1612 Konzepte imperativer Programmierung

Lehrziele

KE1

- Wie ist der Entwicklungsweg vom Problem bis zum Programm?
  ➔ Realweltproblem -> Problembeschreibung -> Problemspezifikation -> Algorithmus -> Programm -> Maschinenprogramm -> Ergebnis
- Definition: "Problembeschreibung"
  ➔ Sprachliche Darstellung eines Problems; bei der informellen Problembeschreibung werden überflüssige Details weggelassen, man konzentriert sich nur auf die für das Problem relevanten Tatbestände.
  ➔ Legt fest, was gemacht werden soll, ist aber oft unpräzise, mehrdeutig, widersprüchlich oder auch unvollständig.
- Definition: "Problemspezifikation"
  ➔ Erhält man durch Präzisierung und Formalisierung der Problembeschreibung, legt auch fest, was gemacht werden soll.
- Unterschiede: Problembeschreibung - Problemspezifikation
  ➔ Die Problembeschreibung ist eine saloppe Definition, die Problemspezifikation dagegen muß bestimmten formalen Regeln genügen und ist dadurch kompakter und präziser.
- Definition: "Algorithmus"
  ➔ Ein Algorithmus ist ein Verfahren, das durch eine Folge von Anweisungen / (primitiven) Operationen beschrieben wird. Durch die schrittweise Abarbeitung dieses Verfahrens wird das gegebene Problem gelöst.
  ➔ Ein Algorithmus ist eine Menge von Regeln für ein Verfahren, um aus gewissen Eingabegrößen bestimmte Ausgabegrößen herzuleiten. Dabei müssen nachfolgende Bedingungen erfüllt sein:
  - Finitheit der Beschreibung: Die Beschreibung muß endlich sein
  - Effektivität: Jeder Schritt muß ausführbar sein
  - Terminierung: Das Verfahren muß in endlich vielen Schritten enden
  - Determiniertheit: Der Ablauf des Verfahrens ist in jedem Punkt fest vorgeschrieben, es gibt keine Auswahlmöglichkeiten
- Definition "Programm"
  ➔ Ein Programm ist ein in einer Programmiersprache verfaßter Algorithmus.
- Unterschiede: Algorithmus - Programm
  ➔ Die einzelnen Anweisungen eines Programmes können durchaus mehrere primitive Operationen eines Algorithmus beinhalten.
- Was ist eine Maschinensprache
  ➔ Eine Sprache, die der Computer versteht. Alle in einer anderen Programmiersprache vorliegenden Programme müssen mit Hilfe eines Compilers in ein Maschinenprogramm übersetzt werden, damit sie vom Computer ausgeführt werden können.

- Welche verschiedenen Programmierparadigmen (Konzepte von Programmiersprachen) gibt es?
  - ➔ imperative Programmiersprache
    - Das Programm besteht aus einer Folge von Befehlen.
    - Das Variablenkonzept basiert auf einer logischen Speicherplatzverwaltung, das heißt der Programmierer muß sich nicht darum kümmern an welcher Speicherstelle der Wert der Variablen abgelegt wird, man spricht sie einfach mit dem Namen an
    - spiegelt die von-Neumann-Architektur wieder
    - Beispiele: ADA, ALGOL-60, ALGOL-68, BASIC, C, COBOL, FORTRAN, MODULA-2, PASCAL, PL/1 und SIMULA.
  - ➔ funktionale / applikative Programmiersprachen
    - Ein Programm entspricht einer Funktion von einer Menge von Eingabewerten eine Menge von Ausgabewerten.
    - Die Eingabewerte entsprechen Parametern
    - Funktional: Die Funktionen werden durch Zusammensetzung von mehreren einfachen Funktionen definiert.
    - Applikativ: Die Funktionen werden durch Anwendung von Funktionen auf ihre Parameter definiert, ein Programm ist somit eine Folge von Funktionsdefinitionen.
    - Beispiele für applikative Programmiersprachen: LISP und LOGO.
  - ➔ Prädikative Programmiersprache
    - Unter Programmieren versteht man hier das Beweisen in einem System von Tatsachen und Schlußfolgerungen. Man gibt eine Menge von Funktionen und Regeln vor und der Rechner prüft dann ob eine Aussage wahr oder falsch ist.
    - Beispiel: PROLOG.
  - ➔ objektorientierte Programmiersprachen
    - Es wird mit Objekten programmiert, die Nachrichten miteinander austauschen und Parameter haben können. Jedes Objekt kann dabei nur bestimmte Nachrichten verstehen.
    - Beispiel: SMALLTALK-80, C++, EIFFEL.

- Definition "Von-Neumann-..."
  - ➔ Rechner:
    - Computer mit speicherbaren Programmen
  - ➔ Architektur:
    - prägt auch heute noch den Aufbau von Computern
    - umfaßt 5 Funktionseinheiten: Steuerwerk, Rechenwerk, Speicher, Ein- und Ausgabewerk
    - fällt in die Klasse der SISD-Architektur (single instruction stream, single data stream). Diese Rechner bestehen aus einem Prozessor, der aus Steuer- und Rechenwerk besteht, die Befehlsabarbeitung erfolgt dabei sequentiell.

- Computer-Eigenschaften
  - ➔ *Geschwindigkeit:* Millionen Operationen pro Sekunde (MIPS)
  - ➔ *Speicherfähigkeit:* große Informationsmengen
  - ➔ *Zuverlässigkeit:* geringe Anzahl von auftretenden Fehlern
  - ➔ *Universalität:* Die Funktionalität ist abhängig vom jeweiligen Programm

- Wo liegen die Unterschiede von niederen und höheren Programmiersprachen.
  - ➔ niedere Programmiersprachen sind maschinennahe Sprachen, wie zum Beispiel Assembler.
  - ➔ höhere Programmiersprachen bieten mehr Komfort und Sicherheit beim Programmieren, die Kommandos ähneln eher der Umgangssprache. Programme von höheren Programmiersprachen werden durch einen Compiler (Übersetzer) in Maschinenprogramme übersetzt. Höhere Programmiersprachen sind meist problemorientiert oder anwendungsspezifisch ausgelegt (Beispiel: COBOL: Datenverwaltung, FORTRAN: mathematisch)
- Einteilung der Programmiersprachen in niedere und höhere Programmiersprachen
  - ➔ niedere Programmiersprachen:
    Maschinensprache, Assembler
  - ➔ höhere Programmiersprachen:
    FORTRAN, COBOL, BASIC, C, PL1, RPG, MODULA-2

