Grundlagen der Programmierung

Uwe Schmidt
FH Wedel

WS 2000/01

Inhaltsverzeichnis

Literatur

[Abelson 85] Abelson,H.,Sussman,G.J.:
Structure and Interpretation of Computer Programms
MIT Press, 1985

[Backhouse 89] Backhouse,R.C.:
Programmkonstruktion und Verifikation
Hanser, München 1989
ISBN 3-446-15056-0

[Duden 93] *Duden "Informatik"*
2.Auflage, BI, Mannheim, 1993
ISBN 3-411-05232-5

[Barber 90] Barber,R.L.:
Fehlerfreie Programmierung für den Software–Zauberlehrling
Oldenbourg Verlag, München 1990
ISBN 3-486-21637-6

[Bauer 91] Bauer,F.L.,Goos,G.:
Informatik: eine einführende Übersicht
4.Aufl., Springer, Berlin, 1991,
ISBN 3-540-52790-7

[Cohen 90] Cohen,E.:
Programming in the 1990s:
An Introduction to the Calculation of Programs
Springer, New York 1990,
ISBN 0-387-97382-6

[Ottmann 90] Ottmann,T., Widmayer,P.:
Algorithmen und Datenstrukturen

BI Wissenschftsverlag, Mannheim, 1990
ISBN 3-411-03161-1

[Wirth 86] Wirth,N.:
Algorithmen und Datenstrukturen mit Modula–2
4.Aufl., Teubner, Stuttgart, 1986
ISBN 3-519-02260-5

[Si 97] Schmidt,U.:
http://www.fh-wedel.de/~si/buecher.html
Eine kommentierte Bücherliste

1 Einleitung

1.1 Algorithmus

Computer führt Routineaufgaben aus

- einfache Operationen (Addition, Vergleich)
- hohe Geschwindigkeit

Fragen Welche Operationen?
Welche Reihenfolge der Operationen?

$\hookrightarrow$ Beschreibung durch **Algorithmus**

Algorithmus *(erste Definition)*
beschreibt eine Methode zur Lösung einer Aufgabe,
besteht aus einer endlichen Folge von Schritten (einfache Operationen)

$\hookrightarrow$ keine Besonderheit der Informatik

Prozeß Abarbeitung eines Algorithmus

Prozessor Einheit die einen Prozeß ausführt

Algorithmus in der Datenverarbeitung:
Ein Algorithmus berechnet aus Eingabedaten Ausgabedaten (Resultate)

formal Ein Algorithmus berechnet eine Funktion

$f : E \longrightarrow A.$

E ist der Wertebereich der Eingabedaten,
A ist der Wertebereich der Ausgabedaten.

Daten Werte aus bestimmten Wertebereichen

Computer ein spezieller Prozessor

Komponenten

- Zentraleinheit **Central Processing Unit, CPU**, Ausführung der Basisoperationen
- Speicher
  - Daten mit denen die Basisoperationen manipulieren
  - Operationen des Algorithmus
    das **Programm**
- Ein– und Ausgabe–Geräte *I/O*

Geschwindigkeit $10^6 - 10^9$ Operationen/Sekunde.

- *traditionell*: sequentiell, eine Operation zur Zeit
- *RISC (reduced instruction set computer)*: mehrere Operationen gleichzeitig, aber nur eine CPU
- *parallel*: mehrere CPU's gleichzeitig

Zuverlässigkeit sehr hoch

Fehlerursache der Algorithmus

Prozeß berechnet eine Funktion für bestimmte Eingabedaten

Speicher immer billiger $\Rightarrow$immer größer

1970 : 64 KByte
1980 : 640 KByte
1990 : 8 MByte
2000 : 256 MByte

Kosten pro Operation immer billiger

1.2 Programme und Programmiersprachen

Algorithmus in einer Sprache formulieren, die der Prozessor versteht

Interpretation
- verstehen, was jeder Schritt bedeutet
- Operation ausführen

Programmier–sprache Sprache, in der ein Algorithmus für einen Computer formuliert wird

Programm ein in einer Programmiersprache formulierter Algorithmus

programmieren Algorithmen in Programme umsetzen

Elementar–operationen Operationen, die ein Prozessor ausführen kann

Sprachhierarchie maschinennah ⇒ problemorientiert

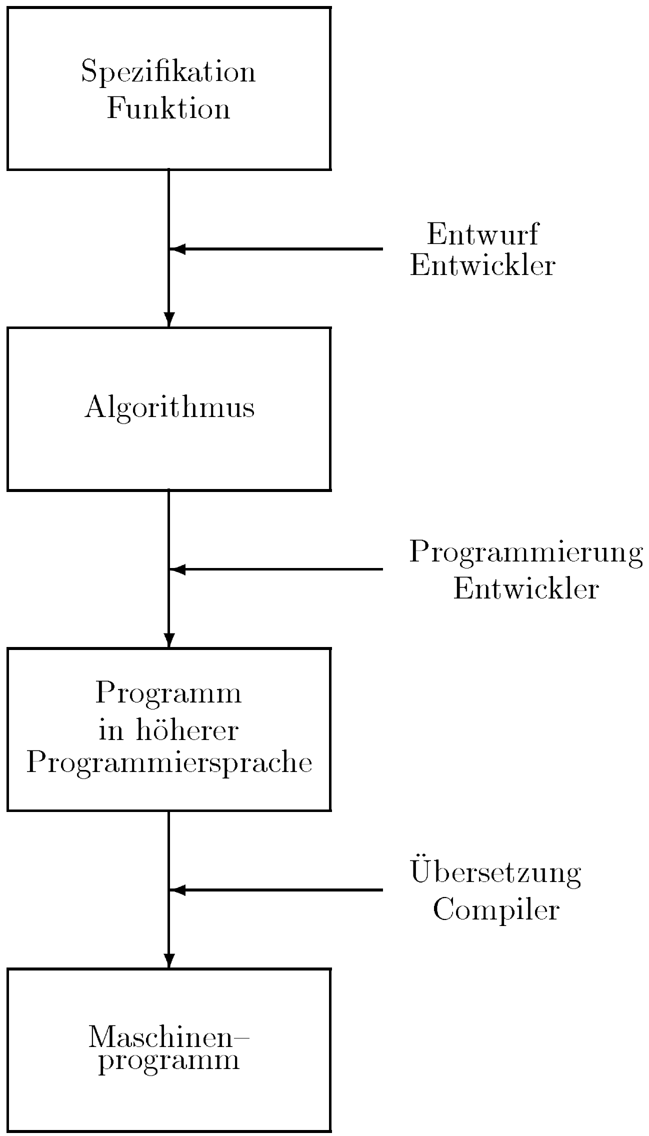
Maschinensprache Programmiersprache, die ein Computer direkt versteht (eine Folge von 0-en und 1-en)

Assemblersprache Maschinensprache in einer für Menschen lesbaren (nicht unbedingt verständlichen) Form: Jede Instruktion erhält einen Namen

Assembler Ein **Programm** zur Transformation einer Assemblersprache in die zugehörige Maschinensprache

höhere Programmier–sprache Zur Vereinfachung der Programmierung Anpassung der Programmiersprache an problem– und aufgabenorientierte Notation
- komplexere Elementaroperationen
- übersichtlichere Anordnung der Anweisungen

Compiler ein **Programm** zur Transformation von Programmen einer höheren Programmiersprache in die Maschinen– oder Assemblersprache eines Computers



Phasen in der Programmentwicklung

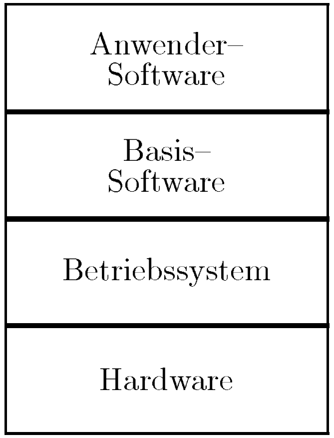
Compiler auch ein Programm

compilieren ausführen eines Algorithmus für die Transformation von Programmen einer höheren Programmiersprache in eine Maschinensprache

Programmier–sprachen

- ≈ **ab 1960** Fortran, Algol 60
- ≈ **ab 1965** Cobol, Basic, Lisp
- ≈ **ab 1970** PL/1, Algol 68, Simula 67
- ≈ **ab 1975** Pascal, C, Prolog
- ≈ **ab 1980** Modula, Ada, Smalltalk
- ≈ **ab 1985** C++, Eiffel
- ≈ **ab 1995** Java, Haskell

Schichtenmodell Rechnersysteme:



Abgrenzung Hardware $\Leftrightarrow$ Software: fließend

Hardware implementiert eine Menge von Elementaroperationen

Betriebssystem erweitert diese Menge um neue Elementaroperationen, die durch (kurze) Programme implementiert sind

Basis–Software erweitert diese Menge nochmals, z.B. durch E/A–Operationen

$\hookrightarrow$ Für die Programmentwicklung ist es unwesentlich, wie die Elementaroperationen implementiert sind, entscheidend ist, welche Operationen verfügbar sind

---

Beispiele

Arithmetik für reelle Zahlen
in Hardware ($\Leftarrow$ Coprozessor)
in Software ($\Leftarrow$ Emulation)

Multiplikation als Instruktion
durch Zurückführen auf Addition

$\hookrightarrow$ Algorithmenentwicklung auch für die Hardware–Entwicklung von Bedeutung

Operationen im Rechner und Betriebssystem sind

Funktionen Eine Funktion ist eine eindeutige Zuordnung von Elementen einer Menge D zu den Elementen einer Menge R. Jedem Element aus D darf höchstens ein Element von R zugeordnet sein. Ist f eine solche Funktion, so schreibt man

$f : D \longrightarrow R$

Urbildbereich D heißt Urbildbereich

Bildbereich R heißt Bildbereich

Definitionsbereich Nicht jedem Element aus D muß ein Element aus R zugeordnet sein. Ist einem Element d kein Element aus R zugeordnet, so ist f für d nicht definiert. Die Menge der Elemente von D, für die f definiert ist, heißt Definitionsbereich und wird mit

$Def(f)$

bezeichnet.

Operationen auf den Wertebereichen sind Funktionen

Alle Operatoren $(+, -, *, \mathsf{div}, \mathsf{mod}, \wedge, \vee, \ldots)$ sind Namen für Funktionen

totale Funktion f heißt totale Funktion, wenn

$Def(f) = D$

ist.

partielle Funktion f heißt partielle Funktion, wenn

$Def(f) \subset D$

ist.

Bild Ordnet die Funktion f dem Element $d \in D$ das Element $r \in R$ zu, so heißt r Bild von d unter f. Man schreibt

$f : d \mapsto r$

oder

$f(d) = r$

einstellig $f : D \longrightarrow R$

n-stellig $f : D_1 \times \ldots \times D_n \longrightarrow R$

Funktion ⇒ Algorithmus ⇒ Programm ⇒ Ausführung

Funktion zu einer Funktion (Spezifikation) gibt es viele verschiedene Algorithmen

Algorithmus zu einem Algorithmus gibt es viele verschiedene Programme

Programm zu einem Programm gibt es viele verschiedene Prozessoren und Maschinenprogramme

---

zentral **Wie entwirft man Algorithmen ?**

☞ Viel schwieriger als die Programmierung
(Umsetzung: Algorithmus ⇒Programm)

☞ **Es gibt keinen Algorithmus zur Entwicklung von Algorithmen !!!**

aber Prinzipien, Techniken, Richtlinien

Berechenbarkeit Gibt es Funktionen (Prozesse) für die es keinen Algorithmus gibt?

↪ Wenn ja ⇒ nicht alles kann mit einem Computer berechnet werden !!!

• Kann man für eine Funktion (Prozeß) entscheiden, ob es hierfür einen Algorithmus gibt?

Komplexität

Fragen Welche und wieviele Betriebsmittel braucht ein Prozeß zur Ausführung eines Algorithmus?

Betriebsmittel

- Zeit
- Speicher
- Prozessoren
- Geräte

Vergleich Wann ist ein Algorithmus besser als ein anderer?

Komplexität eines Algorithmus. Die Komplexität eines Algorithmus ist der Aufwand an Betriebsmitteln bei der Berechnung.

Maschinenmodell mit den elementaren Operationen und Ablaufsteuerungen bildet die Basis für die Komplexitätsabschätzungen und den Vergleich von Algorithmen

- Turing–Maschine
- Registermaschine
- Parallelrechner

Komplexität einer Funktion = Komplexität des bestmöglichen Algorithmus, der diese Funktion berechnet.

Korrektheit

Frage Berechnet ein Algorithmus **immer** genau denselben Wert wie die zugehörige Funktion?

Antwort ist schwierig.

Korrektheitsbeweise für Programme sind sehr umfangreich und schwierig.

Hier ist noch viel Forschung notwendig.

Korrektheitsbeweise und -argumentationen können immer nur relativ zu einer Spezifikation geführt werden !!!

Behauptung: „*Dieses Programm ist richtig*“ setzt eine Spezifikation voraus.

2 Entwurf von Algorithmen

2.1 Algorithmen, Programme, Programmiersprachen

Algorithmus ist eine Verarbeitungsvorschrift, die aus genau bestimmten Elementaroperationen aufgebaut ist, und bei deren Interpretation die Reihenfolge der Ausführung der Elementaroperationen genau festgelegt ist.

in Daten–verarbeitung Die Elementaroperationen berechnen aus Eingabedaten (Parametern) neue Ausgabedaten (Resultate)

Daten Werte aus Wertebereichen (Typen)

- Wahrheitswerte (wahr, falsch)
- ganze Zahlen
- reelle Zahlen
- Namen
- Texte
- Tabellen
- ...

Folgerung Ein Algorithmus beschreibt eine Funktion

$f : E \longrightarrow A$

wobei E der Wertebereich der Eingabedaten ist, A der Wertebereich der Ausgabedaten.

