeführt werden.

Beispiel (MuPAD): `s:=0; for i from 1 to n do s:=s+i end_do;`

Die Laufvariable muss dabei einem Datentyp angehören, der eine standardmäßige Nachfolgerfunktion in seiner Signatur hat. In einigen Programmiersprachen kann die Schrittweite und Zählrichtung optional vorgegeben werden. In C müssen solche Schleifen softwaremäßig implementiert werden; `for` wird in einem allgemeineren Sinne verwendet.

Bedingte Schleifen werden unterschieden nach

Schleifen mit Eingangstest (`while (`*test*`) do` *Körper*)

und **Schleifen mit Ausgangstest** (`repeat` *Körper* `until (`*test*`)`).

C stellt beide Typen als `while` und `do-while` Schleifen zur Verfügung. Außerdem ist die `for` Schleife eine spezielle bedingte Schleife mit Eingangstest.

Struktogramme.

Beispiel Struktogramm: Ausgabe einer Menge von Datensätzen, wobei zu jedem Datensatz eine einzeilige Ausgabe generiert wird, 50 Datensätze auf eine Seite passen und jede Seite einen speziellen Seitenkopf hat.

Ausgabe der Datensätze:
while (noch Eingaben vorhanden)
if (Seitenwechsel) Ausgabe Seitenkopf;
else Ausgabe Seitenrumpf;

Ausgabe Seitenrumpf:

while (noch Eingaben für diese Seite vorhanden)
    Datensatz zu Ausgabezeile verarbeiten;

Beispiel: Berechnung der Summe der ersten n natürlichen Zahlen.

```
int s=0; int i=1; /* Initialisierung */
while (i<=n) {    /* Test */
  s+=i;           /* Schleifenkörper */
  i++;            /* Reinitialisierung */
}
```

Beispiel: Berechnung von $x = \sqrt[k]{y}$ nach dem Newtonverfahren bis auf eine vorgegebene Genauigkeit.

Formel: $x' = \frac{k-1}{k}\,x + \frac{y}{k\,x^{k-1}}$

Entwicklung der Iterationsstruktur: do-while Schleife, die gleich mit Reinitialisierung beginnt! Struktogramm.

```
float root(int k, float y, float epsilon) {
  float x_old,x_new;
  x_new=y;
  do {
    x_old=x_new;
    x_new=((k-1)*x_old+y/pow(x_old,k-1))/k;
  } while (fabs(x_new-x_old)>epsilon);
  return x_new;
}
```

Siehe `wurzel.c`. Übersetzung mit `cc wurzel.c -o wurzel -lm` wegen `fabs`.

Syntax:

Iterator-Anweisung:
  `while` `(` *Bedingung* `)` *Anweisung*
  | `do` *Anweisung* `while` `(` *Bedingung* `)` `;`
  | `for` `(` *init* `;` *cond* `;` *reinit* `)` *Anweisung*

Bemerkung 1:
Schleifen mit Eingangstest werden nicht abgearbeitet, wenn die Bedingung bereits am Beginn nicht erfüllt ist. Schleifen mit Ausgangstest werden stets mindestens einmal abgearbeitet.

Bemerkung 2:
'`repeat A until (b)`' in Pascal ist äquivalent zu '`do A while (!b)`' in C.

Bemerkung 3:
Mit `break` und `continue` kann die Abarbeitung des Schleifenkörpers verkürzt werden. `break` springt zur nächsten Anweisung nach der Schleife, `continue` zum Test.

Bemerkung 4:
'`for(init; test; reinit) Anweisung`' ist äquivalent zu

```
{ init; while (test) { Anweisung; reinit; } }
```

Einziger Unterschied: `continue` springt zu `reinit`, nicht zum Test. Deshalb ist hier eine saubere Trennung zwischen Schleifenkörper und Reinitialisierung möglich. `init` und `reinit` können auch Komma-Ausdrücke sein, womit sich mehrere Variablen initialisieren bzw. reinitialisieren lassen.

Bemerkung 5:
'`for(;;) Anweisung`' ist die in C-Programmen übliche Form der Angabe einer Endlosschleife. `Anweisung` ist dabei eine Blockanweisung, die nicht beim Test, sondern an einer anderen Stelle durch `break` oder `return` verlassen wird.

Bemerkung 6:
Iterationen können zu nicht terminierenden Programmen führen. Manchmal ist es schwierig, die Termination für konkrete oder alle zulässigen Eingaben zu beweisen.

Beispiel: Die $(3n + 1)$-Funktion

```
while (n>1) if (n%2) n=3*n+1; else n=n/2;
```

Terminationsbeweise verwenden meist positive ganzzahlige Größen, die in jedem Schleifendurchlauf kleiner werden.

Beispiel: Euklidscher Algorithmus. $|a| + |b|$ wird in jedem Durchlauf kleiner.

```
int gcd(int a, int b) {
  int c;
  while(b) {
    c=a%b; a=b; b=c;
  }
  return a;
}
```

6.8 Unterprogramme – Prozeduren und Funktionen

Ein **Unterprogramm** ist ein parametrisierter Anweisungsblock mit einem Namen.

Der Block kann über diesen Namen aufgerufen werden. An der Stelle des Aufrufs tritt der Kontrollfluss (nach einigen Vorkehrungen) in diesen Block ein und führt die Anweisungen des Unterprogramms aus. Ein solches Unterprogramm kann (einen oder mehrere) Werte zurückgeben (**Funktion**) oder ohne Wertrückgabe abschließen (**Prozedur**). (Für die weitere Verarbeitung relevante) Rückgabewerte müssen bei der Rückkehr aus dem Unterprogramm abgefangen werden, wozu Funktionsaufrufe in der Regel Teil eines Ausdrucks (etwa eines Zuweisungsausdrucks) sind. Bei Prozeduraufrufen wird die Abarbeitung nach der Rückkehr mit der nächsten auf den Prozeduraufruf folgenden Anweisung fortgesetzt.