- Welche Funktion hat das Steuerwerk ?
  - ➔ Das Steuerwerk ist das "Herz" des Rechners. Es dient zum Laden der Befehle in richtiger Reihenfolge aus dem Speicher, zur Decodierung und Interpretation der Befehle, sowie zur Versorgung der an der Ausführung der Befehle beteiligten Funktionseinheiten mit den nötigen Steuersignalen.
  - ➔ Das Steuerwerk ist mit dem Rechenwerk, dem Speicher und den Ein- und Ausgabegeräten verbunden, da diese Komponenten alle bei der Ausführung eines Befehls beteiligt sind.
  - ➔ Das Steuerwerk besteht aus Befehlsregister, Befehlszählregister, Decodierer und der Mikroprogrammeinheit.
- Welche Funktion hat das Rechenwerk ?
  - ➔ Das Rechenwerk ist die Komponente des Rechners in der arithmetische und logische Verknüpfungen durchgeführt werden, deshalb wird es auch ALU (Arithmetic Logical Unit) genannt.
  - ➔ Das Rechenwerk besteht aus Addierwerk und Komplementation, Hilfs- und Zwischenregistern, der Operationssteuerung und dem Akkumulator für die Aufnahme des Ergebnisses.
- Was ist die CPU?
  - ➔ Steuerwerk und Rechenwerk faßt man auch unter der Bezeichnung CPU (Central Processing Unit) oder Prozessor zusammen.

- Was sind die verschiedenen Speichertypen, wo liegen die Unterschiede?
  - ➔ Die Speichertypen unterscheiden sich in Ihrer Zugriffsart, es gibt Speicher mit wahlfreiem/direktem, zyklischem oder sequentiellen Zugriff.
  - ➔ wahlfreier/direkter Zugriff:
    Jede Speicherzelle ist unabhängig von Ihrer Position im Speicher in der gleichen Zeit erreichbar. Beispiel: Hauptspeicher.
  - ➔ zyklischer Zugriff:
    Die Speicherzellen rotieren und sind nur periodisch zugänglich.
    Beispiele: Magnetplattenspeicher, Diskette.
  - ➔ sequentieller Zugriff:
    Ein Zugriff auf die gewünschte Speicherzelle ist erst dann möglich, wenn alle Vorgänger-Speicherzellen bereits gelesen wurden. Beispiel: Magnetband.

- Wie sieht der grundsätzliche Aufbau von PASCAL-Programmen aus?

  ➔ Programm

  program heading block .

  program Programmname (Ein/Ausgabedaten) ; declaration part statement part

  constant definition type definition variable declaration begin statements end

- Welche Zeichen beinhaltet der Zeichenvorrat in PASCAL?
  - ➔ Groß- und Kleinbuchstaben außer Umlauten
  - ➔ Ziffern
  - ➔ Spezialsymbole: +, - , *, /, ., :, ,, ;, =, <>, <=, <, >=, >, :=, .., ^, (,), [,], {, }
  - ➔ Schlüsselwörter: Bsp.: and, begin, end, array, type, var, const, uses, div, mod, to, downto, do, while, repeat, until, if, then, else, for, odd, ord, program, function, ...
- Was ist ein Bezeichner?
  - ➔ Eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt, und nicht mit Schlüsselwörtern identisch ist
- Welche Standardbezeichner gibt es in PASCAL?
  - ➔ input, output, readln, writeln, read, write
- Welche Trennsymbole gibt es in PASCAL?
  - ➔ Blank, eoln, Kommentar

- Welche Datentypen gibt es in PASCAL?
  - ➔ einfache / unstrukturierte Datentypen :
    - Standarddatentypen
    - selbstdefinierte Datentypen
  - ➔ strukturierte Datentypen (array, record)
  - ➔ Zeigerdatentypen

- Welche Standarddatentypen gibt es in PASCAL?
  - ➔ INTEGER
    - Definitionsbereich: Intervall der ganzen Zahlen, -maxint ..0..maxint ($-2^{15}..0..+2^{15}$)
    - gültige Werte: Folge von Ziffern, die eventuell ein Vorzeichen haben
    - alle Zwischenergebnisse müssen auch in dem gültigen Intervall liegen
    - darauf definierte Standardoperationen: +, -, *, div, mod
    - darauf definierte Standardfunktionen : abs, sqr, succ, pred, odd
    - Jeder Wert außer dem Kleinsten hat einen Vorgänger, jeder außer dem Größten einen Nachfolger.
    - auf den Werten ist eine Ordnung definiert

- ➔ REAL
  - Teilmenge der rationalen Zahlen
  - Zahlen, die über die Grenzen von Integer hinausgehen können oder einen Fraktionsteil enthalten
  - Festpunktdarstellung: vor und hinter dem Dezimalpunkt muß mindestens eine Ziffer stehen. Die Anzahl der Ziffern ist beschränkt
  - Gleitpunktdarstellung zur Basis 10: muß für die Eingabe nicht normalisiert sein, intern wird jedoch mit normalisierten Werten gerechnet. Für die Mantisse ist nur eine bestimmte Stellenanzahl vorhanden, deshalb kann es zu Rundungsfehlern kommen. Interne Form: $z=m*b^e$, Eingabeform: +/-VK.NKE+/-EXP.
  - Standardoperationen: +, -, *, /
  - Standardfunktionen : abs, sqr, sin, cos, arctan, exp, ln, sqrt
- ➔ CHAR
  - Menge aller darstellbaren Zeichen des PASCAL-Zeichensatzes sowie einige Steuerzeichen
  - wird in Hochkommas definiert
  - es existiert eine Ordnung meist entsprechend dem ASCII-Code
  - Ziffern und Großbuchstaben bilden bei ASCII je eine lückenlos geordnete Menge
  - Standardfunktionen: ord, chr, pred, succ
- ➔ BOOLEAN
  - Werte TRUE und FALSE
  - Verknüpfungen mit boolschen Operatoren : and, or, not
  - Verknüpfung mit Vergleichsoperatoren
- ➔ STRING
  - Menge aller Zeichenketten bis zu einer Länge von meist 255 Zeichen
  - gibt's nur in TURBO-PASCAL
  - Vergleichsoperatoren

- Welche <u>selbstdefinierten Datentypen</u> gibt es in PASCAL?
  - ➔ <u>Aufzählungstypen</u>
    - type typname = (a1,a2,...,an);
    - Die Mengen von verschiedenen Aufzählungsdatentypen müssen disjunkt sein, die Elemente dürfen nicht aus Integer oder char sein
    - es existiert eine Ordnung => Standardfunktionen : ord, succ, pred
  - ➔ <u>Ausschnittsdatentyp</u>
    - definiert ein Intervall eines bereits definierten Grundtyps (integer, char oder Aufzählungstyp)
    - type typname = 1..8;