Terminierung Ein Algorithmus terminiert, wenn seine Interpretation nach endlich vielen Schritten ein Ergebnis liefert

Beispiel

nicht terminierend: Sisyphos: mußte einen Felsen auf einen Berg wälzen, von dem der immer wieder herabrollte.

Problem Ist sichergestellt, daß ein Algorithmus für alle möglichen Eingaben terminiert.

$\hookrightarrow$

Korrektheit

(1) ein Algorithmus berechnet immer dengleichen Wert wie die zugehörige Funktion (*partielle* Korrektheit)

(2) der Algorithmus terminiert immer

Sprachen zur Formulierung von Algorithmen

Umgangssprache großes Vokabular, komplizierte Grammatik, mehrdeutig

- ↪ für Computer unverständlich
- ↪ für Menschen verständlich

Mathematische Formelsprache großes Vokabular, exakt, eindeutig, ausdruckskräftig, komplexe Operationen

- ↪ für Computer schon besser geeignet,
- ↪ durch die hohe Ausdruckskraft nicht immer automatisch in eine für Computer verständliche Sprache umsetzbar
- ↪ für Menschen nur mit math. Vorbildung verständlich

höhere Programmier–sprache exakt, eindeutig, einfache Elementaroperationen

- ↪ automatisch umsetzbar (compilierbar)
- ↪ länglicher als Formelsprache
- ↪ noch gut lesbar

Maschinensprache kleines Vokabular, schlecht lesbar, einfache Elementaroperationen

- ↪ gut auf einem Computer auszuführen
- ↪ schlecht zum Entwickeln und Programmieren

Beispiel

Fakultät

Umgangssprache Multipliziere für eine vorgegebene nichtnegative ganze Zahl x die Zahlen 1 bis x miteinander. Dies ist das Ergebnis. War $x = 0$, so soll das Ergebnis 1 sein.

---

viele Programmiersprachen

↪ viele verschiedene Notationen

☞ aber nur wenige Techniken und Prinzipien

Anforderungen an eine Programmiersprache

ausdruckskräftig einfache und kurze Darstellung

genau eindeutig

verständlich für den Entwickler

verständlich für den Computer (compilierbar)

Fehlerquellen bei der Umsetzung: Algorithmus $\Rightarrow$Programm minimieren (z.B. durch Einführung von Redundanz: Deklarationen)

lesbar beim Lesen des Programms sollte die Ausführung nachvollziehbar sein

angepaßt an die Aufgabenstellung

---

gegensätzliche Anforderungen

unterschiedliche Problemfelder

$\hookrightarrow$ viele verschiedene Programmiersprachen

2.2 Syntax und Semantik

Syntax eines Satzes (einer Sprache): grammatikalische Aufbau des Satzes

Semantik eines Satzes (einer Sprache): Interpretation des Satzes, Zuordnung einer Bedeutung zu dem Satz

$\hookrightarrow$ nur sehr wenigen syntaktisch richtigen Sätzen kann eine Bedeutung zugeordnet werden, kann sinnvoll interpretiert werden.

gleiche Bedeutung Es gibt verschiedene Sätze (in der gleichen oder in verschiedenen Sprachen) mit gleicher Bedeutung.

Kunstsprachen Es gibt nicht nur für die Programmierung Kunstsprachen (Schach,Chemie).

verschlüsselte Sprachen Interpretation nur mit zusätzlicher Information möglich (Kryptographie)

Mehrdeutigkeit vielen Wörtern sind mehrere Bedeutungen zugeordnet
viele Sätze können auf mehrere Arten interpretiert werden.

nicht in Programmier– oder Spezifikationssprachen erlaubt.

Prozessor interpretiert Algorithmus in einer Sprache

Sprache

Syntax der Prozessor muß die Sprache des Algorithmus lesen können

Semantik er muß zur Ausführung die Bedeutung der Sprache kennen

---

Syntax–analyse Syntax erkennen

Vokabular erkennen: Wörter, Abkürzungen, Sonderzeichen, Noten, ...

Grammatik erkennen:
Vernünftige Anordnung der Worte

Syntax = Vokabular + Grammatik

Syntax–analyse

- Vokabular erkennen
- Grammatik erkennen
- Syntaxfehler melden

Semantik Bedeutung einer Sprache

relativ die Semantik von Programmiersprachen ist viel einfacher als die Semantik natürlicher Sprache

absolut die Semantik von Programmiersprachen ist schwer zu definieren

syntaktisch korrekt **impliziert nicht: semantisch korrekt**

sinnlos	Farblose grüne Ideen schlafen wild
sinnvoll	Der Elefant aß die Erdnuß
sinnlos	Die Erdnuß aß den Elefanten
mehrdeutig	Der Mittelstürmer ist durchgebrochen

Semantik von

Wörtern: Zuordnung von Bedeutung zu den Wörtern

Satzteilen
Sätzen
Abschnitten
Büchern: wird zusammengesetzt aus der Semantik der Einzelteile

☞ Die Semantik von komplexen Objekten wird erklärt durch die Semantik der Bestandteile dieser Objekte

☞ **fundamentales Prinzip**

Beispiele

semantisch falsch	“Emil“ + 3
semantisch korrekt	$umfang\ :=\ pi\ *\ radius$
logisch falsch	Dies ist kein Algorithmus, der den Umfang eines Kreises aus seinem Radius berechnet.

↪	Korrektheit von Algorithmen:
	Vergleich der Algorithmen mit der zu berechnenden Funktion
↪	Vergleich mit der Spezifikation
↪	erst Spezifikation (was?) dann Algorithmus (wie?)

Algorithmen–entwurf schwierig

- Computer fehlt Intuition
- Präzision, Genauigkeit
- Vollständigkeit

Reduktion der Komplexität	**schrittweise Verfeinerung**
top–down design	teile und herrsche *divide and conquer*

Algorithmus	zur schrittweisen Verfeinerung
(1)	spezifiziere die Gesamtfunktion
(2)	wiederhole Schritt **(3)** so lange, bis ein ausreichender Detaillierungsgrad erreicht ist und die Aufgabe überschaubar ist
(3)	zerlege die Funktion in Teilfunktionen

3 Ausdrücke

3.1 Prädikate

elementare Bausteine einer Programmiersprache

Werte Zahlen, Wahrheitswerte $0, 1, 2, \ldots, \mathsf{true}, \mathsf{false}$

Konstante Namen für Werte

Variablen Programmvariablen
Namen für Objekte, die Werte speichern können $x, y, z, i, j, k, \ldots$

Ausdrücke bestehen aus Konstanten, Variablen und Operatoren $(+, -, *, /, =, \neq, >, <, \ldots)$

Prädikat Bedingung: ist ein Ausdruck der zu true oder false ausgewertet wird

Auswertung von Ausdrücken: Wenn allen Variablen in einem Ausdruck ein Wert zugewiesen ist, kann ein Ausdruck ausgewertet werden $x + y$
$a > b$

3.1.1 Prädikate: Boolesche Operatoren

Wahrheitswerte Wertebereich Bmit 2 Werten false und true, 0 oder 1, *falsch* oder *wahr*

$\mathsf{B} = \{\mathsf{false}, \mathsf{true}\}$

Boolesche Funktion ist eine Abbildung

$f : \mathsf{B} \times \ldots \times \mathsf{B} \longrightarrow \mathsf{B}$

Boolesche Operatoren $\Leftrightarrow, \Rightarrow, \wedge, \vee, \oplus, \neg$

Boolesche Ausdrücke Formeln aufgebaut aus

1. Booleschen Konstanten: true und false
2. Variablen
3. Funktionssymbole für Boolesche Funktionen mit Booleschen Ausdrücken als Teilausdrücke

Auswertung Boolescher Ausdrücke: Wird allen Variablen ein Wert zugeordnet, so kann ein Boolescher Ausdruck ausgewertet werden und ein Resultat true oder false zugeordnet werden.

$\hookrightarrow$ **Interpretation**

Gesetz, Satz oder Tautologie ist ein Boolescher Ausdruck, der bei jeder Belegung der Variablen zu true ausgewertet wird

Äquivalenz	$\equiv, =, \Leftrightarrow$
Wahrheitstabelle	$\Leftrightarrow : \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}$
	$\Leftrightarrow : \ \mathsf{false} \ \ \mathsf{false} \ \mapsto \mathsf{true}$
	$\Leftrightarrow : \ \mathsf{false} \ \ \mathsf{true} \ \mapsto \mathsf{false}$
	$\Leftrightarrow : \ \mathsf{true} \ \ \mathsf{false} \ \mapsto \mathsf{false}$
	$\Leftrightarrow : \ \mathsf{true} \ \ \mathsf{true} \ \mapsto \mathsf{true}$
Gesetze	
assoziativ	$((p \Leftrightarrow q) \Leftrightarrow r) \quad \Leftrightarrow \quad (p \Leftrightarrow (q \Leftrightarrow r))$
symmetrisch kommutativ	$(p \Leftrightarrow q) \quad \Leftrightarrow \quad (q \Leftrightarrow p)$
neutrales Element	$(p \Leftrightarrow \mathsf{true}) \quad \Leftrightarrow \quad p$
Beweise	durch Transformation zu true
	$p \Leftrightarrow \ldots \Leftrightarrow \ldots \Leftrightarrow \mathsf{true}$

Disjunktion	logisches ODER: or , $\vee$
Wahrheitstabelle	$\vee : \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}$
	$\vee : \ \mathsf{false} \ \ \mathsf{false} \ \mapsto \mathsf{false}$
	$\vee : \ \mathsf{false} \ \ \mathsf{true} \ \mapsto \mathsf{true}$
	$\vee : \ \mathsf{true} \ \ \mathsf{false} \ \mapsto \mathsf{true}$
	$\vee : \ \mathsf{true} \ \ \mathsf{true} \ \mapsto \mathsf{true}$
Konvention	$\vee$ bindet stärker als $\Leftrightarrow$
	$(p \vee q \Leftrightarrow r) \quad \Leftrightarrow \quad ((p \vee q) \Leftrightarrow r)$
Gesetze	
assoziativ	$(p \vee q) \vee r \quad \Leftrightarrow \quad p \vee (q \vee r)$
symmetrisch kommutativ	$p \vee q \quad \Leftrightarrow \quad q \vee p$
neutrales Element	$p \vee \mathsf{false} \quad \Leftrightarrow \quad p$
idempotent	$p \vee p \quad \Leftrightarrow \quad p$
distributiv	$p \vee (q \Leftrightarrow r) \quad \Leftrightarrow \quad (p \vee q \Leftrightarrow p \vee r)$

Konjunktion logisches UND: and , $\wedge$

Wahrheitstabelle

$$\begin{array}{llll} \multicolumn{4}{l}{\wedge : \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}} \\ \wedge : & \mathsf{false} & \mathsf{false} & \mapsto \mathsf{false} \\ \wedge : & \mathsf{false} & \mathsf{true} & \mapsto \mathsf{false} \\ \wedge : & \mathsf{true} & \mathsf{false} & \mapsto \mathsf{false} \\ \wedge : & \mathsf{true} & \mathsf{true} & \mapsto \mathsf{true} \end{array}$$

Konvention $\wedge$ bindet genauso stark wie $\vee$

Gesetze

assoziativ	$(p \wedge q) \wedge r \quad \Leftrightarrow \quad p \wedge (q \wedge r)$
symmetrisch kommutativ	$p \wedge q \quad \Leftrightarrow \quad q \wedge p$
neutrales Element	$p \wedge \mathsf{true} \quad \Leftrightarrow \quad p$
idempotent	$p \wedge p \quad \Leftrightarrow \quad p$
distributiv	$p \wedge (q \vee r) \quad \Leftrightarrow \quad (p \wedge q) \vee (p \wedge r)$
distributiv	$p \vee (q \wedge r) \quad \Leftrightarrow \quad (p \vee q) \wedge (p \vee r)$
Absorption	$p \vee (p \wedge q) \quad \Leftrightarrow \quad p$
Absorption	$p \wedge (p \vee q) \quad \Leftrightarrow \quad p$

Negation nicht, $\neg$

Wahrheitstabelle

$$\begin{array}{ll} \multicolumn{2}{l}{\neg : \mathsf{B} \longrightarrow \mathsf{B}} \\ \neg : \mathsf{false} & \mapsto \mathsf{true} \\ \neg : \mathsf{true} & \mapsto \mathsf{false} \end{array}$$

Konvention $\neg$ bindet stärker als alle anderen Operatoren

Gesetze

Gesetz vom ausgeschlossenen Dritten	$p \vee \neg p$
Widerspruch	$\neg(p \wedge \neg p)$
doppelte Verneinung	$\neg\neg p \quad \Leftrightarrow \quad p$
de Morgan	$\neg(p \vee q) \quad \Leftrightarrow \quad \neg p \wedge \neg q$
	$\neg(p \wedge q) \quad \Leftrightarrow \quad \neg p \vee \neg q$

Implikation $p \Rightarrow q$: aus p folgt q, wenn p, dann q

Wahrheitstabelle

$\Rightarrow : \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}$

$\Rightarrow :\ \mathsf{false}\ \ \mathsf{false}\ \mapsto \mathsf{true}$
$\Rightarrow :\ \mathsf{false}\ \ \mathsf{true}\ \mapsto \mathsf{true}$
$\Rightarrow :\ \mathsf{true}\ \ \mathsf{false}\ \mapsto \mathsf{false}$
$\Rightarrow :\ \mathsf{true}\ \ \mathsf{true}\ \mapsto \mathsf{true}$

Konvention $\Rightarrow$ bindet stärker als $\Leftrightarrow$
$\Rightarrow$ bindet schwächer als $\vee$ und $\wedge$