Da C keine Mehrfachzuweisungen (der Art `(a,b)=(c,d)`) unterstützt, können C-Funktionen nur **einen** Wert zurückgeben. Weiterhin sind in C Prozeduren als Funktionen mit dem Rückgabetyp `void` verfügbar. Wir sprechen deshalb im Weiteren von Funktionen statt Unterprogrammen.

Das Funktionskonzept ist eines der mächtigsten Konzepte imperativer Programmiersprachen: Hierdurch kann jede als Programm formulierte Vorschrift zu einer elementaren Anweisung in einem anderen Programm werden, womit Programmteile leicht wiederverwendet werden können.

Funktionen werden darüber hinaus zur Zerlegung und Strukturierung umfangreicher Programme eingesetzt (**Modularisierung**). So vermeidet man es, häufig benötigte Programmstücke wiederholt niederzuschreiben. Auch können auf diese Weise Details der Implementierung von Datenstrukturen und Operationen vor den Teilen des Programms verborgen werden, die darüber nicht Bescheid wissen müssen (**Kapselung**). Dadurch wird alles klarer, und es ist weniger schwierig, Änderungen vorzunehmen.

6.8.1 Funktionsdefinitionen und -aufrufe

Für Funktionen ist zwischen **Definition**, **Aufruf** und **Abarbeitung des Aufrufs** zu unterscheiden. Die ersten beiden Begriffe beziehen sich auf den Programmtext, der dritte auf die Laufzeit.

Die **Funktionsdefinition** erfolgt einmal, und zwar im Deklarationsteil des Programms. Diese Definition wird zur Übersetzungszeit ausgewertet. Sie besteht aus einem **Kopf** und einem **Rumpf**. Der Kopf enthält den Bezeichner der Funktion sowie die **Liste der formalen Parameter** mit der Deklaration der Übergabeparameter und des Typs des Rückgabewerts. Der Rumpf enthält lokale Deklarationen und Definitionen sowie die Anweisungsfolge, die von der Funktion abzuarbeiten ist. In C ist der Rumpf einer Funktionsdefinition eine Block-Anweisung. Allgemeine Syntax (ohne Spezialfälle):

> *Funktions-Definition:*
>
> > *Rückgabetyp Funktionsname* (*Parameterliste*) *Block-Anweisung*
>
> *Parameterliste:*
>
> > [*Typbezeichner Name* [{, *Typbezeichner Name*}]]

Beispiel (aus `wurzel.c`):

```
float root(int k, float y, float epsilon) {
  float x_old,x_new;
  x_new=y;
  do {
    x_old=x_new;
    x_new=((k-1)*x_old+y/pow(x_old,k-1))/k;
  } while (fabs(x_new-x_old)>epsilon);
  return x_new;
}
```

Ein **Funktionsaufruf** bewirkt, dass an dieser Stelle zur Laufzeit der Kontrollfluss an den Funktionsrumpf übergeht. Der Aufruf erfolgt über den Namen der Funktion, dem die Liste der aktuellen Parameter angefügt ist. Im Programmtext muss an dieser Stelle also wenigstens der Funktionskopf deklariert (und diese Deklaration gültig) sein.

> *Funktions-Aufruf:*
>
> > *Funktionsname* (*Liste-der-aktuellen-Parameter*)

Beispiel (aus `wurzel.c`):

```
int main() {
  int a; float b,c;
  ...
  printf("%d-root(%g) = %g\n",a,b,root(a,b,c));
}
```

Funktionen können in ihrem Definitionsteil andere Funktionen aufrufen. Auf diese Weise entsteht zur Laufzeit eine **Aufrufhierachie**.

Eine Funktion kann sich dabei auch selbst aufrufen (**rekursiver Aufruf**) bzw. können sich Funktionen gegenseitig aufrufen (**verschränkt rekursiver Aufruf**). Insbesondere können sich zu einem Laufzeitmoment mehrere Kopien derselben Funktion in Abarbeitung befinden.

Funktionsdefinitionen mit verschränkt rekursiven Aufrufen sehen wie folgt aus:

```
T1 f1(...) { ...  f2(...)  ...  }

T2 f2(...) { ...  f1(...)  ...  }
```

Da keine Anordnung der Funktionsdefinitionen möglich ist, in der jede der beiden Funktionen vor dem Aufruf bereits **definiert** ist, muss f2 vorab **deklariert** werden (forward-Deklaration in Pascal, Funktionsdeklaration oder Prototyp in C). Damit wird der Funktionsname und -kopf eingeführt, die Definition dagegen auf eine spätere Stelle im Programmtext verschoben:

```
T2 f2(...);

T1 f1(...) { ...  f2(...)  ...  }

T2 f2(...) { ...  f1(...)  ...  }
```

Beispiel:

```
int ungerade(int x);
int gerade(int x) {
  if (x==0) return 1;
  if (x<0) return ungerade(x+1);
  else return ungerade(x-1);
}

int ungerade(int x) {
  if (x==0) return 0;
  if (x<0) return gerade(x+1);
  else return gerade(x-1);
}
```

Betrachten wir nun näher, welche Arbeitsschritte der Compiler bei einem Funktionsaufruf in das Laufzeitsystem einzubauen hat. Zunächst müssen die aktuellen Aufrufparameter dem Funktionsrumpf zur Verfügung gestellt werden. Da zu einem Moment mehrere Kopien der Funktion aktiv sein können, können die Aufrufparameter nicht an eine feste Stelle im Datensegment kopiert werden. Statt dessen muss für jeden Aufruf ein Rahmen auf dem Laufzeitstapel eingerichtet werden.

Dieses Vorgehen entspricht der Einrichtung eines Blocks mit den formalen Funktionsparametern als lokalen Variablen, auf die vor Beginn der Abarbeitung des Rumpfes die Aufrufparameter übertragen werden. Dabei sind verschiedene Übertragungsarten denkbar, die wir weiter unten besprechen werden.

Bild der zwei ineinander geschachtelten Blöcke.

Das Verfahren des Einrichtens dieses Blocks bezeichnet man als **Kopierregel**. Den Block, der einen Funktionsaufruf nach der Kopierregel ersetzt, bezeichnet man als **Inkarnation** oder **Instanz** der Funktion. Die Lebensdauer der lokalen Variableninstanzen ist identisch mit der Existenz der zugehörigen Funktionsinkarnation.