- Welche Arten von <u>Anweisungen</u> gibt es in PASCAL?
  - ➔ <u>einfache Anweisungen</u>
    - leere Anweisung: ';'
    - Zuweisung / Wertzuweisung : 'x:=expr'
    - Zusammengesetzte Anweisung : ' begin ... end'
    - Spezialanweisungen (Ein- und Ausgabeanweisungen)
    - Aufrufe von Funktionen oder Prozeduren
  - ➔ <u>strukturierte Anweisungen</u>
    - Bedingungsanweisungen: if then else und if then
    - Wiederholungsanweisungen: while ... do, repeat ... until, for ... do

- Was versteht man unter dem 'dangling-else'-Problem?
  - ➔ Darunter versteht man das Problem der richtigen Zuordnung von else-Zweigen. Falls man ein Semikolon zuviel macht, kann es zu einem semantischen Fehler kommen.
  - ➔ Man kann dies dadurch umgehen, daß man nur im else-Zweig schachtelt, denn dann wird ein solcher Fehler vom Compiler entdeckt, weil dann zugleich auch ein Syntax-Fehler vorliegt, da es kein If mehr zu einem Else-Zweig gibt.

- Wie sind die Prioritäten bei der Auswertung eines Ausdrucks?
  - ➔ not *Höchste Priorität*
  - ➔ *, /, div, mod, and (Punktoperationen)
  - ➔ +, -, or (Strichoperationen)
  - ➔ =, <=, <, >=, > (Vergleichsoperatoren) *niedrigste Priorität*
  - ➔ bei gleicher Priorität wird von links nach rechts ausgewertet
  - ➔ Klammern sind stärker als die definierten Prioritäten

KE2

- Was ist eine Wiederholungsanweisung?
  - ➔ Eine Wiederholungsanweisung / Schleife führt eine Folge von Anweisungen so lange aus, wie die Schleifenbedingung erfüllt ist. Wenn die Schleifenbedingung nicht mehr erfüllt ist, bricht die Schleifeniteration ab.
  - ➔ Abbruchbedingung / -Kriterium = Negation der Schleifenbedingung
  - ➔ Schleifendurchlauf = einmalige Ausführung der zu wiederholenden Anweisungsfolge
  - ➔ Endlosschleife
    - entsteht, wenn das Abbruchkriterium nicht erfüllt werden kann, dann wird der Schleifenrumpf unendlich oft abgearbeitet.
- Welche Wiederholungsanweisungen es in PASCAL?
  - ➔ FOR-Schleife (for lauf := Anfexp to / downto Endexp do begin statements end;)
    - wird auch Laufanweisung genannt
    - wird benutzt, wenn die Anzahl der Schleifendurchläufe vor Eintritt der Schleife bekannt ist
    - Die Kontrollvariable / Laufvariable muß einem unstrukturierten Datentyp außer real angehören.
    - Anfangsausdruck, Endausdruck und die Kontrollvariable müssen denselben Datentyp haben.
    - Anfangsausdruck und Endausdruck werden nur beim erstmaligen Eintritt in die Schleife berechnet.
    - Der Wert der Kontrollvariablen wird automatisch nach dem Schleifendurchlauf erhöht (to) bzw. vermindert (downto).
    - Nach der Schleife ist der Wert der Kontrollvariablen wieder undefiniert.
  - ➔ WHILE-Schleife (while (Bedingung) do begin statements end;)
    - wird dann verwendet, wenn die Anzahl der Schleifendurchläufe im voraus noch nicht bekannt ist und jeder Schleifendurchlauf von der Schleifenbedingung abhängt
    - Die Bedingung wird vor jedem Schleifendurchlauf überprüft.
    - Falls die Bedingung erfüllt ist, wird der Schleifenrumpf abgearbeitet, sonst wird die Bearbeitung des Schleifenrumpfes abgebrochen bzw. übersprungen.
    - Besteht die Schleifenbedingung aus mehreren Teilausdrücken, ist beim Abbruch unklar, warum sie abgebrochen ist. Falls dies von Bedeutung für den weiteren Programmverlauf ist, muß dies durch ein if abgefragt werden.
    - Die Schleifenbedingung muß eine Variable enthalten, deren Wert sich bei der Bearbeitung der Schleife ändert, ansonsten ist die Endlosschleife schon vorprogrammiert.
  - ➔ REPEAT-UNTIL-Schleife (repeat statements until (Bedingung);)
    - wird verwendet, wenn der erste Durchlauf unabhängig von der Schleifenbedingung immer erfolgen muß. Beispiel: Eingabe eines Wertes, solange bis er korrekt eingegeben wurde.
    - Der Schleifenrumpf wird mindestens einmal abgearbeitet.
    - Die Prüfung des Abbruchkriteriums erfolgt nach Abarbeitung des Schleifenrumpfes.

- Umwandlung einer while-Schleife in eine repeat-Schleife

  ➔ while (Bedingung) do
  statement;

  wird zu

  if (Bedingung) then
  repeat
  statement
  until (not Bedingung);

- Umwandlung einer repeat-Schleife in eine while-Schleife

  ➔ repeat
  statement
  until (Bedingung);

  wird zu:

  statement;
  while (not Bedingung) do
  statement;

- Welche strukturierten Datentypen gibt es in PASCAL?
  ➔ ARRAY
  - type
    typname = array [tindex] of componenttype;
  - Der Indextyp (tIndex) beschreibt eine endliche Folge von Werten und muß ein Aufzählungs- oder Ausschnittstyp, char, integer oder boolean sein.
  - Der Komponententyp darf ein beliebiger Datentyp sein.
  - Eine Bereichsüberschreitung liegt vor, falls eine Komponente angesprochen wird, die nicht existiert, d.h. falls der Indexwert außerhalb der definierten Werte liegt.
  - Die Initialisierung eines Feldes (Belegung mit Anfangswerten) ist gut in einer Schleife realisierbar.
  - Es ist sinnvoll, die Feldgrenzen als Konstanten zu definieren und die Konstatnen auch für die Bearbeitung des Feldes zu verwenden, oder das Feld ausreichend groß zu definieren und durch eine Variable, die die Größe angibt nur den belegten Teil des Feldes zu bearbeiten.
  - Ein eindimensionales Feld kann zur Darstellung von Vektoren verwendet werden.
  - Mehrdimensionales Array:
    - enthält mehrere Indextypen
    - ist zur Darstellung von Matrizen geeeignet
  - Strings:
    - eigentlich arrays mit dem Komponententyp char
    - Indextyp ist dann ein Ausschnitt von integer der bei 1 beginnt
    - Definition: string[maxlänge]
    - Funktionen mit strings: length: gibt die aktuelle Länge, also die Anzahl der