Gesetze

$p \Rightarrow q \quad \Leftrightarrow \quad \neg p \vee q$

$p \Rightarrow q \quad \Leftrightarrow \quad (p \vee q \Leftrightarrow q)$

modus ponens $p \wedge (p \Rightarrow q) \quad \Rightarrow \quad q$

modus tolens $\neg q \wedge (p \Rightarrow q) \quad \Rightarrow \quad \neg p$

Kontraposition $p \Rightarrow q \quad \Leftrightarrow \quad \neg q \Rightarrow \neg p$

transitiv
Kettenschluß $(p \Rightarrow q) \wedge (q \Rightarrow r) \quad \Rightarrow \quad (p \Rightarrow r)$

Beweise $p \Rightarrow \ldots \Rightarrow \ldots \Rightarrow q$

exklusives ODER entweder oder, xor , $\oplus$

Wahrheitstabelle

$\oplus : \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}$

$\oplus :\ \mathsf{false}\ \ \mathsf{false}\ \mapsto \mathsf{false}$
$\oplus :\ \mathsf{false}\ \ \mathsf{true}\ \mapsto \mathsf{true}$
$\oplus :\ \mathsf{true}\ \ \mathsf{false}\ \mapsto \mathsf{true}$
$\oplus :\ \mathsf{true}\ \ \mathsf{true}\ \mapsto \mathsf{false}$

Konvention bindet genauso wie $\wedge$ und $\vee$

Gesetze

assoziativ $(p \oplus q) \oplus r \quad \Leftrightarrow \quad p \oplus (q \oplus r)$

symmetrisch
kommutativ $p \oplus q \quad \Leftrightarrow \quad q \oplus p$

$p \oplus q \quad \Leftrightarrow \quad \neg(p \Leftrightarrow q)$

$p \oplus p \quad \Leftrightarrow \quad \mathsf{false}$

Boolesche Algebra Eine Menge M mit zwei Verknüpfungen $+$ und $\cdot$ heißt Boolesche Algebra, wenn für alle $x, y, z \in M$ gilt:

Kommutativität $x{\cdot}y = y{\cdot}x$

$x{+}y = y{+}x$

Assoziativität $x{\cdot}(y{\cdot}z) = (x{\cdot}y){\cdot}z$

$x{+}(y{+}z) = (x{+}y){+}z$

Absorption $x{\cdot}(x{+}y) = x$

$x{+}(x{\cdot}y) = x$

Distributivität $x{\cdot}(y{+}z) = (x{\cdot}y){+}(x{\cdot}z)$

$x{+}(y{\cdot}z) = (x{+}y){\cdot}(x{+}z)$

neutrales Element es gibt ein Element $0 \in M$:
$0{\cdot}x = 0$ und $0{+}x = x$

es gibt ein Element $1 \in M$:
$1{\cdot}x = x$ und $1{+}x = 1$

Komplement zu jedem Element $x \in M$ gibt es ein $y \in M$ mit $x{\cdot}y = 0$ und $x{+}y = 1$

$\hookrightarrow$ B mit $\wedge,\vee,\mathsf{true},\mathsf{false}$ für $\cdot, +, 1, 0$ bilden eine Boolesche Algebra

3.1.2 Aussagenlogik

logisches Schließen heißt, aus

Prämissen Voraussetzungen die

Konklusion die Schlußfolgerung herzuleiten

Aussage eine Behauptung, die wahr oder falsch sein kann

Formalisierung durch Umformen der Umgangssprache in logische Formeln zusammengesetzt aus elementaren Aussagen und booleschen Operatoren

Abstraktion vom konkreten Problem durch Ersetzen der Elementaraussagen durch Buchstaben

(1) Wenn k schließlich x gleicht, bricht der Algorithmus ab.
k ist schließlich gleich x.
Darum wird der Algorithmus beendet.

(2) Wenn das Buch in der Literaturliste steht, ist ein Exemplar in der Bibliothek.
Das Buch steht in der Literaturliste.
Daher ist ein Exemplar in der Bibliothek.

Schema Wenn A dann B
A
Also B

modus ponens $\underbrace{a}_{Prämisse} \wedge \underbrace{(a \Rightarrow b)}_{Argumentation} \Rightarrow \underbrace{b}_{Konklusion}$

Beispiel Aussagen über Supermann

Wenn Supermann das Böse verhindern kann und will, dann wird er es tun. Wenn Supermann das Böse nicht verhindern kann, dann ist er machtlos; wenn er es nicht verhindern will, dann ist er böswillig. Supermann verhindert das Böse nicht. Wenn Supermann existiert, ist er weder machtlos noch böswillig. Darum existiert Supermann nicht.

Prämissen

(1) Wenn Supermann das Böse verhindern kann und will, dann wird er es tun.

(2) Wenn Supermann das Böse nicht verhindern kann, dann ist er machtlos.

(3) Wenn Supermann es nicht verhindern will, dann ist er böswillig.

(4) Supermann verhindert nicht das Böse.

(5) Wenn Supermann existiert, ist er weder machtlos noch böswillig.

Konklusion

$\hookrightarrow$ Supermann existiert nicht.

logisches Puzzle

Ritter & Knechte Auf einer einsamen Insel gibt es zwei Arten von Bewohner

Ritter erzählen immer die Wahrheit

Knechte lügen immer

Jeder Bewohner ist entweder Ritter oder Knecht

Frage an eine beliebige Person a: *Sind sie ein Ritter?*

Antwort *Wenn ich ein Ritter bin, freß ich einen Besen*

Behauptung a muß einen Besen fressen.

zu beweisende Aussage a sagt, wenn ich ein Ritter bin, dann freß ich einen Besen.

2.Problem Es gib 2 Personen a und b.
a sagt: *Wenn B ein Ritter ist, dann bin ich ein Knecht*

Frage Was sind a und b ?

Logelei aus Knusiland

Abianer sagen immer die Wahrheit

Bbianer lügen immer

Cbianer sagen abwechselnd die Wahrheit und lügen, man weis aber nicht, in welchem Zustand sie sich gerade befinden

1.Frage an einen Blonden: *Zu welchem Stamm gehören Sie?*
Ich bin Abianer.

1.Frage an einen Schwarzhaarigen: *Zu welchem Stamm gehören Sie?*
Ich bin Bbianer.

2.Frage an den Schwarzhaarigen: *Hat der Blonde die Wahrheit gesagt?*
Ja.

1.Frage an einen Rothaarigen: *Zu welchem Stamm gehören Sie?*
Ich bin Abianer.

2.Frage an den Rothaarigen: *Zu welchem Stamm gehört der Blonde?*
Der Blonde ist Cbianer

Aufgabe Wer ist von welchem Stamm?

3.1.3 Prädikate mit arithmetischen Ausdrücken

Zahlen

N_0 unsigned int , cardinal: natürliche Zahlen ab 0

N_1 natürliche Zahlen ab 1

Z Intg, integer: ganze Zahlen

R real, float: reelle Zahlen

Operationen auf natürlichen Zahlen

elementare Prädikate

$$= : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{B}$$
$$\neq : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{B}$$

elementare Operationen

$$+1 : \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad \textit{succ}$$
$$-1 : \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad \textit{pred}$$

Arithmetik

$$+ : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad (=,+1,-1)$$
$$- : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad (=,-1)$$
$$\cdot : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad (+,=,-1)$$
$$\mathsf{div} : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad (-,\geq,=,+1)$$
$$\mathsf{mod} : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{N}_0 \qquad (\mathsf{div},-,=)$$

$+,\cdot$ totale Funktionen
$-$,div,mod partielle Funktionen

Vergleiche

$$\geq : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{B} \qquad (=,-1)$$
$$> : \mathsf{N}_0 \times \mathsf{N}_0 \longrightarrow \mathsf{B} \qquad (=,-1)$$

$\geq,>$ totale Funktionen

Prädikate

Aussagenlogik Formeln mit Booleschen Variablen (absolute Wahrheiten)

Prädikatenlogik Formeln die aus Relationen ($=, \geq, >, \ldots$) aufgebaut sind und mit logischen Operatoren verknüpft sind.

Prädikat

- einfache Relationen $=, \geq, >, \ldots$
- aus Prädikaten und logischen Operatoren zusammengesetzte Formeln

$\hookrightarrow$ Variablen in den Formeln sind nicht mehr ausschließlich aus B sondern aus beliebigen Bereichen (N_0,N_1,Z,R,...)

Interpretation

- Variablen werden Werte zugeordnet
- Formeln werden ausgewertet

erfüllbare Prädikate: Formeln, für die es eine Variablenbelegung gibt (eine Zuordnung von Werten zu Variablen), so daß die Formel zu `true` ausgewertet wird

unerfüllbare Prädikate: Formeln, die bei jeder Belegung zu `false` ausgewertet werden

gültige Prädikate (Sätze): Formeln, die bei jeder Belegung zu `true` ausgewertet werden.

Ungleichungen rechnen mit Ungleichungen
i, j, k seien ganze Zahlen

gültige Prädikate

transitiv	$i \leq j \wedge j \leq k \quad \Rightarrow \quad i \leq k$
reflexiv	$i \leq i$
antisymmetrisch	$i \leq j \wedge j \leq i \quad \Rightarrow \quad i = j$
	$i \leq j \wedge 0 \leq k \quad \Rightarrow \quad i * k \leq j * k$
transitiv	$i < j \wedge j < k \quad \Rightarrow \quad i < k$
	$\neg(i < i)$
	$\neg(i < j \wedge j < i)$
	$i < j \quad \Rightarrow \quad i + k < j + k$
	$i < j \wedge 0 \leq k \quad \Rightarrow \quad i * k \leq j * k$
	$i < j \wedge 0 < k \quad \Rightarrow \quad i * k < j * k$
	$i < j \quad \Rightarrow \quad i + 1 \leq j$

3.1.4 Prädikate mit Quantoren

Prädikate über die Elemente einer Menge

$\hookrightarrow$ *„jedes Element der Menge hat die Eigenschaft, daß … “*

$\hookrightarrow$ *„es gibt Elemente in der Menge mit der Eigenschaft, daß … “*

Quantoren Operationen, die auf Mengen definiert sind

All-Quantor
$\forall$ um zu bestimmen, daß jedes Element einer Menge eine bestimmte Eigenschaft besitzt

$\forall\, i \in M \bullet P(i)$

für alle i aus der Menge M gilt das Prädikat $P(i)$

$P(i_1) \land P(i_2) \land \ldots \land P(i_j) \land \ldots$

Existenz-Quantor
$\exists$ um zu bestimmen, daß mindestens ein Element einer Menge eine bestimmte Eigenschaft besitzt

$\exists\, i \in M \bullet P(i)$

es gibt ein i in der Menge M, für das das Prädikat $P(i)$ gilt

$P(i_1) \lor P(i_2) \lor \ldots \lor P(i_j) \lor \ldots$

gebundene Variablen i ist eine an den Quantor gebundene Variable, sie kann Werte aus ihrem Definitionsbereich M annehmen

freie Variablen alle ungebundenen Variablen

Beispiel

Aussage *„alle Aldi-PCs sind schlecht gebaut“*

Grundmenge PCs

elementare Prädikate $vonAldi(x)$

$gut(x)$

Formel mit All–Quantor

$\forall\, pc \in PCs \bullet vonAldi(pc) \Rightarrow \neg gut(pc)$

Aussage *„es gibt eine gerade Primzahl“*

Grundmenge N_0

elementare Prädikate $gerade(x)$

$prim(x)$

Formel mit Existenz–Quantor

$\exists\, n \in \mathsf{N}_0 \bullet gerade(n) \land prim(n)$

Grundmenge $i \in S$

Intervall häufiger Sonderfall von Mengen:
Intervall aus den ganzen Zahlen

Notation $a \leq i < b$

alternative Notation für Formeln mit Quantoren

$\forall\, a \leq i < b \bullet P(i)$	halb offenes Intervall
$\forall\, a \leq i \leq b \bullet P(i)$	abgeschlossenes Intervall
$\forall\, a \leq i \bullet P(i)$	unbeschränktes Intervall

Felder f : array $[0 \mathinner{..} n-1]$ of *Element*

f ist eine Variable zum Speichern einer ganzen Menge von Werten

Referenzierung $f[i], f(i), f_i$ Zugriff auf i-te Element in f

Indexbereich ein Intervall (hier $0 \leq i < n$)

Beispiele für Prädikate mit $\forall$-Quantoren

gegeben zwei Felder a und b
vom Typ array $[0 \mathinner{..} n-1]$ of Z

Aufgabe Wie sehen die zugehörigen Prädikate aus?

(1) a ist eine exakte Kopie von b, alle Elemente sind gleich

(2) Für alle i ist das i-te Element von a kleiner als das i-te Element von b

(3) Jedes Element von a ist kleiner als jedes Element von b

(4) Wenn die Elemente von a in aufsteigender Reihenfolge sortiert sind, so auch die Elemente von b

(5) Alle Elemente von a sind untereinander verschieden

(6) Jedes Element von a ist von jedem Element von b verschieden

Beispiele für Prädikate mit $\exists$- und $\forall$-Quantoren

gegeben zwei Felder a und b
vom Typ $\mathsf{array}\ [0\,..\,n{-}1]\ \mathsf{of}\ \mathsf{Z}$

Aufgabe Wie sehen die zugehörigen Prädikate aus?

(1) Einige Elemente von a sind ungleich 0

(2) das Feld a ist nicht in aufsteigender Reihenfolge sortiert

(3) mindestens ein Element von a ist größer als alle Elemente von b

(4) Jedes Element von b ist eine Kopie eines Elements von a

(5) b enthält alle Zahlen von 0 bis $n{-}1$ (eine Permutation aller Zahlen von 0 bis $n{-}1$)

(6) b zeigt die numerische Ordnung von a an, d.h. $b[0]$ ist der Index auf das kleinste Element von a, $b[1]$ der Index auf das zweitkeinste Element usw.