Bei rekursiven Funktionen können bei der Abarbeitung des Programms mehrere ineinander geschachtelte Inkarnationen derselben Funktion existieren.

Beispiel: `gerade(5)`

6.8.2 Parameterübergabe an Funktionen

Wir hatten gesehen, dass bei einem Funktionsaufruf ein neuer Rahmen mit Instanzen der lokalen Variablen sowie der formalen Parameter der Funktionsdefinition auf dem Laufzeitstack angelegt wird, in den die aktuellen Aufrufparameter der Funktion übertragen werden.

Imperative Programmiersprachen kennen die folgenden vier Übergabekonzepte aktueller auf formale Parameter:

- call by value (Wertparameter)
- call by reference (Referenzparameter)
- call by name
- call by value-result

Die wichtigsten Übergabeverfahren sind die ersten beiden; in C (und Java) werden alle Parameter als Wertparameter übergeben.

call by value: Der aktuelle Parameter a_i wird ausgewertet, und der Wert wird dem formalen Parameter e_i (genauer: seiner Inkarnation) zugewiesen. Dieser Mechanismus ist der am weitesten verbreitete und entspricht der Zuweisung e_i=a_i. Da von a_i nur der R-Wert verwendet wird, können Wertparameter mit beliebigen (typgerechten) Ausdrücken belegt werden. Änderungen, die innerhalb der Inkarnation am Wert von e_i vorgenommen werden, gehen (zusammen mit e_i) am Ende des Blocks verloren, haben also **keine** Auswirkungen auf das aufrufende Programm, auch wenn a_i eine Variable ist!

Bemerkung:
Das gilt nicht für die mit dynamischen Variablen verbundenen Objekte, denn es wird (als Wert der Variablen) die Objektreferenz kopiert, nicht das Objekt selbst.

call by reference: Der aktuelle Parameter a_i ist ein Bezeichner (oder allgemeiner ein Ausdruck mit L-Wert, also z.B. eine Komponente eines Felds oder Verbunds). Der L-Wert von a_i wird für den L-Wert des formalen Parameters eingesetzt, so dass beide denselben Speicherplatz referenzieren. Der formale Parameter ist also ein **alias** für den aktuellen Parameter; dessen Wert wird also **nicht kopiert**. Jede Änderung des Wertes des Parameters ist sofort und auch nach Rückkehr aus der Funktion im umgebenden Programm wirksam.

Referenzparameter sind insbesondere für die Übergabe großer (statischer) Objekte (etwa Felder oder Verbunde) geeignet, die als Wertparameter aufwändig kopiert werden müssten. Pascal (`var` *param*) und C++ (`T&` *param*) kennen Referenzparameter, C und Java nicht.

Bei diesem Aufrufverfahren wird also nicht der Wert, sondern die Referenz kopiert, wobei im Weiteren Dereferenzieren und Referenzieren automatisch vorgenommen wird. In C kann man Referenzparameter simulieren, indem explizit mit Referenzen gearbeitet wird.

Beispiel:

```
typedef struct { int F[MAX]; int max; } feld ;

void print_feld(feld *F) {
  int i;
  printf("[%d",F->F[0]);
  for(i=1;i<F->max;i++) {
    printf(", %d",F->F[i]);
  }
  printf("]\n");
}
```

```
int main() {
  feld F=random_feld(10);
  print_feld(&F);
}
```

Java kann weitgehend auf diesen Mechanismus verzichten, da Klassenvariablen dynamische Variablen sind und damit Original und Kopie auf dasselbe Objekt verweisen. Objekte sind, wenn erforderlich, explizit zu kopieren.

Bemerkung:
Wird allerdings einer dynamischen Variablen innerhalb des Funktionsrumpfs ein **neues** Objekt zugewiesen, so ist diese Zuweisung für dynamische Referenzvariablen nach Verlassen des Funktionsblocks noch gültig, für dynamische Wertvariablen (wie in Java) lebt dagegen die alte Zuordnung auf. Die neue Referenz muss ggf. über den Funktionswert zurückgegeben und einer im aufrufenden Kontext gültigen Variablen zugewiesen. (Das zugehörige Objekt wurde auf der Halde angelegt und lebt wenigstens so lange, so lange es referenziert wird.)

call by value-result: a_i muss dazu einen L-Wert besitzen. Die Auswertung erfolgt wie im Fall *call by value*, aber am Ende (also in der Inkarnation nach Verlassen des Funktionsrumpfes) wird das Ergebnis der Rechnung auf a_i zurückkopiert. Die Wirkung ist wie bei Referenzparametern, allerdings wird kopiert. Dieses Verfahren wurde in frühen FORTRAN-Varianten verwendet.

call by name: Der aktuelle Parameter ist ein beliebiger Ausdruck, der ohne jede Auswertung textuell für den formalen Parameter substituiert wird. Seine Auswertung erfolgt also nicht während der Initialisierung der Inkarnation, sondern an jeder Stelle des Funktionsrumpfes, an der der formale Parameter vorkommt. Lässt sich technisch realisieren, indem diesem Parameter während der Initialisierung der Inkarnation eine Funktion zugeordnet wird, die den aktuellen Parameter auswertet. Spielt in klassischen Programmiersprachen kaum eine Rolle, jedoch wichtig im symbolischen Rechnen.

Beispiel:

```
int i=0;
int a[]={10,20};

void p(int x) {
  i=i+1; x=x+2;
}

int main() {
  p(a[i]);
  printf("%d %d\n",a[0],a[1]);
}
```

Ergebnis *call by value* (z.B. C): '10 20'

Ergebnis *call by reference*: '12 20'

Ergebnis *call by name*: '10 22'

call by need: Zur Parameterübergabe zählt auch der Zeitpunkt, zu dem der aktuelle Parameter ausgewertet wird.