Zeichen in integer zurück

➔ RECORD-Typ (Verbund, Verbundtyp)
- type
  typname = record
  n1: typ1;
  ...
  end;
- Der Komponententyp darf beliebig sein.
- Die Komponentenbezeichner innerhalb eines records müssen eindeutig sein.
- Ein record selbst kann Komponente eines Arrays sein.
- Der Zugriff auf eine Komponente erfolgt folgendermaßen :
  "Verbundname.Komponentenbezeichner"

- Funktionsvereinbarung
  - ➔ function Funktionsname (Liste der formalen Parameter): Ergebnistyp; block;
  - ➔ Der Funktionskopf (siehe oben) beschreibt, wie die Funktion heißt und wie man sie aufruft.
  - ➔ Im Block / Funktionsrumpf wird mit Hilfe der Parameter der Funktionswert berechnet. Der Block besteht wie bei Programmen aus Deklarations- und Anweisungsteil. Die Vereinbarungen von Konstanten, Typen, Variablen, Funktionen und Prozeduren sind nur innerhalb dieses Blocks gültig somit also lokal.
  - ➔ erfolgt im declaration part im Anschluß an die Variablendeklaration
  - ➔ legt fest, wie die Funktion heißt, wie sie aufgerufen wird und wie der Funktionswert berechnet wird
  - ➔ Der Anweisungsteil einer Funktion muß mindestens eine Anweisung enthalten, nämlich die in der dem Funktionsnamen ein Wert vom Ergebnistyp zugewiesen wird.
  - ➔ Ein Funktionsaufruf entspricht einem Ausdruck, dabei werden die Werte der aktuellen Parameter nachdem sie ausgewertet wurden den formalen Parametern zugewiesen.
  - ➔ Einem Funktionsbezeichner kann zwar mehrmals ein Wert zugewiesen werden, aber man kann nicht lesend auf ihn zugreifen, man kann ihn also nicht als lokale Variable verwenden, da jedes Auftreten des Funktionsbezeichners als Funktionsaufruf angesehen wird.

- Definition "Gültigkeitsbereich"
  - ➔ von Objekten ist der Block in dem sie vereinbart worden sind
- Definition "Lebensdauer / Existenz von Variablen"
  - ➔ Lokale Variablen existieren nur während der Ausführung der Prozedur / Funktion. Beim Aufruf werden sie bereitgestellt, d.h. es wird Speicherplatz reserviert und am Ende werden sie zerstört (der Speicherplatz wird dann wieder freigegeben).
  - ➔ Es gibt keine Beziehung zwischen den Werten einer lokalen Variablen bei aufeinanderfolgenden Aufrufen.
- Definition "formaler Parameter"
  - ➔ Ein formaler Parameter ist der Parameter, wie er im Funktions- / Prozedurkopf vereinbart wird.

- Definition "aktueller Parameter"
  - ➔ Aktuelle Parameter sind die Parameter, die einer Funktion beim Funktionsaufruf mitgegeben werden. Es kann ein Ausdruck sein.
  - ➔ Die Anzahl der aktuellen Parameter muß gleich der Anzahl der fomalen Parameter sein.
  - ➔ Jeder aktuelle Parameter ersetzt denjenigen formalen Parameter, der an derselben Position der Parameterliste steht.
  - ➔ Der Typ des aktuellen Parameters muß dem Typ des entsprechenden formalen Parameters entsprechen.
- Vorteile von Funktionen / Prozeduren bei korrekter Verwendung des Parameterkonzeptes, d.h. ohne globale Variablen
  - ➔ Strukturierung des Programms
    - ▫ Das Programm wird dadurch übersichtlicher.
    - ▫ Das Programmieren kann leichter auf mehrere Programmierer verteilt werden.
    - ▫ Das Programm ist einfacher zu testen.
  - ➔ Mehrfachverwendung innerhalb des Programms
    - ▫ Dadurch wird Coderedundanz vermieden.
    - ▫ Das Programm kann leichter geändert werden, da man nur an einer Stelle ändern muß und nicht an mehreren.
    - ▫ Das Programm wird kürzer.
  - ➔ Wiederverwendbarkeit in anderen Programmen
    - ▫ Zum Beispiel durch Ablage der Funktionen und Prozeduren in einer Bibliothek

KE3

- Prozeduren
  - ➔ Die Deklaration erfolgt analog zu der Deklaration von Funktionen, jedoch mit dem Schlüsselwort 'procedure' und ohne Ergebnistyp.
  - ➔ Prozeduraufruf: Prozedurname mit der aktuellen Parameterliste. Der Prozeduraufruf ist eine eigenständige Anweisung.

- Parameterkonzept von ParaPASCAL
  - ➔ Eingangs- bzw. in-Parameter
    - entsprechen lokalen Konstanten, die beim Prozedur- / Funktionsaufruf ihren Wert vom aktuellen Parameter erhalten.
    - Sowohl der formale als auch der aktuelle Parameter können durch die Funktion / Prozedur nicht geändert werden
    - Funktionen dürfen nur in-Parameter haben.
  - ➔ Ausgangs- bzw. out-Parameter
    - entsprechen lokalen Variablen, deren Wert nicht abgefragt werden kann, man kann den out-Parametern nur Werte zuweisen.
    - Der Wert des formalen Parameters wird dem aktuellen Parameter unmittelbar vor Beendigung der Prozedur zugewiesen
    - Hat der aktuelle Parameter bereits beim Prozeduraufruf einen Wert, so ist dieser unbedeutend, er wird bei Beendigung der Prozedur überschrieben
  - ➔ Änderungs- bzw. inout-Parameter
    - entsprechen lokalen Variablen, denen beim Prozeduraufruf der Wert des aktuellen Parameters übergeben wird.
    - Während der Prozedur darf diese Variable sowohl abgefragt, als auch verändert werden.
    - Vor Beendigung der Prozedur wird dem aktuellen Parameter der Inhalt des formalen Parameters übergeben.

- Parameterarten in PASCAL
  - ➔ Wertübergabe, call by value
    - Ein 'call bei value'-Parameter entspricht dem in-Parameter von ParaPASCAL. Es ist ein reiner Eingabeparameter, ein Wertparameter
    - Die Vereinbarung erfolgt ohne ein Schlüsselwort; also ohne "var" und der Bezeichner sollte das Präfix 'in' haben.
  - ➔ Referenzübergabe, call by reference
    - Ein 'call by reference'-Parameter entspricht den inout-Parametern.
    - Beim Prozeduraufruf wird ein Verweis (Referenz, Zeiger, Adresse der Speicherzelle) auf den aktuellen Parameter an den formalen Parameter übergeben.
    - Jede Änderung des formalen Parameters ist also zugleich auch eine Änderung des aktuellen Parameters.
    - Ein solcher Parameter wird mit dem Schlüsselwort 'var' vereinbart.
    - Der Bezeichner sollte das Präfix 'io' haben.
    - out-Parameter müssen durch call by reference simuliert werden, zur Unterscheidung sollten ihre Bezeichner den Präfix 'out' haben.