(7) Wenn alle Elemente in a paarweise verschieden sind, dann enthält b die Elemente von a in sortierter Reihenfolge

(8) b enthält die Elemente von a in sortierter Reihenfolge

Eigenschaften von Quantoren

Gesetze

(1) $\forall\, i \in \{\,\} \bullet P(i) \quad\Leftrightarrow\quad \mathsf{true}$

(2) $\exists\, i \in \{\,\} \bullet P(i) \quad\Leftrightarrow\quad \mathsf{false}$

(3) $\exists\, i \in M \bullet \neg P(i) \quad\Leftrightarrow\quad \neg\forall\, i \in M \bullet P(i)$

$\forall\, i \in M \bullet \neg P(i) \quad\Leftrightarrow\quad \neg\exists\, i \in M \bullet P(i)$

Verallgemeinerung von de Morgan

(4) $\forall\, i \in M \bullet P(i) \wedge j \in M \quad\Rightarrow\quad P(j)$

(5) $j \in M \wedge P(j) \quad\Rightarrow\quad \exists\, i \in M \bullet P(i)$

Intervalle

(1a) $\forall\, a \le i < a \bullet P(i) \quad\Leftrightarrow\quad \mathsf{true}$

(2a) $\exists\, a \le i < a \bullet P(i) \quad\Leftrightarrow\quad \mathsf{false}$

(3a) $\exists\, a \le i < b \bullet \neg P(i) \quad\Leftrightarrow\quad \neg\forall\, a \le i < b \bullet P(i)$

$\forall\, a \le i < b \bullet \neg P(i) \quad\Leftrightarrow\quad \neg\exists\, a \le i < b \bullet P(i)$

(4a) $\forall\, a \le i < b \bullet P(i) \wedge a \le j < b \quad\Rightarrow\quad P(j)$

(5a) $a \le j < b \wedge P(j) \quad\Rightarrow\quad \exists\, a \le i < b \bullet P(i)$

(6a) $\forall\, a \le i < b \bullet P(i) \wedge \forall\, b \le i < c \bullet P(i)$
$\quad\Leftrightarrow\quad \forall\, a \le i < c \bullet P(i)$

Negation von Prädikaten mit Quantoren

Aussage *„alle Aldi-PCs sind schlecht gebaut“*

Frage Welche der folgenden Aussagen ist die Negation dieser Aussage

(1) Alle nicht-Aldi-PCs sind gut gebaut

(2) Alle Aldi-PCs sind gut gebaut

(3) Einige Aldi-PCs sind gut gebaut

(4) Einige Aldi-PCs sind schlecht gebaut

(5) Einige nicht-Aldi-PCs sind schlecht gebaut

(6) Einige nicht-Aldi-PCs sind gut gebaut

Quantoren Verallgemeinerung von 2–stelligen Operationen, die kommutativ (symmetrisch) und assoziativ sind und für die es ein neutrales Element gibt.

nicht nur für logische Operatoren möglich

$\forall$	aus	$\wedge$	und	true
$\exists$	aus	$\vee$	und	false
$\sum$	aus	$+$	und	0
$\prod$	aus	$\cdot$	und	1
MAX	aus	max	und	$-\infty$
MIN	aus	min	und	∞

4 Zuweisungen, Verzweigungen und Schleifen

4.1 Spezifikationen

Notation

$\hookrightarrow$ $\{\ \underbrace{V}_{\substack{Vor- \\ bedingung}}\ \}\ \underbrace{S}_{\substack{An- \\ weisung}}\ \{\ \underbrace{P}_{\substack{Nach- \\ bedingung}}\ \}$

$\hookrightarrow$ Notation zur Begründung (Erklärung, Dokumentation), warum ein Programm(–Stück) korrekt arbeitet

$\{\ldots\}$ ist ein Kommentar, hat bei der Programmausführung keinen Effekt enthält ein Prädikat (Bedingung, Ausdruck, der zu `true` oder `false` ausgewertet werden kann)

V **und** P sind Prädikate

S ist ein Programm

Zustandsbeschreibung der Zustand eines Programms (Belegung der Variablen) kann mit solchen Prädikaten beschrieben werden

V spezifiziert den Definitionsbereich der zu berechnenden Funktion

P spezifiziert die zu berechnende Funktion

$\hookrightarrow$ V und P werden Zusicherungen *(assertions)* genannt

Beispiele Programmspezifikation

ein Programm soll ...

(1) ... eine Variable x auf den Wert 42 setzen, es soll immer funktionieren

(2a) ... die Summe zweier Zahlen x und y in einer Variablen s speichern

(2b) ... das Maximum zweier Zahlen x und y in einer Variablen r speichern

(3) ... die Werte zweier Variablen vertauschen

(4) ... 2 natürliche Zahlen x und y ganzzahlig mit Rest teilen und in den Variablen q und r Resultat und Rest speichern

(5) ... die 1. n natürlichen Zahlen ($n \geq 0$) in einer Variablen s aufsummieren

(6) ... für eine natürliche Zahl n den größten Teiler $r < n$ bestimmen

Korrektheit	von Programmen
relativ	zu einer Spezifikation
$\hookrightarrow$	ein Programm(-stück) ist korrekt,
(1)	wenn es in einem Anfangszustand gestartet wird, in dem die Vorbedingung V gilt, d.h die Variablenbelegung gehört zum Definitionsbereich,
(2)	wenn es terminiert
(3)	wenn im Endzustand die Nachbedingung P gilt
(1+3)	partielle Korrektheit
$\hookrightarrow$	Vorbedingung darf verstärkt werden $\hookrightarrow$ Definitionsbereich wird eingeschränkt
$\hookrightarrow$	Nachbedingung darf abgeschwächt werden $\hookrightarrow$ *„es wird weniger berechnet“*
Beweisregel	Stärkung einer Vorbedingung und Schwächung einer Nachbedingung
falls	
(1)	$V \Rightarrow V_1$
(2)	$\{V_1\}\ S\ \{P_1\}$
(3)	$P_1 \Rightarrow P$
dann gilt	$\{V\}\ S\ \{P\}$

Vorgehen	Programmentwurf
(1)	Problem spezifizieren
(1a)	Mit welchen Variablen soll gearbeitet werden $\Rightarrow$ Deklarationen
(1b)	Was soll berechnet werden $\Rightarrow P$
(1c)	Wann soll das Programm funktionieren $\Rightarrow V$
(2)	hieraus ein Programm konstruieren
Variablen	Die in der Spezifikation und im Programm benutzten Variablen und deren Typ (Wertebereich) werden mit Hilfe von Deklarationen festgelegt. Alle übrigen Namen sind Konstanten
$\hookrightarrow$	var x : N_0
$\hookrightarrow$	var b : B
$\hookrightarrow$	var f : array $[0\,..\,n{-}1]$ of N_0

Spezifikationen spezifiziere ein Programm, das …

(1a) … für ein $n \geq 0$ in x die größte 2-er Potenz $\leq n$ berechnet

(2a) … für ein Feld f in einer Booleschen Variablen *sortiert* berechnet, ob f sortiert ist.

(2b) … für eine Variable w testet, ob w in dem Feld f enthalten ist. Das Resultat soll in einer Variablen *enthalten* stehen

(2c) … ein Feld f_1 in ein Feld f_2 sortiert, wobei angenommen wird, daß alle Elemente in f_1 verschieden sind

(3a) … einen Index i für das größte Element in einem Feld f bestimmt

(3b) … den größten Wert in einem Feld f bestimmt

(3c) … den 2.-größten Wert in einem Feld f berechnet.

4.2 Anweisungen: Syntax und Semantik

4.2.1 Zuweisungen

Zuweisung bestehen aus einer Variablen und einem Ausdruck

$Variable \ := \ Ausdruck$ $\qquad x := 1$

$b := \mathsf{true}$

Zustand Programmzustand:
Belegung der Programmvariablen mit einem Wert zu einem bestimmten Zeitpunkt der Programmausführung

Bedeutung Semantik

informell der *Ausdruck* wird ausgewertet und der *Variablen* zugewiesen,
danach speichert die Variable den berechneten Wert

$\hookrightarrow$ Zustandstransformation, Zustandsänderung

Bedeutung Semantik

formal durch eine Regel, die beschreibt, wie aus einer Nachbedingung und einer Zuweisungsanweisung die (schwächste) Vorbedingung berechnet werden kann

Notation P sei ein Ausdruck, in dem x vorkommt, A sei irgendein beliebiger Ausdruck

$P[x \leftarrow A]$ ist der Ausdruck, in dem alle Vorkommen von x durch A ersetzt worden sind (und möglicherweise durch zusätzliche Klammern der syntaktische Aufbau von P erhalten bleibt).

Beweisregel für Zuweisungen

$$\{P[x \leftarrow A]\} \;\; x := A \;\; \{P\}$$

☞ Argumentation läuft rückwärts

☞ Prädikate dürfen in gleichwertige einfachere Prädikate umgeformt werden

Erweiterung mehrfache Zuweisungen

Syntax $x_1, \ldots, x_n := A_1, \ldots, A_n$

Notation

$P[x_1 \leftarrow A_1, \ldots, x_n \leftarrow A_n]$

ist der Ausdruck, in dem alle Vorkommen von x_i durch A_i (gleichzeitig) ersetzt worden sind

Semantik

Beweisregel für mehrfache Zuweisungen

$$\{P[x_1 \leftarrow A_1, \ldots, x_n \leftarrow A_n]\} \;\; x_1, \ldots, x_n := A_1, \ldots, A_n \;\; \{P\}$$

$\hookrightarrow$ für Spezifikation häufig sehr nützlich

4.2.2 Anweisungsfolgen

zusammengesetzte Bausteine der Programmiersprache

Anweisungen können aufgebaut werden aus

- ↪ Zuweisungen
- ↪ Anweisungsfolgen
- ↪ bedingte Anweisungen
- ↪ Schleifen–Anweisungen

Anweisungsfolge Zwei (oder mehrere) Anweisungen können zu eine Anweisungsfolge zusammengesetzt werden

Syntax $Anweisung_1$; $Anweisung_2$

$$x := 3; y := 5$$

Semantik anschaulich:
Nacheinanderausführen der einzelnen Anweisungen in der angegebenen Reihenfolge

formal

Beweisregel für Anweisungsfolgen

falls

(1) $\{V\}\ \ S_1\ \ \{P_1\}$

(2) $\{P_1\}\ \ S_2\ \ \{P\}$

dann gilt $\{V\}\ \ S_1; S_2\ \ \{P\}$

4.2.3 bedingte Anweisungen

if –Anweisung eine Bedingung (Prädikat, Ausdruck, der zu true oder false ausgewertet wird) und zwei Anweisungen können zu einer bedingten Anweisung zusammengesetzt werden

Syntax

```
if Bedingung
      then Anweisung1
      else Anweisung2
end if
```

Semantik informell

(1) die *Bedingung* wird ausgewertet

(2a) ist das Resultat true,
wird die $Anweisung_1$ ausgeführt

(2b) ist das Resultat false,
wird die $Anweisung_2$ ausgeführt anschließend wird

Variante eine *Bedingung* und eine *Anweisung* können zu einer bedingten Anweisung zusammengesetzt werden

Syntax

```
if Bedingung
      then Anweisung
end if
```

Semantik formal

Beweisregel für bedingte Anweisungen

falls

(1) $\{V_1\}\ \ S_1\ \ \{P\}$

(2) $\{V_2\}\ \ S_2\ \ \{P\}$

dann gilt $\{(V_1 \wedge B) \vee (V_2 \wedge \neg B)\}$ if B then S_1 else S_2 end if $\{P\}$

Beweisregel für bedingte Anweisungen nur mit then Zweig

falls

(1) $\{V\}\ \ S\ \ \{P\}$

dann gilt $\{(V \wedge B) \vee (P \wedge \neg B)\}$ if B then S end if $\{P\}$

Mehrfach –Auswahl Mehrfach–Verzweigung

$\hookrightarrow$ in Programmiersprachen gibt es häufig noch eine Mehrwegverzweigung (case -Anweisung).

Syntax

```
case Ausdruck of
    Wert_1 : Anweisung_1
        ⋮
    Wert_n : Anweisung_n
    else  : Anweisung_0
end case
```

Bedeutung erklärt mit bedingter Anweisung

```
Wert := Ausdruck
if Wert=Wert_1
    then Anweisung_1
else
    ⋮
if Wert=Wert_n
    then Anweisung_n
else
        Anweisung_0
end if ... end if
```

4.2.4 Schleifenanweisungen

Wiederholung

while **–Schleife** — aus einer Bedingung und einer Anweisung kann eine while –Schleife aufgebaut werden

Syntax

```
while Bedingung do
      Anweisung
end while
```

Bedeutung — informell

(1) — die *Bedingung* wird ausgewertet

(2a) — ist das Resultat false,
so wird die Ausführung des Programms hinter der Schleife fortgesetzt

(2b) — ist das Resultat true,
so wird die Anweisung (der Schleifenrumpf) ausgeführt und anschließend die Schleife erneut ausgeführt.

↪ der Schleifenrumpf wird $0, 1, 2, \ldots$ mal ausgeführt.

↪ in Programmiersprachen gibt es häufig noch weitere Schleifenarten (repeat –, for –Schleifen). Diese können alle auf die while –Schleife zurückgeführt werden.

Konstruktion — von while –Schleifen

Initialisierungs-anweisungen — für Variable, die in der Schleife verwendet werden, häufig

- ↪ eine Laufvariable
- ↪ eine Variable für das Resultat

Abbruchkriterium — eine Bedingung, die bestimmt, wie lange der Schleifenrumpf wiederholt ausgeführt wird.