Beispiel:

```
float Q(int b, float x) {
  if (b) return x*x;
```

```
  else return 1.0;
}
```

Ist f eine aufwändig zu berechnende Funktion, so bedeutet der Aufruf `Q(0,f(7.5))`, dass zunächst die aktuellen Parameter ausgewertet und übergeben werden. Hier wird also `f(7.5)` berechnet, obwohl es im Weiteren gar nicht benötigt wird. Manche Programmiersprachen schieben deshalb die Parameterübergabe auf, bis sie unumgänglichgeworden ist. Diese verzögerte Auswertung (lazy evaluation) kann aber unangenehme Seiteneffekte haben.

Das Verfahren wird in C bei der Auswertung boolescher Ausdrücke angewendet.

6.8.3 Rekursion

Als **Rekursion** bezeichnet man die Definition eines Problems, einer Funktion oder eines Verfahrens durch Bezug auf sich selbst. Genauer: Das Ergebnis für eine Eingabe x wird auf die Berechnung des Ergebnisses für eine „einfachere“ Eingabe y zurückgeführt, dieses wiederum auf die Berechnung für eine Eingabe z usw. Damit ein rekursiv angegebenes Verfahren terminiert, muss das Ergebnis für mindestens einen Wert, der in dieser Kette von Argumenten erreicht wird, bekannt sein (**Abbruchbedingung**).

Beispiel: Türme von Hanoi (`hanoi.c`)

```
void Hanoi(int k,int a,int b) {
  if(k>=1) {
    Hanoi(k-1,a,6-a-b);
    printf("\t%i. Scheibe : %i => %i\n",k,a,b);
    Hanoi(k-1,6-a-b,b);
  }
}
```

Hinter relativ harmlosen Rekursionsbeziehungen können sich gewaltige **Aufrufhierarchien** verbergen. Beispiel `Hanoi(n)`.

Anzahl der Aufrufe: $A(n) = 2 \cdot A(n-1) + 1,\ A(0) = 1$. Ergibt die Formel $A(n) = 2^{n+1} - 1$. Beweis durch vollständige Induktion.

Neben der Anzahl der Funktionsaufrufe, die zur Laufzeit rekursiv ausgelöst werden, ist auch die **Rekursionstiefe** wichtig, d.h. die Anzahl der gleichzeitig aktiven Inkarnationen der Funktion und damit die auf dem Laufzeitstack maximal gleichzeitig abgelegten Rahmen mit lokalen Variablen. Die ist in den meisten Beispielen linear in einem der Aufrufparameter.

Beispiel: Die Fibonacci-Zahlen

```
long fib(int n) {
  if (n<2) return 1;
  else return fib(n-1) + fib(n-2);
}
```

Explizite Formel für die Anzahl der Aufrufe bestimmen.

$A(n) = A(n-1) + A(n-2) + 1,\ A(0) = A(1) = 1$ oder $(A(n)+1) = (A(n-1)+1) + (A(n-2)+1),\ A(0)+1 = A(1)+1 = 2$.

Zur Erinnerung: Fibonaccifolge $F_0 = F_1 = 1,\ F_{m+2} = F_{m+1} + F_m$

Also gilt $A(n) = 2 \cdot F_n - 1 \sim O(\alpha^n)$ (siehe Kap. 2.5).

Iterationen lassen sich generell auch rekursiv aufschreiben: Der Schleifenkörper wird als Prozedur definiert mit formalen Parametern entsprechend den Iterationsvariablen und der Testbedingung der Schleife als Abbruchbedingung:

Iterative Version

```
p1=e1; ... pk=ek;   /*  Initialisierung  */
do {
  Körper;  /* (Neu)berechnung von result */
  p1=a1; ... pk=ak; /* Reinitialisierung */
} while (Bedingung)
return result;
```

Gleichwertige rekursive Version

```
/* Funktionsdefinition */
resulttyp Rfunc(p1,...,pk) {
  Körper;  /* Berechnung von result */
  if (! Bedingung) return result;
  return Rfunc(a1,...,ak);
}

...
/* Funktionsaufruf */
Rfunc(e1,...,ek);
...
```

Beispiel: Summe der ersten n Zahlen. Rekursionstiefe und Speicherverbrauch im Vergleich zur iterativen Version.

```
/* iterative Lösung */
int isum(int n) {
  int s=0;
  while(n>0) {
    s=s+n; n=n-1;
  }
  return s;
}

/* rekursive Lösung */
int rsum(int n) {
  if (n==0) return 0;
  else return n + rsum(n-1);
}
```

Beispiel: Berechnung des ggT mit Hilfe des Euklidschen Algorithmus.

```
/* iterative Lösung */
int igcd(int a, int b) {
  int c;
  while(b!=0) {
    c=a%b;
    a=b; b=c;
```

```
  }
  return a;
}

/* rekursive Lösung */
int rgcd(int a, int b) {
  if (b==0) return a;
  else return rgcd(b,a%b);
}
```

Solche rekursiven Funktionsdefinitionen lassen sich oft einfach zum Beweis der Exaktheit eines Verfahrens verwenden. Ihnen entspricht das Beweisverfahren der vollständigen Induktion.

Exaktheitsbeweis für `rgcd`.

6.8.4 Rekursion und Iteration

Das Konzept der Rekursion ist dann besonders mächtig und elegant, wenn mehrere rekursive Aufrufe in der Funktionsdefinition abgesetzt werden, etwa zur Bearbeitung baumartiger Datenstrukturen. Der Aufruf so definierter Funktionen erzeugt dann einen Kontrollfluss, der durch wiederholtes Auf- und Absteigen in der Hierarchie der lokalen Funktionsrahmen der Struktur der verarbeiteten Daten folgt.

Weniger sinnvoll ist die Verwendung von Funktionsdefinitionen mit nur einem rekursiven Aufruf, wenn sich diese auch iterativ implementieren lassen, da dann der zusätzliche Aufwand für die zu jedem Aufruf zu erstellenden Speicherrahmen wegfällt.