- Definition "statische Blockstruktur"
  - ➔ Die Gesamtstruktur eines PASCAL-Programms besteht aus einer Menge von Blöcken, die auch ineinander geschachtelt sein können. Da diese Struktur vom (statischen) Programmtext ablesbar ist, wird sie statische Blockstruktur genannt.
- Definition "dynamische Blockstruktur"
  - ➔ Die dynamische Blockstruktur ergibt sich aus den (dynamischen) Funktions- und Prozeduraufrufen während der Laufzeit. Jeder Aufruf erzeugt einen neuen inneren Block, in dem die lokalen Vereinbarungen vorgenommen werden.

- Definition "Schachtelung"
  - ➔ Eine Schachtelung liegt dann vor, wenn eine Funktion / Prozedur innerhalb einer anderen Funktion / Prozedur deklariert wird.
- Definition "globale Variable"
  - ➔ Eine globale Variable ist global für alle inneren Blöcke.
  - ➔ Die Standardbezeichner gelten als vereinbart in einem fiktiven Block, der das gesamte Programm umschließt. Sie sind also global zum Programm vereinbart.
- Definition "lokale Variable"
  - ➔ Eine Variable ist lokal zu dem Block, in dem sie deklariert wurde.

- Definition "Gültigkeitsbereich"
  - ➔ Der Gültigkeitsbereich von Bezeichnern ist an der statischen Blockstruktur ablesbar. Es ist immer der Block, in dem der Bezeichner vereinbart wurde, also auch die inneren Blöcke mit Ausnahme von denen, in denen der Bezeichner nochmals vereinbart wird (Namenskonflikt).
  - ➔ Der Gültigkeitsbereich entspricht dem Sichtbarkeitsbereich von Variablen. (Scope-Regeln)
- Definition "Lebensdauer / Existenz"
  - ➔ Die Lebensdauer / Existenz von Bezeichnern ist abhängig von der dynamischen Blockstruktur, da lokale Bezeichner nur während der Ausführung der Prozedur / Funktion existieren in der sie vereinbart wurden (Binderegeln).
- Definition "Namenskonflikt"
  - ➔ Ein Namenskonflikt liegt dann vor, wenn ein Bezeichner eines äußeren Blocks in einem inneren Block nochmals vereinbart wurde. Im inneren Block kann dann nur auf den dort vereinbarten Bezeichner dieses Namens zugegriffen werden.
- Definition "Seiteneffekt"
  - ➔ Seiteneffekte können auftreten, wenn innerhalb einer Funktion oder Prozedur nicht nur die Änderungs- und Ausgangsparameter manipuliert werden, sondern zum Beispiel auch globale Variablen. Eine andere Art von Seiteneffekt ergibt sich, wenn man Funktionen Änderungs- oder Ausgangsparameter übergibt.

- Was versteht man unter dem Zeigerkonzept / Referenzkonzept ?
  - ➔ Beim Zeiger- bzw. Referenzkonzept, werden Objekte anstatt über ihren Namen über Verweise (Referenzen) angesprochen, d.h. über ein Objekt in dem die Speicheradresse des gesuchten Objektes abgelegt ist.
  - ➔ Damit kann man dynamische Datenstrukturen realisieren
  - ➔ Mit Hilfe einer Zeigervariablen kann man den Inhalt des Objektes auf das letztendlich gezeigt wird õndern

- Definition eines Zeigertyps
  - ➔ type
    tRefTyp = ^tTyp;
  - ➔ dabei darf tTyp ein beliebiger Datentyp sein
- Deklaration einer Zeigervariablen
  - ➔ var
    zeig : tRefTyp;
  - ➔ Mit zeig^ kann man auf den Inhalt der Variablen zugreifen, auf den zeig zeigt
- Definition "nil"
  - ➔ Hat ein Zeiger den Wert nil, bedeutet das, da_ er auf kein Objekt zeigt
- Operationen mit Zeigern
  - ➔ Zuweisung von Werten anderer Zeigervariablen
  - ➔ Erzeugen eines neuen Objektes: new(zeig);
    Dabei wird ein neues Objekt vom Zeigertyp erzeugt, der Inhalt des Objektes ist solange undefiniert, bis zeig^ initialisiert wurde.
  - ➔ Löschen eines Objektes: dispose(zeig)
    Nach diesem Aufruf ist kein Zugriff auf das Objekt mehr m÷glich und der Wert des Zeigers ist undefiniert. Der Speicherplatz wird freigegeben.
  - ➔ Vergleichen von Zeigern
    Zwei Zeiger sind gleich, wenn sie auf dasselbe Objekt zeigen.

- Was ist ein typisches Merkmal von dynamische Datenstrukturen?
  - ➔ Die Anzahl der Objekte einer dynamischen Datenstruktur kann während des Programmlaufes geändert werden.

- Lineare Liste
  - ➔ Eine lineare Liste entsteht durch Verkettung von mehreren Objekten.
- Typdefinition einer Linearen Liste
  - ➔ type
    tRefListe = ^tListe;
    tListe = record
    info : integer;
    next : tRefListe
    end;
  - ➔ Die Typdefinition ist eine Vorwärtsdefintion, der Typ tListe muß im selben Typdefinitionsteil definiert sein.
  - ➔ Die Liste muß immer einen Anfangszeiger haben, der auf das erste Listenelement zeigt

- Listenoperationen
  - ➔ Listenaufbau
    Ein neues Listenelement wird stets am Anfang der Liste eingefügt
  - ➔ Listendurchlauf
    Ein Listendurchlauf erfolgt zum Beispiel zur Ausgabe der Listenelemente, dabei aber nur solange wie zeiger <> nil, d.h. while (zeiger <> nil) do
  - ➔ Suchen eines Elementes in der Liste
    - Sonderfall: Liste ist leer
    - deshalb:
      if (Anf <> nil) then und dann
      while (zeig^.next <> nil) and
      (zeig^.info <> such);
  - ➔ Entfernen aus der Liste, dabei gibt es folgende Sonderfälle:
    - Wenn das erste Element gelöscht werden muß, ist der Anfangszeiger weiterzusetzen, der Anfangszeiger muß deshalb mit var deklariert werden.
    - Die Liste ist leer, es kann nichts entfernt werden.
    - Das Element ist nicht in der Liste.
    - Das Element ist gefunden, der Vorgänger muß auf den Nachfolger umverzeigert werden.