Diese Bedingung muß mindestens eine Variable enthalten, die im Schleifenrumpf verändert wird, häufig

- ↪ Test der Laufvariablen gegen einen Endewert

Rumpf — enthält Anweisungen zur Veränderung von Variablen, häufig

- ↪ Inkrementieren der Laufvariable
- ↪ Akkumulieren des Resultats in der zugehörigen Variablen

Endlosschleife — Wird in einem Schleifenrumpf keine Variable verändert, die im Abbruchkriterium vorkommt, so erhält man eine Endlosschleife

Konstruktion — aus Spezifikationen mit Quantoren

↪ an Quantoren gebundene Variablen werden zu Laufvariablen

↪ aus Bereichsgrenzen werden Initialisierung und Abbruchkriterium abgeleitet

Beweisregel	für `while` –Schleifen
falls	
(1)	$\{I \wedge B\}\ \ S\ \ \{I\}$
dann gilt	$\{I\}$ while B do S end while $\{I \wedge \neg B\}$
Invariante	I ist die Schleifeninvariante, d.h. eine Eigenschaft, die $\hookrightarrow$ vor der Ausführung der Schleife $\hookrightarrow$ vor jeder Ausführung des Rumpfes $\hookrightarrow$ nach jeder Ausführung des Rumpfes $\hookrightarrow$ nach der Ausführung der Schleife gilt
Konstruieren	der Invarianten aus der Spezifikation
Faustregel	Invariante ist eine Verallgemeinerung der Anfangs- und Endesituation
	hier wird noch nichts über die Terminierung ausgesagt
$\hookrightarrow$	Regel noch unvollständig

Terminierung	zusätzliche Bedingungen
falls	
Invariante	$\{I \wedge B\}\ \ S\ \ \{I\}$
Fortschritt	$\{I \wedge B \wedge t > T\}\ \ S\ \ \{t = T\}$
Beschränkung	$I \wedge t \leq 0 \Rightarrow \neg B$
dann	gilt die Nachbedingung $I \wedge \neg B$ und die Schleife terminiert: $\{I\}$ while B do S end while $\{I \wedge \neg B\}$
Variante	t ist eine ganzzahlige Funktion T ist eine Konstante
$\hookrightarrow$	wenn I erfüllt ist vor der Ausführung von S, dann auch nach der Ausführung von S
$\hookrightarrow$	$\{t > T\}\ \ S\ \ \{t = T\}$ beschreibt den Fortschritt der Schleife, da t mindestens um 1 verkleinert wird
$\hookrightarrow$	$I \wedge t \leq 0 \Rightarrow \neg B$: wird $t \leq 0$ so terminiert die Schleife
$\hookrightarrow$	Anfangswert von t liefert obere Grenze für die Anzahl der Schleifendurchläufe
Beschränkung	ist gleichwertig mit der Bedingung $I \wedge B \Rightarrow t > 0$

Rezept zur Konstruktion von Schleifen

Ziel $\{V\}\ \ Prog\ \ \{P\}$

Regel für Schleifen:

$\{I\}$ while B do S end while $\{I \wedge \neg B\}$

Schritt 1 Bedingungen I und B so wählen,daß gilt

$I \wedge \neg B \Rightarrow P$

$\hookrightarrow$ $\{I\}$ while B do S_1 end while $\{P\}$

Schritt 2 Initialisierungsanweisungen
ein Programmstück S_0 konstruieren, so daß gilt:

$\{V\}\ \ S_0\ \ \{I\}$

Schritt 3 zusammensetzen:

$\{V\}$ S_0; while B do S_1end while $\{P\}$

Schritt 4 den Schleifenrumpf S_1 so entwickeln, daß gilt:

$\{I \wedge B\}\ \ S_1\ \ \{I\}$

Schritt 5 Terminierung sichern mit einer Varianten t:

$\{I \wedge B \wedge t > T\}\ \ S\ \ \{t = T\}$

Schritt 6 alles zusammensetzen

repeat –**Schleife** aus einer Bedingung und einer Anweisung kann eine repeat –Schleife aufgebaut werden

Syntax

```
repeat
    Anweisung
until Bedingung
```

Semantik mit Hilfe der while –Schleife

```
Anweisung;
while ¬Bedingung do
    Anweisung
end while
```

Semantik informell

(1) die Anweisung (der Schleifenrumpf) wird ausgeführt

(2) die Bedingung wird ausgewertet

(3a) ist das Resultat true, so wird die Ausführung des Programms hinter der Schleife fortgesetzt

(3b) ist das Resultat false, so wird die Ausführung der Schleife wiederholt

der Schleifenrumpf wird $1, 2, \ldots$ mal ausgeführt.

nicht für Schleifen zu gebrauchen, deren Rumpf möglicherweise nicht ausgeführt wird.

while –Schleifen allgemeiner

repeat –Schleife **nicht** effizienter als while –Schleife

for –Schleife Zählschleife

Syntax

```
for Variable := Ausdruck_1 to Ausdruck_2 do
        Anweisung
end for
```

Variable ist die Laufvariable

$Ausdruck_1$ ist der Startwert

$Ausdruck_2$ ist der Endwert

Anweisung ist der Schleifenrumpf

Semantik Bedeutung: erklärt mit einer while –Schleife

```
Variable := Ausdruck_1;
Endewert := Ausdruck_2;
while Variable ≤ Endewert do
        Anweisung;
        Variable := Variable +1
end while
```

☞ while –Schleifen allgemeiner

4.2.5 Syntaxdefinition mit kontextfreier Grammatik

kontextfreie Grammatik Regeln zum syntaktischen Aufbau eines Programms

```
Anweisung           ::= Zuweisung
                     |  Anweisungsfolge
                     |  bedingteAnweisung
                     |  SchleifenAnweisung
Zuweisung           ::= einfacheVariable :=  Ausdruck
Anweisungsfolge     ::= Anweisung ; Anweisung
bedingteAnweisung   ::= if Ausdruck
                          then Anweisung
                          else Anweisung
                        end if
                     |  if Ausdruck
                          then Anweisung
                        end if
                     |  ...
SchleifenAnweisung  ::= while Ausdruck do
                          Anweisung
                        end while
                     |  ...
```

5 Funktionen und Prozeduren

5.1 Modularität

Einzelteile	der Algorithmen unabhängig vom Algorithmus selbst
$\hookrightarrow$	unabhängig entwickelbar
$\hookrightarrow$	an verschiedenen Stellen einsetzbar im selben Algorithmus in verschiedenen Algorithmen
$\hookrightarrow$	Teile betrachten wie neue Elementaroperationen

---

Beispiel

Algorithmus

quadratsumme(x,y)	quadrat(x) + quadrat(y)
Benutzung	quadratsumme(3,4)
Name	des Algorithmus: **quadratsumme**
formale Parameter	**x** und **y**
Aufruf	quadratsumme(3,4)
aktuelle Parameter	**3** und **4**
Funktionen	sind Algorithmen, die einen Wert berechnen
Prozeduren	sind Algorithmen, die einen Zustand (globale Variablen) verändern

Syntax

Funktionen Algorithmen zur Berechnung eines Wertes

FunktionsDefinition	::=	*FunktionsKopf FunktionsRumpf*
FunktionsKopf	::=	*FktName* **(** *formaleParamListe* **)**
		: *Typ*
formaleParamListe	::=	*formalerParameter*
	|	*formalerParameter* **;** *formaleParamListe*
formalerParameter	::=	*Variable* **:** *Typ*
FunktionsRumpf	::=	*Ausdruck*
Ausdruck	::=	...
	|	*FunktionsAufruf*
FunktionsAufruf	::=	*FktName* **(** *aktuelleParamListe* **)**
aktuelleParamListe	::=	*Ausdruck*
	|	*Ausdruck* **,** *aktuelleParamListe*

Semantik eines Aufrufs

(1) die aktuellen Parameter des Aufrufs werden berechnet

(2) Der Text der Funktionsdefinition wird kopiert und alle formalen Parameter in diesem Text werden durch die Werte der aktuellen Parameter ersetzt

(3) danach wird dieser neue Ausdruck ausgewertet

(4) nach der Abarbeitung der Funktion wird der Funktionsaufruf durch das berechnete Resultat ersetzt

$\hookrightarrow$ in Funktionsrümpfen können wieder Funktionsaufrufe vorkommen

Rekursion in dem Funktionsrumpf kann wieder die Funktion selbst aufgerufen werden

Syntaxerweiterung für Ausdrücke

Ausdrucksfolgen

$$\begin{array}{lcl} \mathit{Ausdruck} & ::= & \ldots \mid \mathit{Ausdrucksfolge} \\ \mathit{Ausdrucksfolge} & ::= & \mathit{Anweisung}\ ;\ \mathit{Ausdruck} \end{array}$$

Bedeutung (Interpretation, Ausführung) einer Ausdrucksfolge

(1) die *Anweisung*(-en) werden ausgeführt

(2) der *Ausdruck* wird ausgewertet und ergibt das Resultat der *Ausdrucksfolge*

Syntaxerweiterung um lokale Variablen und Blöcke

Idee die Lebensdauer und die Sichtbarkeit von (Hilfs–)Variablen einschränken auf den Teil des Algorithmus, in dem sie verwendet werden (Modularität)

Block syntaktische Einheit

Anweisung	::=	... | *Block*
Block	::=	begin *Deklaration* ; *Anweisung* end
Deklaration	::=	var *Variablenliste*
Variablenliste	::=	*Variablendekl*
	|	*Variablendekl* ; *Variablenliste*
Variablendekl	::=	*Variablen* : *Typ*
Variablen	::=	*Variable*
	|	*Variable* , *Variablen*

↪ entsprechende Syntaxregeln auch für *Ausdrücke*

Bedeutung

(1) beim Eintritt in den Block werden die (lokalen) Variablen aus der Deklaration erzeugt (aber nicht automatisch mit einem Wert initialisiert)

(2) die Anweisung wird ausgeführt

(3) am Ende des Blocks werden die Variablen aus der Deklaration wieder gelöscht

lokale Variablen die in der *Variablenliste* aufgezählten Variablen sind zu dem *Block* lokal, alle anderen Variablen heißen **globale** Variablen

Syntaxerweiterung

bedingte Ausdrücke

Ausdruck	::=	... | *bedingterAusdruck*
bedingterAusdruck	::=	if $Ausdruck_0$
		then $Ausdruck_1$
		else $Ausdruck_2$

Bedeutung

(1) Die Bedingung $Ausdruck_0$ wird ausgewertet

(2a) wenn das Resultat true ist, so wird $Ausdruck_1$ ausgewertet und dieser Wert ist das Resultat des *bedingtenAusdrucks*

(2b) wenn das Resultat false ist, so wird $Ausdruck_2$ ausgewertet und dieser Wert ist das Resultat des *bedingtenAusdrucks*

↪ viele Algorithmen können kürzer und eleganter formuliert werden

Verallgemeinerung mit Algorithmen mit Parametern können ähnliche aber leicht unterschiedliche Rechenvorschriften zu einer Vorschrift zusammengefaßt werden.

Abstraktion das Ersetzen von Konstanten durch Parameter in einem Algorithmus heißt Abstraktion

Beispiel

$piep(n : \mathsf{N}_0)$ bildet eine Abstraktion des Algorithmus für das Spiel mit 7

dieser Prozeß der Abstraktion kann wiederholt angewendet werden:
$piep(n : \mathsf{N}_0; b : \mathsf{N}_1)$

Applikation Die Ersetzung eines Parameters durch einen Wert in einem Algorithmus heißt Applikation (Anwendung). Dieser Prozeß kann wieder (wie die Abstraktion) schrittweise erfolgen.

Vorteile von Algorithmen mit Parametern (Module, Prozeduren und Funktionen)

Schrittweise Verfeinerung mit **top–down** Vorgehen wird in natürlicher Weise unterstützt

Abgeschlossenheit Ein Teilalgorithmus kann unabhängig von dem Kontext, in dem er verwendet wird, entwickelt werden

information hiding ein Modul braucht nur die Schnittstelle und eine Beschreibung **was** der Modul macht, nach außen bekannt zu machen. **Wie** ein Modul eine Funktion berechnet (mit welchem Verfahren), wird versteckt

$\hookrightarrow$

Auswechselbarkeit von Teil–Algorithmen ohne Veränderung der Funktionalität des Gesamtalgorithmus

$\hookrightarrow$

Wartbarkeit Veränderbarkeit, Erweiterbarkeit, Anpassung

Wiederverwendung Teilalgorithmen können in einer *Bibliothek (library)* von Algorithmen gespeichert und wiederverwendet werden

information hiding

Information verstecken ist eine wesentliche Aufgabe der Zerlegung in Teilalgorithmen

es wird nach außen nur die Schnittstelle des Algorithmus (die formalen Parameter und das Resultat) bekanntgemacht

einschließlich der Vorbedingungen

Welche Annahmen werden über die Parameter gemacht?

einschließlich der Invarianten

Welche globalen Variablen werden nicht verändert?

einschließlich der Nachbedingungen

Welche Eigenschaften haben die veränderten globalen Variablen?

die Realisierung (Implementierung) bleibt nach außen hin unbekannt (versteckt) und kann so verändert oder ausgewechselt werden

Prozeduren

2. Art von Algorithmen
berechnen keinen Wert,
sondern verändern den globalen Zustand
(die globalen Programmvariablen)

Syntax Prozeduren

ProzedurDefinition	::=	*ProzedurKopf ProzedurRumpf*
ProzedurKopf	::=	*ProzedurName* (*formaleParamListe*)
	|	*ProzedurName*
ProzedurRumpf	::=	*Anweisung*
Anweisung	::=	... | *Prozeduraufruf*
ProzedurAufruf	::=	*ProzedurName* (*aktuelleParamListe*)
	|	*ProzedurName*

Semantik eines Prozeduraufrufs

(1) die aktuellen Parameter des Aufrufs werden berechnet

(2) Der Text des *ProzedurRumpf*es wird kopiert und alle formalen Parameter in diesem Text werden durch die Werte der aktuellen Parameter ersetzt

(3) danach wird die neu entstandene Anweisung ausgeführt

5.2 Rekursion

Ringelnatz …daß dieser Wurm an Würmern litt, die wiederum an Würmern litten.

in der Natur Schneckenhaus,…

Rekursion in einem Algorithmus ist die Verwendung des Algorithmus selbst zur Berechnung des Resultats

Idee

(1) für einige wenige einfache Parameter (Argumente) ist das Resultat direkt (ohne Rekursion) berechenbar

(2) für die komplexeren Parameterwerte wird die Berechnung auf einfache Werte zurückgeführt

abbrechen die Rekursion muß abbrechen, d.h. die rekursiven Aufrufe müssen immer mit einfacheren Werte ausgeführt werden

einfache Werte was einfache Werte sind, hängt vom Problem ab

kleine Zahlen	große Zahlen
kurze Listen	lange Listen
einfache Ausdrücke	geschachtelte Ausdrücke

nicht strikte Auswertung Boolescher Ausdrücke :

Konjunktionen und Disjunktionen werden nur so weit ausgewertet, bis das Resultat bekannt ist.