Manchmal lassen sich allerdings auch solche Verfahren rekursiv einfacher aufschreiben als iterativ. Beispiel: Ausgabe einer Zahl in einem Positionssystem zur Basis $a < 10$, wo im iterativen Verfahren die Ziffern in der „falschen“ Reihenfolge entstehen.

```
void printint(int n, int a) {
  if (n/a) printint(n/a,a);
  printf("%d",n%a);
}
```

In einer Schleife sind die lokalen Variablen des Schleifenkörpers nur ein einziges Mal vorhanden und werden bei jeder neuen Zuweisung überschrieben. Bei rekursiven Funktionsaufrufen werden dagegen jedes Mal neue Inkarnationen der lokalen Variablen angelegt, so dass die alten Werte nach Verlassen dieses Blocks wieder zur Verfügung stehen. Der Speicherbedarf von Programmen mit rekursiven Funktionen ist deshalb höher, die Gestaltungsmöglichkeiten allerdings auch umfangreicher.

Eine rekursive Funktionsdefinition heißt **endrekusiv**, wenn der rekursive Aufruf als letzte Anweisung der Funktionsdefinition erfolgt und der Kontrollfluss danach den Funktionskörper verlässt. Es ist intuitiv klar, dass sich solche Funktionsdefinitionen sehr leicht in eine iterative Form umwandeln lassen, da man die lokalen Variablen ja ohne Not überschreiben kann, weil sie nach der Rückkehr aus dem rekursiven Funktionsaufruf nicht mehr benötigt werden.

Beispiel: Eine iterative Version von `rgcd`.

```
int itergcd(int a, int b) {
  int a0, b0;
  for(;;) {
    if (b==0) return a;
```

```
    a0=b; b0=a%b; /* Argumente neu berechnen */
    a=a0; b=b0;   /* Reinitialisierung */
  }
}
```

Die Hilfsvariablen `a0,b0` sind erforderlich, weil C keine simultanen Zuweisungen der Form `(a,b)= (b,a%b)` kennt (eine Hilfsvariable hätte ausgereicht).

Andere Funktionen sind nicht unmittelbar endrekursiv, etwa die Summenberechnung `rsum`, denn der letzte Aufruf ist eine Addition. Sie kann aber leicht in eine endrekursive Variante verwandelt werden und diese in eine iterative:

```
int rsum1(int n) {
  return Tailrsum(n,0);
}
int Tailrsum(int n, int k) {
  if (n==0) return k;
  else return Tailrsum(n-1, n+k);
}
int itersum(int n) {
  int k=0;
  for(;;) {
    if (n==0) return k;
    k+=n; n--;  /* Argumente neu berechnen */
  }
}
```

Formale Definition: Eine Funktion $f(x)$ heißt **endrekursiv**, wenn sie sich in der Form

$$f(x) \;=\; \begin{cases} g(x) & \text{wenn } P(x) \text{ gilt} \\ f(r(x)) & \text{sonst} \end{cases}$$

darstellen lässt (wobei die Funktionen g und r die Funktion f nicht rekursiv aufrufen).

Rekursive Implementierung:

```
int f(int x) {
  if (P(x)) return g(x);
  else return f(r(x));
}
```

Gleichwertige iterative Implementierung:

```
int f(int x) {
  while (!P(x)) x=r(x);
  return g(x);
}
```

Moderne Compiler nehmen eine solche Umwandlung endrekursiver Funktionen im Rahmen der Codeoptimierung unter Umständen automatisch vor.

Um komplexere rekursive Funktionen (insbesondere solche mit mehreren rekursiven Aufrufen in der Funktionsdefinition) in eine iterative Variante zu verwandeln, muss zusätzlich vom Programmierer ein Stapelspeicher verwaltet werden, der die noch benötigten Werte von lokalen Variablen verschiedener Inkarnationen der rekursiven Funktion verwaltet. Der Aufwand kann sich allerdings rechnen.

Beispiel: Fibonacci-Funktion. Hier werden nur die beiden unmittelbar zuvor berechneten Funktionswerte benötigt:

```
long iterfib(int n) {
  int a=1; int b=0; int c;
  while(n--) {
    c=a+b;
    b=a; a=c;
  }
  return c;
}
```

Siehe `fib.c`. Die Laufzeit sinkt damit von $2^{O(n)}$ für `fib` auf $O(n)$ für `iterfib`. Eine noch schnellere Berechnung von $fib(n)$ (Zeitbedarf $O(\log(n))$) ist über eine Matrixidentität und schnelles Potenzieren möglich.

7 Zur Geschichte der Informatik

Legen wir die Definition „Informatik“ des [Duden Informatik] zu Grunde

> Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen – insbesondere der automatischen Verarbeitung mit Hilfe von Digitalrechnern.

so beginnt die Geschichte der Informatik mit dem Aufkommen der ersten Digitalrechner Mitte des 20. Jahrhunderts.

Allerdings hat eine solche Informatik eine deutliche und weit zurückreichende **Vorgeschichte**, die sich grob zwei Entwicklungslinien zuordnen lässt:

- den Bemühungen der Menschen um die Algorithmisierung von Tätigkeiten (in denen Rechenverfahren eine wichtige Rolle spielen)
- und der Entwicklung mechanischer Rechenhilfsmittel.

7.1 Zur Entwicklung des Algorithmenbegriffs

Die erste Linie – Handlungsabläufe so detailliert zu spezifizieren, dass sie sich rein syntaktisch und ohne Kenntnis der Hintergründe reproduzieren lassen – hat nicht erst mit dem „dummen Computer“ ihre Bedeutung erhalten, sondern spielt in der Menschheitsgeschichte seit der Trennung von Hand- und Kopfarbeit eine Rolle. Die großen Pyramiden wären nicht gebaut worden, hätte es nicht eine solche Möglichkeit gegeben, Millionen Hände nach einem einheitlichen Plan in Bewegung zu setzen.

Zentrales Element einer solchen Planung sind neben Kenntnissen von Geometrie, Statik, Werkstoffkunde usw. kalkulatorische Fähigkeiten, die im Ägypten der Pharaonen bereits hoch entwickelt waren. Auch die Leistungen der Babylonier, Chinesen, Inder und Araber sowie im europäischen Raum der griechischen und römischen Antike sind hier zu nennen. Für praktische Zwecke waren diese Kenntnisse, über die nur einige wenige am Hof von Fürsten und Königen oder in Klöstern und anderen Orten der Wissenschaft verfügten, in verständliche Handlungsanweisungen im Rahmen eines einheitlichen Plans herunterzubrechen, also zu algorithmisieren.