KE4

- Definition: "Baum"
  - ➔ Ein Baum besteht aus einer Menge von Objekten, die Knoten genannt werden, zusammen mit einer Relation auf der Knotenmenge, die graphisch durch Kanten dargestellt wird. Die Relation organisiert die Knotenmenge hierarchisch
  - ➔ Jeder Baum enthält genau einen Knoten, der keinen Vorgänger hat (die Wurzel)
  - ➔ Jeder Knoten ungleich der Wurzel kann von der Wurzel durch genau eine Kantenfolge erreicht werden.
  - ➔ Die Höhe eines Baumes entspricht der Anzahl der Knoten des längsten Pfades von der Wurzel zu einem Blatt.
  - ➔ Die Wurzel ist der oberste Knoten eines Baumes, sie hat keinen Vorgänger.
  - ➔ Nachfolger, Söhne oder Kinder sind Knoten, die mit einem Vorgängerknoten durch genau eine Kante verbunden sind.
  - ➔ Die Wurzel ist Vorgänger / Vater für ihre direkten Nachfolger.
  - ➔ Ein Blatt ist ein Knoten ohne Nachfolger.
  - ➔ Ein innerer Knoten ist ein Knoten, der weder die Wurzel des Gesamtbaumes noch ein Blatt ist.
  - ➔ Jeder Knoten ist die Wurzel eines Teilbaumes, der entweder leer ist oder sich aus allen von diesem Knoten aus über eine Kantenfolge erreichbaren Nachfolgerknoten zusammensetzt.

- Definition "Binärbaum"
  - ➔ Jeder Knoten hat höchstens zwei Nachfolger.
  - ➔ Die kompakte Darstellung besteht aus einer Wurzel und zwei Teilbäumen.
  - ➔ Typdefinition:

```
type
tRefBinBaum = ^tBinBaum;
tBinBaum      = record
                   info: integer;
                   links :  tRefBinBaum;
                   rechts: tRefBinBaum
                end;
```

- Definition "Binärer Suchbaum"
  - ➔ Es werden in jedem Knoten Zahlen abgelegt, wobei gerade soviel Knoten wie Zahlen vorkommen, d. h. es gibt keinen leeren Knoten.
  - ➔ Für jeden Knoten gilt:
    Die Zahlen im linken Teilbaum sind alle kleiner als die Zahl in der Wurzel, die Zahlen im rechten Teilbaum sind alle größer als die Zahl in der Wurzel.
  - ➔ Die Operationen Suchen, Einfügen und Entfernen werden hier besser unterstützt als bei linearen Listen.

- Operationen auf Binärbäumen (Suchbäumen)
  - ➔ iteratives Suchen
    - Der gesuchte Wert muß mit der Wurzel verglichen werden, ist er gleich so ist die Wurzel bereits der gesuchte Knoten, ist er kleiner, so wird zum rechten Sohn verzweigt, ist er größer, so wird zum linken Sohn verzweigt und der Vorgang so lange wiederholt bis der Knoten gefunden ist oder der Vergleichsknoten ein Blatt ist, bzw. den gewünschten Nachfolger nicht hat.
    - while (zeiger <> nil) and not gefunden do
  - ➔ iteratives Einfügen
    - Zum Einfügen muß man zuerst die richtige Einfügestelle ermitteln
    - Dazu benötigt man zwei Hilfszeiger Vater und Sohn, die man in einer Schleife immer weitersetzt: while (sohn <> nil) and not gefunden do
    - Falls der einzufügende Knoten noch nicht im Baum enthalten ist muß er eingefügt werden. Dabei muß unterschieden werden ob der Vater = nil ist (das heißt, der Baum war leer), oder ob man den neuen Knoten als rechten oder linken Nachfolger einfügen muß.
  - ➔ Aufbau des Suchbaums
    - erfolgt durch den Aufruf der Einfügeroutine

- Definition "natürlicher Baum"
  - ➔ Ein natürlicher Baum ist ein durch iteriertes Einfügen einer Zahlenfolge in einen anfangs leeren Baum erstellter binärer Suchbaum.
- Definition "degenerierter Suchbaum"
  - ➔ Ein degenerierter Suchbaum entspricht einer linearen Liste und hat, wenn er n Knoten enthält, das heißt, wenn n Zahlen eingefügt wurden, die Höhe n.
- Definition "vollständiger Suchbaum"
  - ➔ Ein vollständiger Suchbaum hat die minimale Höhe $\log_2 n$, d. h. alle Blätter liegen auf einem oder auf 2 direkt aufeinanderfolgenden Niveaus.
- Definition "Balancierter Binärbaum"
  - ➔ Balancierte Binärbäume sind Binärbäume, die bei keiner Eingabereihenfolge entarten, da falls beim Einfügen ein Höhenunterschied >1 entsteht, dieser automatisch durch Rotation der Teilbäume ausgeglichen wird.

- Charakterisierung: "rekursive Funktion / Prozedur"
  - ➔ ruft sich selbst für eine immer kleiner werdende Problemgröße auf
  - ➔ nach endlich vielen Aufrufen muß eine Situation erreicht werden, in der kein rekursiver Aufruf mehr erfolgt
  - ➔ müssen eine Abbruchbedingung beinhalten
  - ➔ ist je nach Anwendung genauso effizient, wie eine iterative Lösung
  - ➔ die Programmkomplexität ist of erheblich geringer, d.h. es gibt weniger Fallunterscheidungen
  - ➔ ist nur dann effizient, wenn sie eine geringe Rekursionstiefe (Anzahl der ineinandergeschachtelten rekursiven Aufrufe) haben
  - ➔ für jeden rekursiven Aufruf wird die aktuelle Situation der Variablen
  - ➔ auf dem Laufzeitstack abgelegt, das kostet Zeit und Speicherplatz rekursive Datenstrukturen

- Rekursive Definition für eine lineare Liste
  - ➔ Eine Lineare Liste ist entweder leer oder besteht aus einem Knoten (Anfangsknoten), der auf eine lineare Liste zeigt.

- Rekursive Definition eines Binärbaums
  ➔ Ein Binärbaum ist entweder leer oder er besteht aus einem Anfangsknoten (der Wurzel), der auf zwei Binärbäume zeigt.

- Rekursive Operationen auf Listen
  ➔ sind nicht zu empfehlen, da sie eine hohe Rekursionstiefe haben.