$\hookrightarrow$ das Resultat ist auch dann definiert, wenn die Auswertung eines Teilausdrucks nicht definiert ist

x∧y $\Leftrightarrow$ `if x then y else false`

$\Leftrightarrow$ `if x then y else x`

x∨y $\Leftrightarrow$ `if x then true else y`

$\Leftrightarrow$ `if x then x else y`

Auswertung Boolescher Ausdrücke

```
eval(be : Boolescher Ausdruck) : B
  if konstant(be)
    then be
    else
  if Variable(be)
    then „der be zugeordnete Wert“
    else
  if Negation(be)
    then b1 := eval(Teilausdruck(be))
      ¬b1
    else
  if Konjunktion(be)
    then b1 := eval(linkerTeilausdruck(be))
      if b1
        then eval(rechterTeilausdruck(be))
        else b1
    else
  if Disjunktion(be)
    then b1 := eval(linkerTeilausdruck(be))
      if b1
        then b1
        else eval(rechterTeilausdruck(be))
    else
  if Prädikat(be)
    then evalPrädikat(be)
    else ...
```


Programmier–sprachen fast alle Programmiersprachen unterstützten Rekursion

Ausnahmen: alte Versionen von FORTRAN, BASIC, COBOL

Iteration Schleifen sind ein Spezialfall von Rekursion

$\hookrightarrow$ jede Schleife kann in eine Rekursion transformiert werden

Rekursion ist allgemeiner, nicht jede Rekursion kann in eine Schleife transformiert werden

Wirth: **Die weit verbreitete Antipathie gegen Rekursion ist eine Folge schlechter Erziehung.**

Problem	Geld wechseln
	Wieviele verschiedene Möglichkeiten gibt es, 1,- DM zu wechseln mit Fünfzig-, Zehn-, Fünf-, Zwei-, und Einpfennigstücken?
allgemein	Können wir einen Algorithmus schreiben, der die Anzahl der Möglichkeiten zum Wechseln eines beliebigen Geldbetrages berechnet?
Lösung	rekursiv
# gesamt	den Betrag a mit n verschiedenen Münzen zu wechseln
=	
#	Möglichkeiten, den Betrag a mit allen außer der 1. Münzart zu wechseln
+	
#	Möglichkeiten, den Betrag $a-d$ mit allen n Münzarten zu wechseln, wobei d der Nennwert der 1. Münzart ist.
elementare Fälle	Wann kennen wir das Ergebnis ohne weitere Berechnungen?
$a = 0$	genau eine (1) Wechselmöglichkeit
$a < 0$	keine (0) Wechselmöglichkeiten
$n = 0$	keine (0) Wechselmöglichkeiten

sortieren	und mischen
Idee	• sortiere 1. Hälfte • sortiere 2. Hälfte • mische beide Hälften

Funktion

```
sortiere(l : Liste) : Liste
    if Laenge(l) <= 1
    then l
    else
        mische(
            sortiere(1.Haelfte(l)),
            sortiere(2.Haelfte(l))
        )
```

$\hookrightarrow$ Arbeit wird beim Mischen gemacht

5.3 Parallelität

Konstrukte	Folge, Auswahl und Wiederholung werden sequentiell ausgeführt,
$\hookrightarrow$	Auswertung von Ausdrucken und aktuellen Parametern werden sequentiell ausgeführt
Vorstellung	ein Prozessor führt genau einen Algorithmus Schritt für Schritt hintereinander aus
	Viele Algorithmen enthalten Anweisungen und Ausdrücke, die unabhängig voneinander ausgeführt werden können, deren Berechnungsreihenfolge beliebig ist
$\hookrightarrow$	diese Algorithmen sind geeignet, von Mehrprozessormaschinen ausgeführt zu werden
$\hookrightarrow$	viele Prozessoren $\Rightarrow$viel gleichzeitig $\Rightarrow$Zeitgewinn

---

teile & herrsche	rekursive Algorithmen, die ein Problem in mehrere einfache Probleme dergleichen Art aufteilen, enthalten oft unabhängige Berechnungsschritte, die unabhängig voneinander (gleichzeitig) ausgewertet werden können.
Beispiele	Fibonacci, sortieren und mischen
nicht	Türme von Hanoi

5.4 Zusammenfassung

Algorithmen–Konstruktion

Sequenz	
Verzweigung	if . . . then . . . else . . . end if
Wiederholung	while . . . do . . . end while for . . . to . . . do . . . end for
Teilalgorithmen	Prozeduren und Funktionen mit Parametern
Rekursion	durch Wiederverwendung des gleichen Algorithmus mit *einfacheren* Werten
Parallelität	durch gleichzeitiges Ausführen unabhängiger Teile Beispiel: teilen und herrschen

---

ungenau

Daten	zusammengesetzte Werte (Felder, Listen, . . .)
Operationen	auf den Werten (Vergleiche, Addition, Selektion, . . .)
Speicherung	und Verwaltung der Werte
$\hookrightarrow$	**Datenstrukturen**

---

6 Formale Sprachen und Grammatiken

6.1 Einleitung

formale Sprache besteht aus:

- einem festen endlichen Grundvokabular
- einer festen Syntax und Semantik

Syntaxdefinition

Alphabet A eine nichtleere, endliche Menge von Zeichen

Wort über einem Alphabet A:
eine endliche Folge von Zeichen aus A

ϵ leeres Wort

A^* Menge aller Wörter über dem Alphabet A

Sprache über A Jede Teilmenge $L \subseteq A^*$ heißt Sprache über A

Definition von Sprachen über einem Alphabet A

$\hookrightarrow$

Grammatik Regelmenge zur Syntax–Definition
(Konstruktion) einer formalen Sprache

Notationen

.1

Erkennender Automat Akzeptor
akzeptiert syntaktisch korrekte Zeichenreihen

$\hookrightarrow$ Spezialfall:
endlicher Automat

.2

Grammatik Regelmenge zur Syntaxdefinition

BNF Spezialfall:
Backus–Naur–Form für kontextfreie Sprachen

.3

Syntaxdiagramm graphische Darstellung einer Regelmenge einer kontextfreien Grammatik

6.2 Endliche Automaten

endlicher Automat	*finite automaton* *finite state machine*	
$\hookrightarrow$	sehr einfaches Automatenmodell	
	nicht alle praktisch interessanten Sprachen lassen sich mit endlichen Automaten beschreiben	
Bestandteile	ein *erkennender* endlicher Automat A ist ein 5–Tupel $A = (I, Q, \delta, q_0, F)$	
Eingabealphabet	I	endliche, nichtleere Menge
Zustandsmenge	Q	endliche, nichtleere Menge
Anfangszustand	q_0	$q_0 \in Q$
Endzustände	F	$F \subseteq Q$
Übergangstabelle	δ	$\delta : Q \times I \longrightarrow Q$
$\hookrightarrow$	jedem Zustands–Eingabepaar (q, i) ist (höchstens) ein Folgezustand q' zugeordnet	
$\hookrightarrow$	deterministischer endlicher Automat	
Verallgemeinerung	der Übergangstabelle	
	δ	$\delta : Q \times I \longrightarrow Q$ -set
$\hookrightarrow$	*nichtdeterministischer* endlicher Automat	

akzeptierte Sprache	ein akzeptierender endlicher Automat A akzeptiert eine Sprache $L(A)$
$L(A)$	$L(A) = \{w \in I^* \mid (q_0, w) \overset{*}{\vdash} q,\ q \in F\}$
$(q_0, w) \overset{*}{\vdash} q$	der Anfangszustand q_0 wird durch Eingabe von w in einen Endzustand $q \in F$ überführt
$\hookrightarrow$	ein endlicher Automat definiert eine formale Sprache
genauer	ein endlicher Automat definiert eine *reguläre Sprache*
$\hookrightarrow$	*reguläre Ausdrücke*
EA	die Menge der Sprachen EA, die von endlichen Automaten akzeptiert werden.

6.3 Grammatiken

Grammatik eine Grammatik G ist ein 4–Tupel
$G = (T, N, P, S)$

Bestandteile

Terminalsymbole T — eine nichtleere Menge von Zeichen
das Alphabet

Nichtterminal-symbole N — eine nichtleere Menge von Zeichen
$T \cap N = \{\}$

Regeln P — Produktionssystem
eine nichtleere Menge von Regeln
bestehend aus linker und rechter Seite

allgemeine Form $X_1 \ldots X_n \ ::= \ Y_1 \ldots Y_m$

$X_i, Y_j \ \in \ N \cup T$
$n \geq 1$
$m \geq 0$
mindestens ein $X_i \in N$

Startsymbol S — $S \in N$

Ausgangspunkt für eine Ableitungsfolge

Ableitungsschritt

gegeben $x \ \in \ (N \cup T)^*$ — ein Wort
$p \ ::= \ q \ \in \ P$ — eine Regel
$x = x_1 p x_2$ — p ist Teilwort von x

Ergebnis $y = x_1 q x_2$

$\hookrightarrow$ p wird in x durch q ersetzt

Notation $x_0 \underset{p::=q}{\longrightarrow} x_n$

Ableitung $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$
$x_0 \overset{*}{\underset{G}{\longrightarrow}} x_n$

Sprache $L(G)$ die Menge aller Wörter, die aus dem Startsymbol S ableitbar sind und nur aus Terminalsymbolen bestehen

$\hookrightarrow$ $L(G) \ = \ \{w \in T^* \mid S \overset{*}{\underset{G}{\longrightarrow}} w\}$

BNF Backus–Naur–Form

$\hookrightarrow$ eine kompakte Form der Darstellung der Grammatikregeln

Beispiel

$$\begin{aligned} S &::= S + S \\ &\mid S * S \\ &\mid (S) \\ &\mid a \end{aligned}$$

ist die BNF von

$$\begin{aligned} S &::= S + S \\ S &::= S * S \\ S &::= (S) \\ S &::= a \end{aligned}$$

$\hookrightarrow$ | bedeutet *oder*

Syntaxdiagramm graphische Darstellung der Syntaxregeln

Klassifikation von Grammatiken

$\hookrightarrow$ durch die Form der Regeln

$\hookrightarrow$ Einschränkungen an die linken und rechten Seiten

6.4 Rechtslineare Grammatiken

Regeln alle Regeln sind von folgender Form:

$$A ::= w_1B \qquad A \in N,\ w_1 \in T^*,\ B \in N$$

$$A ::= w_2 \qquad A \in N,\ w_2 \in T^*$$

$\hookrightarrow$ im Regelkopf (linke Seite) genau ein Nichtterminalsymbol

$\hookrightarrow$ im Regelrumpf (rechte Seite) eine Folge von Terminalsymbolen, optional gefolgt von einem Nichtterminalsymbol

RL die Menge der Sprachen RL, die mit rechtslinearen Grammatiken erzeugt werden können

$\hookrightarrow$ die Menge RL der rechtslinearen Sprachen ist gleich der Menge der Sprachen EA, die von endlichen Automaten akzeptiert werden.

$\hookrightarrow$ zu jeder rechtslinearen Grammatik gibt es einen akzeptierenden endlichen Automaten und umgekehrt.

Effizienz von endlichen Automaten

$\hookrightarrow$ sehr gut

Zeit der Test, ob ein Wort w zu einer Sprache L gehört, kann für $L \in RE$ in einer Zeit $\sim$ Länge des Wortes w durchgeführt werden.

$\hookrightarrow$ w in den endlichen Automaten stecken

$\delta : Q \times I \longrightarrow Q$

Übergangstabelle schrittweise für alle Zeichen aus w auswerten.

$\hookrightarrow$ ein Zustandsübergang benötigt konstante Rechenzeit

Speicher die Syntaxanalyse benötigt nur Speicher für den Zustand, d.h. eine Variable zur Speicherung von Werten aus der Menge Q.

$\hookrightarrow$ der Speicherplatzbedarf hängt nicht von dem zu analysierenden Wort w ab.

$\hookrightarrow$ rechtslineare Grammatiken und endliche Automaten werden für die *lexikalische Analyse* von Programmiersprachen eingesetzt.

lexikalische Analyse Zerlegen der Eingabezeichenfolge in eine Folge von Symbolen *token*.

nicht alle interessanten Sprachen L sind aus RE $(= EA)$.

6.5 kontextfreie Grammatiken

$\hookrightarrow$ Regeln allgemeiner als bei rechtslinearen Grammatiken

Regeln $A ::= w$ $\quad A \in N$, $w \in (N \cup T)^*$

Einschränkung Regelkopf wie bei rechtslinearen Grammatiken

keine Einschränkung an die rechte Seite

$\hookrightarrow$ jede rechtslineare Grammatik ist auch eine kontextfreie Grammatik

CF die Menge der Sprachen, die mit kontextfreien Grammatiken definiert werden können, ist die Menge der *kontextfreien Sprachen CF*.

$\hookrightarrow$ $RE \subset CF$

Ableitungen in kontextfreien Grammatiken können als Bäume dargestellt werden

Bedeutung des Ableitungsbaums:

Die Semantik eines Wortes (eines Satzes, eines Programms) wird durch die Struktur des Ableitungsbaums festgelegt.