Sicher geht ein derart breites Algorithmenverständnis als Vorgeschichte der Informatik zu weit. Ich beschränke mich deshalb im Weiteren auf überlieferte Rechenverfahren wie etwa das von Euklid (etwa 300 v.Chr.) zur Bestimmung des größten gemeinsamen Teilers. Solche Beschreibungen algorithmischer Verfahren spielen in der wissenschaftlichen Literatur der vorantiken Zeit die zentrale Rolle, während der deduktiven Darstellung von Zusammenhängen oder gar Beweisen nur eine marginale Bedeutung zukommt.

Algorithmisches Wissen prägte auch die mittelalterliche Mathematik Europas, die besonders von arabischen Gelehrten beeinflusst wurde, die die indischen und griechisch-hellenistischen Traditionen bewahrt und fortgeführt hatten. Eine zentrale Rolle kommt dabei dem Hauptwerk des im 9. Jh. lebenden persischen Mathematikers und Astronoms Ibn Musa Al-Chwarismi zu, dem Lehrbuch „Kitab al jabr w'almuqabala“ („Regeln der Ersetzung und Reduktion“). Der Name des Gelehrten ist die Wurzel des Worts „Algorithmus“ und der Titel des Buches die des Namens des mathematischen Teilgebiets „Algebra“.

Das Rechnen hat die Menschen von jeher fasziniert und es wurde – mehr noch als das Schreiben – bis zum Beginn der Neuzeit als Kunst angesehen, die wenigen Auserwählten vorbehalten war. Viele große Mathematiker, die wir heute vor allem wegen ihrer analytischen Arbeiten schätzen, waren zu ihren Lebzeiten vor allem als große Rechenmeister bekannt. So spielte das Hauptwerk von Gauß, seine „Disquisitiones arithmeticae“, zu Lebzeiten des Meisters kaum eine Rolle. Berühmt wurde er damals vor allem mit seinen vielfältigen Berechnungen der Bahnen von Himmelskörpern (etwa denen, die zur Wiederauffindung des Planetoiden Ceres führten, den Astronomen entdeckt, dann aber wieder verloren hatten) sowie durch seine trigonometrischen Landvermessungen im Braunschweigischen.

Die herausragende Rolle von Wissenschaft und Wissenschaftlern zu jener Zeit ergibt sich daraus, dass es nur wenige Wissenschaftler gab, im Alltag dagegen wissenschaftliche Kenntnisse eine absolut untergeordnete Rolle spielten. Selbst elementare Rechenfertigkeiten, die heute bereits in der Grundschule erlernt werden, wurden erst im 16. Jahrhundert von Rechenmeistern wie Adam Riese (1492-1559) „unters Volk“ gebracht. Es ist hier nicht der Raum, auch die objektiven Gründe für eine solche Entwicklung näher zu beleuchten, die sicher mit dem aufkommenden Industriezeitalter zu tun haben.

Das Verständnis der Metapher von den „Millionen Händen, die nach einem einheitlichen Plan in Bewegung gesetzt werden“, ist mit einer solchen weiteren Verbreitung wissenschaftlicher Kenntnisse sicher zu modifizieren, auch wenn das gerade zu Ende gegangene 20. Jahrhundert noch einmal von gigantischen Projekten und verschiedenen Gesellschaftssystemen geprägt war, die sich den „einheitlichen Plan“ (und dazu die einheitliche Führung) auf die Fahnen geschrieben hatten. Pläne werden in Zukunft kompliziertere Struktur haben und Begriffen wie Kompetenz, Kooperation und Kreativität, also individueller Selbstentfaltung, stärker Raum geben müssen.

Auch die Möglichkeiten des „dummen Computers“ als dem dann letzten Refugium eines eng sequenziell-deterministischen Algorithmenbegriffs sind mit einem solchen Bild des unterwürfigen (und manchmal auch aufmüpfigen) Sklaven oder gar nur Rechenknechts nicht ausgereizt. Autonome Agenten, Suchmaschinen und Roboter zeigen, dass sich mit dem Begriff der **Nebenläufigkeit** von Prozessen auch in der Computerwelt ganz neue Fenster öffnen, die – vielleicht – sogar über den Anspruch der Church'schen These hinausreichen.

7.2 Zur Geschichte mechanischer Rechenhilfsmittel

Der zweite Zweig der Vorgeschichte, die **Geschichte mechanischer Rechenhilfsmittel**, reicht vom Abakus über verschiedene Rechenmaschinen von Schickard, Leibniz und Pascal bis hin zu lochkartengesteuerten Webstühlen und Fakturiermaschinen.

Die Geschichte des **Abakus**, des in verschiedenen Formen auch heute etwa in Rußland noch gebräuchlichen Rechenbretts, kann bis ins China der Chon-Dynastie (2. Jh. vor Chr.) zurückverfolgt werden. In seiner Grundidee entspricht er bereits dem Rechenwerk eines Computers, wobei die Arithmetik von den geschickten Fingern seines Benutzers ausgeführt wird und Ein- bzw. Ausgabe durch visuelle Inspektion erfolgt.

Rechenmaschinen spielen in Europa seit dem 17. Jahrhundert eine Rolle, nachdem sich mit der Verbreitung von Rechenfertigkeiten (Adam Riese) sowie der Verdrängung der römischen durch die indischen Zahlzeichen die Bedürfnisse nach und Voraussetzungen für die Automatisierung von Rechenvorgängen verändert hatten.

Die erste Rechenmaschine wurde 1623 vom Tübinger Professor Wilhelm Schickard (1592-1635) für dessen Freund Johannes Kepler konstruiert, die die vier Grundrechenarten beherrschte, aber weitgehend unbeachtet blieb.

Eine ähnliche Maschine wurde im Jahre 1641 von Blaise Pascal (1623-1666) entwickelt, mit der man sechsstellige Zahlen addieren konnte.