- Rekursive Operationen auf Bäumen
  ➔ Suchen, Einfügen
    - Besser als die iterative Variante, wenn man davon ausgeht, daß die Höhe des Baumes logarithmisch beschränkt ist, denn dann ist die Rekursionstiefe ebenfalls logarithmisch beschränkt.
  ➔ Durchlaufen eines Baumes in Hauptreihenfolge (preorder)
    - Wurzel bearbeiten
    - linken Teilbaum bearbeiten
    - rechten Teilbaum bearbeiten
  ➔ Durchlaufen eines Baumes in symmetrischer Reihenfolge (inorder)
    - linken Teilbaum bearbeiten
    - Wurzel bearbeiten
    - rechten Teilbaum
  ➔ Iteratives Durchlaufen eines Baumes ben÷tigt eine Hilfsdatenstruktur, in der die noch zu verarbeitenden Teilbaumwurzeln abgelegt werden m³ssen

- Empfehlungen zur Verwendung von Rekursion / Iteration
  ➔ Gibt es für das betrachtete Problem eine einfache iterative L÷sung, so ist diese zu verwenden.
  ➔ Benötigt die iterative Variante einen Hilfsstapel, dann ist die rekursive Lösung vorzuziehen.
  ➔➔ Es ist also immer die Variante mit der geringeren Programmkomplexität zu verwenden.

KE5

- Was ist ein Modul?
  - ➔ Ein Modul ist eine in sich geschlossene Einheit, die aus logischer Sicht eine klar abgegrenzte und präzise definierte Teilaufgabe innerhalb eines Programms erfüllt.
  - ➔ Diese Einheit besteht aus einer Datenstruktur (Daten, Datentypen, Variablen + Konstanten) und den zugehörigen Operationen (Prozeduren und Funktionen)
  - ➔ Die strukturelle Mächtigkeit von Modulen liegt oberhalb von Prozeduren. Die Strukturierung von Prozeduren ist auf die operationale Ebene beschränkt, die von Modulen beinhaltet Daten und Operationen.
    Das heißt: Strukturierung (Prozedur) $\leq$ Strukturierung (Modul)
  - ➔ In TURBO-PASCAL wird ein Modul durch eine UNIT realisiert.
  - ➔ Zu einem Modul gehört die Schnittstelle (interface), die Implementation und die Initialisierung.
- Wie kann man ein Modul sinnvoll einsetzen?
  - ➔ Man kann ein Modul in einer Bibliothek ablegen, so daß die Module mit ihren Funktionen und Prozeduren von mehreren Programmen verwendet werden können.
- Was ist das Geheimnisprinzip?
  - ➔ Unter dem Geheimnisprinzip versteht man das Prinzhip, daß das "wie" etwas realisiert wird, dem Benutzer verborgen bleibt.
  - ➔ Unter dem Geheimnisprinzip versteht man das Verbergen der Realisierung eines Algorithmus vor dem Benutzer

- Was sind die Vorteile der modularen gegenüber der prozeduralen Programmierung?
  - ➔ Das Modul ist unabhängig von seinen Benutzern, daraus folgt:
    - leichte Änderbarkeit (Änderungen können lokal gehalten werden)
    - höhere Sicherheit, da der Benutzer nur über die in der Schnittstelle definierten Prozeduren und Funktionen zugreifen kann.
  - ➔ bessere Arbeitsteilung beim Programmieren
  - ➔ Wiederverwendbar als Softwarebaustein in anderen Programmen
  - ➔ Schnittstelle dient gleichzeitig zur Dokumentation
  - ➔ weitergehende Strukturierung. Dies ist besonders vorteilhaft für große Programme, da hier Daten und zugehörige Operationen eine Einheit bilden.

- Wie ist eine UNIT in TURBO-PASCAL aufgebaut?

  ➔ unit Unitname;
  interface; *Schnittstellenbeschreibung mit den für den Benutzer relevanten Operationen und Daten (Exportschnittstelle)*
  type ...; *Hier werden die Datentypen definiert, die im aufrufenden Programm bekannt sein müssen*
  procedure xy (...); *Hier werden die Schnittstellen der Prozeduren und*
  function ab (...); *Funktionen definiert, die der Benutzer verwenden darf.*
  Die Semantik der Operationen muß dem Benutzer
  ... *bekannt sein.*

  implementation; *Modulimplementation mit den Realisierungsdetails, die der Benutzer nicht sieht.*
  type ...; *Typdefinition interner Typen*
  var .... ; *Deklaration interner globaler Variablen*
  procedure xy; *Implementierung von internen und Export-Prozeduren*
  function ab; *Implementierung von internen und Export-Funktionen*

  begin *Initialisierung der verwendeten Datenstrukturen*
  end

- Wie kann man Funktionen und Prozeduren einer UNIT importieren, so daß man sie in einem Programm verwenden kann?
  ➔ Im declaration part des Programmes muß als erste Anweisung, also noch vor der Konstantendefinition, die Unit mit der Anweisung "uses Unitname" bekannt gemacht werden. Alle Datentypen, Funktionen und Prozeduren, die im interface-Bereich des Moduls aufgeführt sind, können dann im Programm verwendet werden. Um Namenskonflikte zu vermeiden werden sie mit "Unitname.Funktions-/Prozedur-/Datentypname" angesprochen.

- Definition: "Alphabet"
  ➔ Ein Alphabet ist eine endliche Menge von unterscheidbaren Zeichen einer Schriftsprache.
- Definition: "Schriftsprache"
  ➔ Eine Schriftsprache ist eine natürliche Sprache (Deutsch, Englisch, Französisch,...) in geschriebener Form oder eine künstliche Sprache (Programmiersprache, selbstdefinierte Sprache).
- Definition: "Sprache"
  ➔ Jede Menge $A \subseteq \Sigma^*$ heißt Sprache über Σ. Es gilt $\emptyset \subseteq \Sigma^*$. Also ist auch $\emptyset$ eine Sprache über Σ, die leere Sprache.
- Definition: "Wort"
  ➔ Ein Wort ist eine endliche Folge von Zeichen eines Alphabets.
- Definition: "leeres Wort"
  ➔ Das leere Wort ε (Epsilon) ist ein Wort ohne Zeichen
- Gleichheit von Worten
  ➔ Zwei Worte u und v sind gleich, d.h. u = v, wenn sie gleich lang sind und zeichenweise übereinstimmen.
- Länge eines Wortes
  ➔ Die Länge eines Wortes ist die Anzahl der Zeichen des Wortes.

- Was ist das allgemeine Konzept von Syntaxdiagrammen bzw. einer Grammatik?
  - ➔ Sie beschreiben mit Hilfe einer endlichen Menge von Regeln eine unendliche Menge von Ausprägungen.
  - ➔ Es sind Konsturktionsvorschriften zur Erzeugung von Sprachen.
- Definition: "kontextfreie Grammatik"
  - ➔ Eine Grammatik G=(N,T,P,S), wobei N die Menge der Nichtterminalsymbole, T die Menge der Terminalsymbole, P die Menge der Regeln und S das Startsymbol ist, und N, T, P paarweise disjunkt sind, heißt kontextfrei, wenn alle Ersetzungen eines Nichtterminalsymbols unabhängig vom Kontext sind, in dem dieses Nichtterminalsymbol steht.
  - ➔ mit einer Grammatik kann man Worte einer Sprache erzeugen
- Regeln in Backus-Naur-Form (BNF)
  - ➔ Links von '::=' steht ein Nichtterminalsymbol und rechts davon die Ersetzungsalternativen.
- Definition: "Syntaxdiagramm"
  - ➔ Mit einem Syntaxdiagramm kann man Worte einer Sprache erzeugen.
  - ➔ Ein Syntaxdiagramm besteht aus Rechtecken (Nichtterminalen), Kreisen (Terminalen) und Pfeilen.