$\hookrightarrow$ rekursive Interpretation des Ableitungsbaums

Problem:

zu einem Wort kann es mehrere strukturell verschiedene Ableitungsbäume geben.

$\hookrightarrow$ mehrere Bedeutungen zu einem Satz.

mehrdeutige Grammatiken.
In Programmiersprachen unerwünscht.

es gibt kontextfreie Sprachen, die nur durch mehrdeutige Grammatiken definiert werden können (z.B. PASCAL).

Effizienz der Analyse von kontextfreien Sprachen

allgemeine kontextfreie Sprachen können nur mit *nichtdeterministischen Kellerautomaten* analysiert werden.

es ist kein Verfahren bekannt, mit dem man *nichtdeterministische Kellerautomaten* effizient implementieren kann.

allgemeine kontextfreie Sprachen sind nicht effizient zu analysieren.

weitere Einschränkungen erforderlich!

Einschränkung	an kontextfreie Sprachen
deterministische	kontextfreie Sprachen
Definition	eine deterministische kontextfreie Sprache L ist eine Sprache, zu der es eine kontextfreie Grammatik G gibt, bei der jedes Wort $w \in L$ genau einen Ableitungsbaum besitzt.
DCF	die Menge der deterministischen kontextfreien Sprachen
$\hookrightarrow$	$DCF \subset CF$
$\hookrightarrow$	deterministische kontextfreie Sprachen können von *deterministischen Kellerautomaten* akzeptiert werden.
Effizienz	für deterministische Kellerautomaten gibt es effiziente Implementierungen
Zeit	Laufzeit $\sim$ Länge des Eingabewortes
Speicher	Kellerspeicher in der Größe $\sim$ zur Länge der Eingabe
$\hookrightarrow$	mehr Platz als bei endlichen Automaten

Generatoren	für die Syntaxanalyse
$\hookrightarrow$	für Teilmengen von DCF gibt es effiziente Syntaxanalysatoren
$LR(1)$	die Menge der Sprachen, die durch $LR(1)$–Grammatiken definiert werden können.
.1	ein Wort w kann erkannt werden, indem es von links (L) nach rechts gelesen wird
.2	dabei wird eine Rechtsableitung (R) konstruiert
.3	es muß maximal ein Zeichen ((1)) im Eingabestrom vorausgeschaut werden
$\hookrightarrow$	$LR(1) \subset DCF$
$\hookrightarrow$	fast alle Programmiersprachen besitzen eine $LR(1)$–Grammatik
Generatoren	für $LR(1)$–Grammatiken gibt es Generatoren, die aus einer Grammatik einen Parser (Syntaxanalyseprogramm) erzeugen
yacc, bison	die bekanntesten Parsergeneratoren für $LR(1)$–Grammatiken

6.6 kontextsensitive Grammatiken

$\hookrightarrow$	Regeln allgemeiner als bei kontextfreien Grammatiken
Regeln	$v_1 A v_2 \ ::= \ v_1 w v_2 \qquad v_1, v_2, w \in (N \cup T)^*$, $A \in N$
$\hookrightarrow$	$A \ ::= \ w$ darf nur im Kontext $v_1 \ldots v_2$ angewendet werden.
CS	die Menge der Sprachen, die mit kontextsensitiven Grammatiken erzeugt werden können.
$\hookrightarrow$	$CF \subset CS$
	es ist kein effizientes Verfahren bekannt, um kontextsensitive Sprachen zu analysieren.
$\hookrightarrow$	für *Programmier*sprachen keine praktische Bedeutung
$\hookrightarrow$	für Programmiersprachen andere Techniken, um zusätzliche Bedingungen festzulegen, die nicht mit kontextfreien Grammatiken formuliert werden können.
Bedingungen	Sind alle Variablen deklariert?
	Werden alle Variablen nur gemäß ihrer Deklaration verwendet?
Techniken	z.B. attributierte Grammatiken.

6.7 CH-0– Grammatiken

$\hookrightarrow$	Regeln ohne weitere Einschränkungen
CH-0	die Menge aller Sprachen, die mit CH-0–Grammatiken erzeugt werden können.
$\hookrightarrow$	alle berechenbaren Sprachen
$\hookrightarrow$	alle Sprachen, deren Worte durch einen Algorithmus aufgezählt werden können.
TM	die Menge aller Sprachen, die von einer Turing–Maschine akzeptiert werden können.
$\hookrightarrow$	$CH\text{-}0 = TM$

6.8 Chomsky–Hierarchie

Chomsky N. Chomsky, amerikanischer Sprachwissenschaftler, * 1928

$\hookrightarrow$ Anordnung der Sprachklassen in einer Hierarchie

$\hookrightarrow$
$$\begin{aligned} RL &= EA \\ &\subset LR(1) \\ &\subset DCF \\ &\subset CF \\ &\subset CS \\ &\subset CH\text{-}0 = TM \end{aligned}$$

$\hookrightarrow$ im Gebiet der Formalen Sprachen und Automatentheorie wird diese Hierarchie noch wesentlich verfeinert

7 Berechenbarkeit und Komplexität

7.1 Berechenbarkeit

zentrale Frage	Gibt es zu jedem Problem einen Algorithmus der dieses Problem löst?
Antwort	**!!! NEIN !!!**
	nur für „wenige“ Probleme gibt es einen Algorithmus
Gödel	1931
Unvollständigkeits–	
Theorem	Es gibt keinen Algorithmus, der als Eingabe eine Aussage über natürliche Zahlen erhält, und dessen Ausgabe feststellt, ob diese Aussage wahr oder falsch ist.
$\hookrightarrow$	Church, Kleene, Post, Turing
$\hookrightarrow$	andere nicht berechenbare Probleme

Halteproblem	Gibt es einen Algorithmus, der feststellt, ob ein beliebiges Programm eine Endlosschleife enthält?
???	Gibt es einen Algorithmus, der feststellt, ob ein beliebiges Programm jemals stoppt?
!!!	**!!! NEIN !!!**
Halt–Test	Eingabe: Programm P und Daten D
	Ausgabe: ja, wenn $P(D)$ jemals hält nein, wenn $P(D)$ in Endlosschleife läuft
Beweis	durch Widerspruch
(1)	Nehme an, es gibt ein Programm $Halt - Test$
(2)	Benutze $Halt - Test$ zur Konstruktion von *nudelt*
(3)	Zeige, daß *nudelt* widersprüchliche Eigenschaften hat (nicht hält, nicht in Endlosschleife läuft)
(4)	folgere, daß die Annahme falsch ist

Totalitäts– Problem	Hält ein Programm P bei allen Eingaben D?
Äquivalenz– Problem	Berechnen zwei Programme P_1 und P_2 immer die gleiche Ausgabe (oder laufen beide in eine Endlosschleife)?
	beide Probleme sind nicht berechenbar

nicht berechenbare Probleme

Collatz–Funktion Terminierung unbekannt

Grammatik

Gleichheit zweier kontextfreien Sprachen

Durchschnitt zweier kontextfreier Sprachen

---

partielle Berechenbarkeit Ein Problem mit Antworten ja und nein ist partiell berechenbar, wenn es einen Algorithmus gibt, der ja ausgibt und hält, wenn die Antwort ja lautet, der aber möglicherweise in eine Endlosschleife läuft, wenn die Antwort nein lautet.

Halteproblem partiell berechenbar
Programm ausführen

Totalitäts– und Äquivalenz–Problem nicht partiell berechenbar

7.2 Komplexität

berechenbar impliziert nicht **durchführbar**

Betriebsmittel *resource*

zur Ausführung von Algorithmen notwendig

Speicher *memory*

zur Aufbewahrung von Zwischenergebnissen

Zeit *time*

zur Ausführung von Operationen

Komplexität eines Problems

Komplexität des bestmöglichen Algorithmus zur Lösung des Problems

Abschätzung oft wird eine Abschätzung der Komplexität gemacht, da eine exakte Berechnung nicht möglich ist

asymptotisches Verhalten wird angegeben

Ordnung qualitative Abschätzung einer Funktion

$$f, g : \mathsf{N}_0 \longrightarrow \mathsf{R}$$

$f(n) = O(g(n))$ f hat die Ordnung von g

wenn es Konstanten $c, n_0 \in \mathsf{N}_0$ gibt mit:

$|f(n)| \leq c|g(n)|$ für alle $n \geq n_0$

typische Ordnungsfunktionen

$\log n$	logarithmisch	sehr gut
n	linear	gut
$n \log n$	fast linear	gut
n^2	quadratisch	schon schlecht
n^m	polynomisch vom Grad m	
e^n	exponentiell	undurchführbar

alle Algorithmen, die Resourcen exponentiell verbrauchen sind praktisch unbrauchbar

alle Probleme, die nur mit exponentiellen Algorithmen gelöst werden können, sind sogenannte harte Probleme (NP–vollständig)

Zeitbedarf von Algorithmen mit unterschiedlicher Zeitkomplexität

n	$\log n$	n	n^2	2^n
10	0.000003 sec	0.00001 sec	0.001 sec	0.001 sec
100	0.000007 sec	0.0001 sec	0.01 sec	10^{16} Jahre
1000	0.00001 sec	0.001 sec	1 sec	∞
10000	0.000013 sec	0.01 sec	1.7 min	∞
100000	0.000017 sec	0.1 sec	2.8 std	∞

Abschätzungen

im Mittel durchschnittlicher Bedarf

im schlechtesten Fall *worst case*

im besten Fall *best case*

rekursive Relationen typisch für teile und herrsche Algorithmen

NP-vollständig	ein Problem, das bei sequentiellen Maschinen exponentiellen Zeitaufwand zur Lösung erfordert
$\hookrightarrow$	ein Problem, das nur mit einer **nichtdeterministischen** Turing–Maschine in **polynomischer** Zeit zu lösen ist
nicht–deterministisch	eine Maschine, die beliebig viele Prozessoren hat (die beliebig viele Operationen gleichzeitig durchführen kann)

Beispiele

Rucksackproblem	*knapsack problem*, Kastenproblem
Handlungs–reisender	*traveling salesman problem*

8 Beispielprogramme

algsumme.pas

```
program p(output);

{ berechne die Summe aller Funktionswerte zwischen 1 und n
  einer Funktion auf den natuerlichen Zahlen
  Summantion mittels while-Schleife
}

type Nat0 = 0..maxint;

function fsumme(n : Nat0;
                function f(i : Nat0) : Nat0) : Nat0;
var
  result : Nat0;
begin
  result := 0;
  while not (n = 0) do
  begin
    result := result + f(n);
    n := n - 1;
  end;
  fsumme := result;
end;

function quadrat(i : Nat0) : Nat0;
begin
  quadrat := i * i
end;

function fib(i : Nat0) : Nat0;
begin
  if i <= 1
  then fib := i
  else fib := fib(i-1) + fib(i-2)
end;

function ident(i : Nat0) : Nat0;
begin
  ident := i;
end;

begin
  writeln('Berechnung der Summe von Zahlenfolgen');
  writeln('Quadratsumme   von 1 bis  25 = ',fsumme( 25,quadrat));
  writeln('Fibonaccisumme von 1 bis  10 = ',fsumme( 10,fib    ));
  writeln('Summe          von 1 bis 100 = ',fsumme(100,ident  ));
end.
```


max31.pas

```
program p(output);

{ berechne das Maximum dreier Zahlen nur mit Verzweigungen }

type Nat0 = 0..maxint;

function max3(n : Nat0;
              m : Nat0;
              p : Nat0 ) : Nat0;
begin
  if n >= m
  then                                            { n >= m }
    if n >= p
    then                             { (n >= m) and (n >= p) }
      max3 := n
                  { (n >= m) and (n >= p) and (max3 = n) }
    else                             { (n >= m) and (n <  p) }
      max3 := p
                  { (p >  n) and (n >= m) and (max3 = p) }
    {end if}
  else                                            { n < m }
    if m >= p
    then                             { (n <  m) and (m >= p) }
      max3 := m
                  { (m >  n) and (m >= p) and (max3 = m) }
    else                             { (n <  m) and (m <  p) }
      max3 := p
                  { (p >  m) and (m >  n) and (max3 = p) }
    {end if}
  {end if}
          { ((max3 = n) or (max3 = m) or (max3 = p)) and
             (max3 >= n) and (max3 >= m) and (max3 >= p) }
end;

begin
  writeln('Berechnung des Maximums 3-er natuerlicher Zahlen');
  writeln('max(4,3,2) = ',max3(4,3,2));
  writeln('max(3,4,2) = ',max3(3,4,2));
  writeln('max(3,2,4) = ',max3(3,2,4));
  writeln('max(2,4,3) = ',max3(2,4,3));
  writeln('max(0,0,0) = ',max3(0,0,0));
end.
```

max32.pas

```
program p(output);

{ berechne das Maximum dreier Zahlen
  durch Zurueckfuehren auf das Maximum 2-er Zahlen
}

type Nat0 = 0..maxint;

function max (n : Nat0;
              m : Nat0 ) : Nat0;
begin
  if n >= m
  then
    max := n
  else
    max := m
  {end if}
  { ((max = n) or (max = m)) and (max >= n) and (max >= m) }
end;

function max3(n : Nat0;
              m : Nat0;
              p : Nat0 ) : Nat0;
begin
  max3 := max(max(n,m),p)
end;

begin
  writeln('Berechnung des Maximums 3-er natuerlicher Zahlen');
  writeln('max(4,3,2) = ',max3(4,3,2));
  writeln('max(3,4,2) = ',max3(3,4,2));
  writeln('max(3,2,4) = ',max3(3,2,4));
  writeln('max(2,4,3) = ',max3(2,4,3));
  writeln('max(0,0,0) = ',max3(0,0,0));
end.
```