1674 konstruiert Gottfried Wilhelm Leibniz (1646-1716) eine Rechenmaschine mit Staffelwalzen für die vier Grundrechenarten. Die Operanden wurden dabei auf dem Räderwerk eines verschiebbaren Schlittens eingestellt, was als Operationsprinzip von fast allen nachfolgenden Rechenmaschinen übernommen wurde.

Hauptproblem blieb aber die unzureichende Feinmechanik jener Zeit. Erst um 1774 gelang es Philipp Matthäus Hahn (1739-1790), serienmäßig Rechenmaschinen mit hoher mechanischer Zuverlässigkeit herzustellen. Ab 1818 wurden Rechenmaschinen nach dem Leibniz'schen Vorbild serienmäßig hergestellt und dabei ständig weiterentwickelt.

All diesen Maschinen fehlte aber ein wichtiges Element des Computers – die **Programmierbarkeit**. Sie konnten zwar Algorithmen für die vier Grundrechenarten ausführen, aber keine anderen. Die Algorithmen manifestierten sich in ihrem Räderwerk und waren damit sozusagen „fest eingebaut". Sie entsprachen damit funktional dem Rechenwerk moderner Computer.

Erste Überlegungen für einen frei programmierbaren „Computer" entwickelte Charles Babbage (1792-1871) im Jahre 1838 mit seiner „Analytical Engine", nachdem er mit seiner „Difference Engine" eine eigene Rechenmaschine entwickelt und gebaut hatte. Bei seiner „Analytical Engine" sollte die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene Lochkarten gesteuert werden. Die Maschine sollte einen Zahlenspeicher, ein Rechenwerk, eine Steuereinheit und einen Programmspeicher besitzen. Allerdings ließen sich diese Ideen zu einer solchen „programmgesteuerten Maschine" zu jener Zeit noch nicht praktisch realisieren und wurden als „Unsinn" abgetan, obwohl sie das Prinzip des von-Neumann-Rechners bereits vorweg nahmen. Bis zur Entwicklung moderner Rechenmaschinen gerieten sie in Vergessenheit.

Lochkarten spielten zur Programmsteuerung aber auch auf anderen Gebieten eine Rolle. Die ersten derartigen maschinell lesbaren Informationsträger wurden für Webstühle von dem französischen Mechaniker Falcon um 1728 entwickelt. Sie bestanden aus Holzplättchen, in denen der Code für ein bestimmtes Webmuster gespeichert war. Die Plättchen wurden als Lochkartenkette über einen mechanischen Ablesemechanismus geführt, der die Kettfäden entsprechend positionierte. Diese Idee wurde aber erst 1805 von Joseph-Marie Jacquard in Lyon zu einer ausgereiften Technik entwickelt. Später wurden solche Lochkarten auch in Grammophonen, Spielorgeln und automatischen Klavieren angewendet.

1886 entwickelt Hermann Hollerith (1860-1929) in den USA eine elektrisch arbeitende Zählmaschine für Lochkarten, mit denen die statistischen Auswertungen der Volkszählungen vorgenommen werden. Er konzipiert dazu Lochkarten, in die Angaben über Personen nach einer bestimmten Verschlüsselung eingegeben wurden. Zur Auswertung der Daten wurden die Karten in eine elektrische Apparatur eingelegt, die über Fühler die Informationn der Lochungen an magnetische Zählwerke weitergab. Weiterhin konstruierte Hollerith eine Sortieranlage für Karten, die durch Klappen in bestimmte Sortierfächer abgelegt wurden. Diese Erfindungen leiteten den eigentlichen Beginn der modernen Datenverarbeitung ein.

7.3 Der Start ins Computerzeitalter

Damit kommen wir zur eigentlichen **Geschichte der Datenverarbeitung**. Sie beginnt im Jahr 1934 mit Konrad Zuses (1910-1995) Arbeiten an einer programmgesteuerten Rechenmaschine, die in ihrem Grundkonzept der Maschine von Charles Babbage glich. Zudem erkannte **Zuse** frühzeitig den Vorteil, den die Durchführung von Rechenoperationen mit Dualzahlen bietet. Der erste, allerdings nie richtig funktionsfähige Prototyp, die noch auf rein mechanischer Basis arbeitende Z1, wurde 1937 fertig. Der Durchbruch zu einer voll funktionsfähigen Anlage gelang Zuse dann 1941 mit der Z3, einer auf der Basis von Relais arbeitenden elektro-mechanischen Anlage. Dies ist der erste funktionsfähige programmgesetuerte Rechenautomat. Er beherrschte die vier Grundrechenarten und war auch in der Lage, Wurzeln zu berechnen. Das Programm wird mit Lochstreifen eingegeben. Die Anlage verfügte über 2000 Relais und eine Kapazität von 64 Worten á 22 Bit. Multiplikationszeit: ca. 4 s. Zuses Verdienst ist auch die Einführung der Gleitkommadarstellung.

Howard H. Aiken (1900-1973) erstellte im Jahr 1944 in Zusammenarbeit mit der Harvard-Universität und der Firma IBM eine ebenfalls auf elektromechanischer Basis arbeitende Rechenmaschine, die **Mark 1**. Sie war beträchtlich größer als die von Zuse (15 m lang, 100 000 Einzelteile, 80 km

Leitungsdrähte). Die Maschine arbeitete noch im Dezimalsystem. Multiplikationszeit: ca. 6 s.

Inzwischen hatte man gelernt, dass sich auch Elektronenröhren, wie sie damals in der Radiotechnik üblich waren, als Schalter benutzen lassen. Bei ihnen fallen alle mechanisch bewegten Teile weg, und die Zeit für die Ausführung eines Schaltvorgangs konnte von der Größenordnung 100 ms ($\hat{=}$10 Hz) bei Relais auf 0.1 ms ($\hat{=}$10 kHz) verringert werden. 1946 stellten John P. Eckert und John W. Mauchly mit der **ENIAC** (electronic numerical integrator and automatic calculator) den ersten solchen funktionsfähigen Röhrenrechner her. Multiplikationszeit: 2.8 ms. Haupteinsatzgebiet war die Berechnung der Bahn von Flugkörpern.

Es fehlt aber nch eine Eigenschaft, die die Entwicklungen Zuses und der Amerikaner von den heutigen Computern unterscheidet: die **Art der Programmspeicherung**. Bei Zuse geschah dies mit Lochstreifen und beim ENIAC waren es Steckbretter, wie sie in Telefonzentralen zur Handvermittlung benutzt wurden. John von Neumann (1903-1957) hatte die Idee, das Programm im gleichen Speicher des Rechners abzulegen wie die Daten. Der Zugriff auf das Programm wurde damit nicht nur viel schneller als über den (langsamen) Lochstreifenleser, sondern auch viel flexibler. Am bedeutsamsten aber war, dass dadurch die Programme wie Zwischenergebnisse im Speicher standen, vom Rechner direkt ausgeführt und sogar von einem Programm verändert werden konnten. Von Neumann hatte diese Idee schon 1946 schriftlich festgehalten, sie wurde aber erst 1949 von M.V. Wilke in der Rechenanlage EDSAC (electronic delay storage automatic calculator) an der Univ. Cambridge, England, realisiert. Das ist zugleich der erste universelle Rechner mit gespeichertem Programm.

7.4 Die Computer-Generationen

Ab 1950 werden Rechner industriell produziert und das Computerzeitalter hat endgültig begonnen. Die folgende Zeit wird grob in folgende Perioden eingeteilt:

0. Generation (bis 1946)

Programmierbare *elektromechanische* Rechenmaschinen nach den Prinzipien von Babbage.

1. Generation (bis 1955)

Übergang zu Röhrenrechnern und Einsatz von Elektronik.

Einsatz externen Speichern auf Magnettrommeln und Magnetbändern (für wenige 100 Maschinenwörter), handverdrahtet, etwa 1000 FLOPS.

Programmierung direkt in Maschinensprache, symbolische Maschinensprache (1950 ASSEMBLER von H.V. Wilkes), Beginn der Entwicklung erster Betriebssysteme (Gene Amdahl bei IBM)

Erste kommerzielle Systeme, etwa der 1952 in Serie gegangene IBM-Großrechner 701.

Beginnender kommerzieller Rechnereinsatz, aber mit Blick auf die neue Technologie und nicht auf Kostenersparnisse.

2. Generation (1950-1965)

Transistoren als Schaltelementen (dadurch deutlich geringere Leistungsaufnahme, nur noch etwa 100 Watt)

Einsatz von Ferritkernspeichern als Hauptspeicher sowie Band-, Trommel- und Plattenspeichern als Sekundärspeicher, etwa 10 kFLOPS.

Entwicklung von Systemsoftware, Compilern und ersten Universalsprachen (COBOL, FORTRAN, ALGOL), die unabhängig vom Computerfabrikat sind.

Rechnereinsatz im Wesentlichen im Stapelbetrieb, manuelle Datenverarbeitungsmethoden kopierend. Erste anspruchsvollere Anwendungen, z.B. für die Platzreservierung im Luftverkehr.

Programmieren als „Kunst".

3. Generation (1960-1974)

Drastisch sinkende Hardwarepreise durch Einsatz intergierter Schaltkreise, Rechenleistung etwa 500 kFLOPS.

1964: IBMs berühmte Großrechnerserie 360 (entwickelt unter der Leitung von Gene Amdahl).

Aufkommen komfortabler Dialogstationen, die Kommunikation mehrerer Nutzer mit einem Rechner im Dialogbetrieb möglich machen und die Zugriffsmöglichkeiten zu Rechnern wesentlich vereinfachen (Bildschirme, grafische Datenverarbeitung).

Einsatz von Betriebssystemen mit Dialogbetrieb (im time sharing), Datenbanken.

Komplexere Rechnerverbundsysteme führen zur ersten „Softwarekrise", da die zum Teil sehr komplexen Anforderungen mit den bis dahin üblichen Entwurfsmethoden nicht mehr zu bewältigen waren. Methoden der „strukturierten Programmierung" (Pascal), die ingenieurmäßig eingesetzt werden.

Dezentralisierung und Einsatz von Kleinrechnern als Funktionsrechner in der Prozess-Steuerung, wo große Echtzeitrechner versagten. Einsatz von großen Echtzeitrechnern als Dispositions- und Koordinationsrechner.

4. Generation (seit 1970)

Überwiegender Einsatz hochintergierter (VLSI) Schaltkreise, ein Prozessor auf einem Chip, 8-Bit-Architektur, etwa 10 MFLOPS.

Kleinrechner als Personal- und Homecomputer beginnt seinen Siegeszug. (1973: INTEL bringt den 8080 auf den Markt, IBM-Mikrocomputer 5100 mit 64 kByte Arbeitsspeicher, für 8 975 $, 1977: Apple Computer von Steve Jobs und Stephen Wozniak). Dezentralisierung von Datenverarbeitung. Übergang zu multimedialen Anwendungen.

Entwicklung verschiedener Superrechner (CDC, Cray).

Zusammenwachsen kommerzieller Datenverarbeitung und rechnerintegrierter Fertigungssysteme. Rechnernetze (LAN)

Siegeszug von Betriebssystemen wie Unix und MS-DOS, Programmiersprachen wie C, LISP und PROLOG.

Erste elektronische Taschenrechner von Texa Instruments (1972) und Hewlett-Packard (1973), ab 1976 sogar frei programmierbare Taschenrechner von HP

5. Generation (seit 1982)

Höchstintegrierte Schaltkreise, mehrere kleine Prozessoren auf einer CPU, 16- und 32-Bit-Architekturen, Rechnernetze, Parallelrechner

Einsatz verteilter Systeme mit hoher Kommunikationsfähigkeit, gute Arbeits- und Programmierumgebungen. Abkehr von der von-Neumann-Architektur

Verarbeitung nicht-numerischer Daten gewinnt an Bedeutung (komplexe Textverarbeitungen, Datenbanken, Expertensysteme

Zweite Softwarekrise: „Wiederverwendungskrise" (Management von Anpassung und Wiederverwendung der wachsenden Softwarebestände), erste Ansätze objektorientierter Programmierung und Einsatz von Programmgenerator-Werkzeugen.