max33.pas

```
program p(output);

{ berechne das Maximum dreier Zahlen
  durch Vertauschen der Parameter
}

type Nat0 = 0..maxint;

function max3(n : Nat0;
              m : Nat0;
              p : Nat0 ) : Nat0;
begin
  if n < m
  then max3 := max3(m,n,p)
  else
  if m < p
  then max3 := max3(n,p,m)
  else max3 := n
end;

begin
  writeln('Berechnung des Maximums 3-er natuerlicher Zahlen');
  writeln('max(4,3,2) = ',max3(4,3,2));
  writeln('max(3,4,2) = ',max3(3,4,2));
  writeln('max(3,2,4) = ',max3(3,2,4));
  writeln('max(2,4,3) = ',max3(2,4,3));
  writeln('max(0,0,0) = ',max3(0,0,0));
end.
```

plus.pas

```
program p(output);

{ berechne die Summe zweier ganzer Zahlen und
  merke Ueberlaeufe:
  x+y > maxint => r = maxint, overflow
  x+y < minint => r = minint, overflow
  sonst        => r = x + y , ok
}

function plus(x : integer;
              y : integer;
              var overflow : boolean) : integer;
var
  r : integer;
begin
  overflow := false;
  if (x >= 0) = (y < 0) then
    begin
    r := x + y
    end
  else
  if (x >= 0) then
    begin
    if x <= (maxint - y) then
      r := x + y
    else
      begin
      overflow := true;
      r := maxint
      end
    end
  else
    begin
    if (minint - x) <= y then
      r := x + y
    else
      begin
      overflow := true;
      r := minint
      end
    end;
  plus := r;
end;

procedure testplus(x : integer; y : integer);
var
  overflow : boolean;
  result   : integer;
begin
```



```
  result := plus(x,y,overflow);
  writeln(x,' + ',y,' = ',result,
          ' overflow = ',overflow);
end;

begin
  writeln('Berechnung der Summe zweier Zahlen');
  writeln('mit Ueberlauftest');
  testplus(     1,     1);
  testplus(     1,    -1);
  testplus(    -1,     1);
  testplus(    -1,    -1);
  testplus(maxint,     0);
  testplus(minint,     0);
  testplus(maxint,minint);
  testplus(minint,maxint);
  testplus(maxint,     1);
  testplus(maxint,maxint);
  testplus(minint,    -1);
  testplus(minint,minint);
end.
```

mitte31.pas

```
program p(output);

{ berechne die mittlere von drei Zahlen
  auf direktem Weg
}

type Nat0 = 0..maxint;

function mitte3(n : Nat0;
                m : Nat0;
                p : Nat0 ) : Nat0;
begin
  if n >= m
  then
    if m >= p
    then mitte3 := m
    else
      if n >= p
      then mitte3 := p
      else mitte3 := n
  else
    if n >= p
    then mitte3 := n
    else
      if m >= p
      then mitte3 := p
      else mitte3 := m
end;

begin
  writeln('Berechnung der mittleren von 3 natuerlicher Zahlen');
  writeln('mitte(4,3,2) = ',mitte3(4,3,2));
  writeln('mitte(3,4,2) = ',mitte3(3,4,2));
  writeln('mitte(3,2,4) = ',mitte3(3,2,4));
  writeln('mitte(2,4,3) = ',mitte3(2,4,3));
  writeln('mitte(0,0,0) = ',mitte3(0,0,0));
end.
```


mitte32.pas

```
program p(output);

{ berechne die mittlere von drei Zahlen
  mit Hilfe der Minimumsfunktion       }

type Nat0 = 0..maxint;

function min(n : Nat0;
             m : Nat0 ) : Nat0;
begin
  if n >= m
  then min := m
  else min := n
end;

function mitte3(n : Nat0;
                m : Nat0;
                p : Nat0 ) : Nat0;
begin
  if n >= m
  then
    if m >= p
    then mitte3 := m
    else mitte3 := min(n,p)
  else
    if n >= p
    then mitte3 := n
    else mitte3 := min(m,p)
end;

begin
  writeln('Berechnung der mittleren von 3 natuerlicher Zahlen');
  writeln('mitte(4,3,2) = ',mitte3(4,3,2));
  writeln('mitte(3,4,2) = ',mitte3(3,4,2));
  writeln('mitte(3,2,4) = ',mitte3(3,2,4));
  writeln('mitte(2,4,3) = ',mitte3(2,4,3));
  writeln('mitte(0,0,0) = ',mitte3(0,0,0));
end.
```

mitte33.pas

```
program p(output);

{ berechne die mittlere von drei Zahlen
  durch Vertauschen der Parameter
}

type Nat0 = 0..maxint;

function mitte3(n : Nat0;
                m : Nat0;
                p : Nat0 ) : Nat0;
begin
  if n < m
  then mitte3 := mitte3(m,n,p)
  else
  if m < p
  then mitte3 := mitte3(n,p,m)
  else mitte3 := m
end;

begin
  writeln('Berechnung der mittleren von 3 natuerlicher Zahlen');
  writeln('mitte(4,3,2) = ',mitte3(4,3,2));
  writeln('mitte(3,4,2) = ',mitte3(3,4,2));
  writeln('mitte(3,2,4) = ',mitte3(3,2,4));
  writeln('mitte(2,4,3) = ',mitte3(2,4,3));
  writeln('mitte(0,0,0) = ',mitte3(0,0,0));
end.
```


muenzen.pas

```
program muenzen(output);

{ Berechnung der Anzahl der Arten, auf die 1,-DM
  gewechselt werden kann
}

type
  Nat0    = 0..maxint;
  Nat1    = 1..maxint;

function Nennwert(muenze : Nat1) : Nat1;
begin
  if muenze = 1 then Nennwert :=   1
  else
  if muenze = 2 then Nennwert :=   2
  else
  if muenze = 3 then Nennwert :=   5
  else
  if muenze = 4 then Nennwert :=  10
  else
  if muenze = 5 then Nennwert :=  50
  else
  if muenze = 6 then Nennwert := 100
  else
  if muenze = 7 then Nennwert := 200
  else
                     Nennwert := 500
end;

function wechseln(betrag     : Integer;
                  muenzarten : Nat0   ) : Nat0;
begin
  if betrag = 0
  then
    wechseln := 1
  else
  if ( betrag < 0 ) or ( muenzarten = 0 )
  then
    wechseln := 0
  else
    wechseln := wechseln( betrag - Nennwert(muenzarten)
                        , muenzarten)
                +
                wechseln(betrag, muenzarten -1)
end;

function wechselgeld(betrag : Integer) : Nat0;
begin
  wechselgeld := wechseln(betrag,5)
```

```
end;

begin
  writeln('Geld wechseln');
  writeln('0,10 DM kann auf ',wechselgeld( 10):1,
                       ' Arten gewechselt werden');
  writeln('0,50 DM kann auf ',wechselgeld( 50):1,
                       ' Arten gewechselt werden');
  writeln('1,-  DM kann auf ',wechselgeld(100):1,
                       ' Arten gewechselt werden');
end.
```


plus1.pas

```
program plus(output);

{ Zurueckfuehren der Addition zweier Zahlen auf
  inkrementieren und dekrementieren
  in linear rekursiver Form,
  in endrekursiver Form und
  mit der aus der endrekursiven Form
  transformierten while-Schleife
}

type Nat0 = 0..maxint;

function plus1(a : Nat0; b : Nat0) : Nat0;
begin
  if a = 0
  then plus1 := b
  else plus1 := 1 + plus1( a -1, b)
end;

function plus2(a : Nat0; b : Nat0) : Nat0;
begin
  if a = 0
  then plus2 := b
  else plus2 := plus2(a -1, b +1)
end;

function plus3(a : Nat0; b : Nat0) : Nat0;
begin
  while not ( a = 0 ) do
  begin
    a := a -1;
    b := b +1;
  end;
  plus3 := b
end;

begin
  writeln('Addition durch Zaehlen');
  writeln('3 + 4 = ',plus1(3,4):1,
              ' = ',plus2(3,4):1,
              ' = ',plus3(3,4):1);
end.
```

prime1.pas

```
program prime(output);

{ Primzahltest durch Berechnung des kleinsten Teilers
  einer Zahl: rekursive Loesung                         }

type Nat1 = 1..maxint;

function findeTeiler(n : Nat1; i : Nat1) : Nat1;
begin
  if i * i > n
  then
    findeTeiler := n
  else
  if n mod i = 0
  then
    findeTeiler := i
  else
    findeTeiler := findeTeiler(n, i +1)
end;

function kleinsterTeiler(n : Nat1) : Nat1;
begin
  kleinsterTeiler := findeTeiler(n,2)
end;

function istprim(n : Nat1) : Boolean;
begin
  istprim := kleinsterTeiler(n) = n
end;

begin
  writeln('Primzahltest durch Berechnung des kleinsten Teilers');
  writeln('istprim( 13)         = ',istprim(13));
  writeln('istprim( 91)         = ',istprim(91));
  writeln('kleinsterTeiler( 91) = ',kleinsterTeiler( 91));
  writeln('istprim(561)         = ',istprim(561));
  writeln('kleinsterTeiler(561) = ',kleinsterTeiler(561));
end.
```


prime2.pas

```
program prime(output);

{ Primzahltest durch Berechnung des kleinsten Teilers
  einer Zahl
  iterative Loesung
}

type Nat1 = 1..maxint;

function findeTeiler(n : Nat1; i : Nat1) : Nat1;
begin
  while not( i * i > n)
    and not( n mod i = 0)
  do begin
    i := i +1;
  end;
  if i * i > n
  then findeTeiler := n
  else findeTeiler := i
end;

function kleinsterTeiler(n : Nat1) : Nat1;
begin
  kleinsterTeiler := findeTeiler(n,2)
end;

function istprim(n : Nat1) : Boolean;
begin
  istprim := kleinsterTeiler(n) = n
end;

begin
  writeln('Primzahltest durch Berechnung des kleinsten Teilers');
  writeln('istprim( 13)         = ',istprim(13));
  writeln('istprim( 91)         = ',istprim(91));
  writeln('kleinsterTeiler( 91) = ',kleinsterTeiler( 91));
  writeln('istprim(561)         = ',istprim(561));
  writeln('kleinsterTeiler(561) = ',kleinsterTeiler(561));
end.
```

qsumme1.pas

```
program p(output);

{ berechne die Quadratsumme aller Zahlen zwischen 1 und n
  linear rekursive Loesung
}

type Nat0 = 0..maxint;

function quadratsumme(n : Nat0) : Nat0;
begin
  if n = 0
  then quadratsumme := 0
  else quadratsumme := n * n + quadratsumme(n-1)
end;

begin
  writeln('Berechnung der Quadratsumme von natuerlichen Zahlen');
  writeln('Quadratsumme von 1 bis 25 = ',quadratsumme(25));
  writeln('Quadratsumme von 1 bis  0 = ',quadratsumme(0) );
end.
```


qsumme2.pas

```
program p(output);

{ berechne die Summe aller Quadratzahlen zwischen 1 und n
  iterative Loesung
}

type Nat0 = 0..maxint;

function quadratsumme(n : Nat0) : Nat0;
var
  result : Nat0;
begin
  result := 0;
  while not ( n = 0 ) do
  begin
    result := result + n * n;
    n := n - 1;
  end;
  quadratsumme := result
end;

begin
  writeln('Berechnung der Quadratsumme von natuerlichen Zahlen');
  writeln('Quadratsumme von 1 bis 25 = ',quadratsumme(25));
  writeln('Quadratsumme von 1 bis  0 = ',quadratsumme(0) );
end.
```

summe1.pas

```
program p(output);

{ berechne die Summe aller Zahlen zwischen 1 und n
  linear rekursive Loesung
}

type Nat0 = 0..maxint;

function summe(n : Nat0) : Nat0;
begin
  if n = 0
  then summe := 0
  else summe := n + summe(n-1)
end;

begin
  writeln('Berechnung der Summe von natuerlichen Zahlen');
  writeln('Summe von 1 bis 25 = ',summe(25));
  writeln('Summe von 1 bis  0 = ',summe(0) );
end.
```


summe2.pas

```
program p(output);

{ berechne die Summe aller Zahlen zwischen 1 und n
  iterative Loesung
}

type Nat0 = 0..maxint;

function summe(n : Nat0) : Nat0;
var
  result : Nat0;
begin
  result := 0;
  while not ( n = 0 ) do
  begin
    result := result + n;
    n := n - 1;
  end;
  summe := result
end;

begin
  writeln('Berechnung der Summe von natuerlichen Zahlen');
  writeln('Summe von 1 bis 25 = ',summe(25));
  writeln('Summe von 1 bis  0 = ',summe(0) );
end.
```

wurzel.pas

```
program wurzeln(output);
{ Berechnung der Wurzel einer reellen Zahl
  mit dem Newtonschen Iterationsverfahren
  Algorithmus durch schrittweise Verfeinerung entwickelt }

function mittelwert(a : Real; b : Real) : Real;
begin
  mittelwert := (a + b) / 2.0;
end;

function gutgenug(y : Real; x : Real) : Boolean;
begin
  gutgenug := abs(y*y - x) < 0.0001
end;

function verbessern(y : Real; x : Real) : Real;
begin
  verbessern := mittelwert(y, x/y)
end;

function wurzeliter(y : Real; x : Real) : Real;
begin
  if gutgenug(y,x)
  then wurzeliter := y
  else wurzeliter := wurzeliter(verbessern(y,x),x)
end;

function wurzel(x : Real) : Real;
begin
  wurzel := wurzeliter(1.0,x)
end;

begin
  writeln('Wurzel berechnen mit Newton Iterationsverfahren');
  writeln('Die Wurzel von  1.0 ist ',wurzel( 1.0));
  writeln('Die Wurzel von  2.0 ist ',wurzel( 2.0));
  writeln('Die Wurzel von  4.0 ist ',wurzel( 4.0));
  writeln('Die Wurzel von 10.0 ist ',wurzel(10.0));
end.
```