- Definition: "Ableitung"
  - ➔ einmalige Anwendung einer Regel: Ableitungsschritt
  - ➔ Eine Ableitung ist der gesamte sukzessive Ersetzungsvorgang zur Erzeugung eines Wortes w.

- Definition "Prozedurale Zerlegung"
  - ➔ Die Prozedurale Zerlegung bildet das Grundgerüst für ein Programm (meist EVA-Prinzip).
  - ➔ Wirth: Methode der schrittweisen Verfeinerung
    - ▫ Zerlege die Aufgabe in Teilaufgaben.
    - ▫ Betrachte jede Teilaufgabe für sich und zerlege sie wieder in Teilaufgaben bis diese so einfach geworden sind, daß sie programmtechnisch formuliert werden können.
  - ➔ fortschreitende Zerlegung und Präzisierung.
  - ➔ Die Verfeinerung erfolgt funktionsorientiert (operations- und kontrollflußorientiert)

- Charakterisierung "Iterative Vorgehensweise"
  - ➔ Es kann sein, daß sich eine Zerlegung als unbrauchbar erweist und sie daher zurückgenommen werden muß, d.h. man muß manchmal auch wieder einen Schritt zurückgehen, oder gar von vorne anfangen.

- Definition "Prinzip der Abstraktion"
  - ➔ Die Verfeinerung erfolgt vom allgemeinen zum Speziellen, d.h. von einer abstrakten Formulierung zu einer genauen Spezifikation.
  - ➔ Die Lösung einer Aufgabe wird betrachtet, ohne daß alle Realisierungsdetails vollständig bekannt sind. Es werden nur die Details berücksichtigt, die zum Übergang zur nächsten Abstraktionsstufe notwendig sind.

- Definition: "Top-Down-Entwicklung"
  ➔ Schrittweise Verfeinerung, ausgehend von der Aufgabenstellung wird solange Top Down (von oben nach unten) in leichtere Teilaufgaben zerlegt, bis man Operationen vorliegen hat, die leicht als Prozedur oder Funktion formuliert werden können.

- Definition: "Backtracking-Verfahren"
  ➔ Beispiel: Suche eines Weges durch ein Labyrinth. Immer zuerst links gehen, falls man in einer Sackgasse landet, wieder bis zur Gabelung zurück und einen anderen Weg ausprobieren.

- Definition: "Pseudo-Code"
  ➔ Unter Pseudo-Code versteht man umgangssprachlich formulierte Programmstücke. Sie werden oft für Top-Down-Zwischenschritte verwendet.

KE6

- Semantik
  - ➔ eines Programms
    - ▫ setzt sich zusammen aus der Semantik der einzelnen Anweisungen
    - ▫ dazu muß man die Semantik der Programmiersprache kennen
  - ➔ von Programmiersprachen
    - ▫ wurde früher umgangssprachlich erklärt
    - ▫ inhaltliche Bedeutung einer Sprache
  - ➔ Quellcode und Zielprogramm müssen semantisch die gleiche Bedeutung haben
  - ➔ Die Aussage über die Semantik der Schleife ist nicht allein durch die Invariante gegeben, die Terminierung der Schleife muß auch mit einbezogen werden.

- Definition "denotationale Semantik" (Funktionensemantik)
  - ➔ Die Semantik ist eine Abbildung der Eingabemenge auf die Ausgabemenge. Es wird nur festgelegt, was berechnet wird und nicht wie es berechnet wird.
  - ➔ Zwei unterschiedliche Programme, die dieselbe Funktion berechnen haben in diesem Sinne dieselbe Semantik.
- Definition "operationale Semantik" (Interpretersemantik)
  - ➔ Die Semantik ist eine Folge von semantisch präzise definierten Berechnungsschritten, welche Eingabedaten in die Ausgabedaten überführt. Es wird festgelegt wie die Berechnung der Funktion abläuft. Zwei unterschiedliche Funktionen, die dieselbe Funktion berechnen, haben somit eine unterschiedlliche Semantik.
- Definition "axiomatische Semantik" (Prädikatensemantik)
  - ➔ Die Semantik wird durch eine Menge von Aussagen über Ein- und Ausgabedaten bestimmt. Zum Beispiel mit Hilfe von Vor- und Nachbedingungen. Die Aussagen erlauben den Nachweis, daß ein gegebenes Problem vom Programm korrekt gelöst wird.
  - ➔ Die Semantik wird durch Angabe einer gültigen Programmformel {P} S {Q} beschrieben

- Definition "logische Implikation: $A_1 \Rightarrow A_2$"
  - ➔ A_1 impliziert (logisch) A_2, geschrieben $A_1 \Rightarrow A_2$, wenn A_2 mindestens in allen Zeilen der Wahrheitstabelle den Wert 1 animmt, in denen A_1 den Wert 1 hat.
  - ➔ A_1 ist höchstens dann wahr, wenn A_2 wahr ist.
  - ➔ Ist A_1 falsch, dann kann man über A_2 nichts aussagen, d. h. A_2 kann wahr oder falsch sein.
  - ➔ Ist A_2 falsch, dann ist auch A_1 falsch.
  - ➔ Ist A_2 wahr, kann man nichts über A_1 schließen, der Umkehrschluß $A_2 => A_1$ ist also falsch.

- Definition "logische Äquivalenz:$A_1 <=> A_2$"
  - ➔ A_1 und A_2 sind logisch äquivalent, wenn die Wahrheitstabellen von A_1 und A_2 gleich sind.
- Definition "Tautologie"
  - ➔ Ein logischer Ausdruck, der unabhängig von der Belegung der Variablen immer wahr ist, wird Tautologie genannt.
- Definition "Kontradiktion"
  - ➔ Ein logischer Ausdruck, der unabhängig von der Belegung der Variablen immer falsch ist, wird Kontradiktion genannt.

- Definition "Validierung"
  - ➔ Unter Validierung versteht man die Überprüfung einer allgemeinen Aussage mit positivem Ausgang

- Definition: "Verifikation"
  - ➔ Unter Verifikation eines Programmes versteht man den formalen Beweis der Korrektheit des Programmes.
  - ➔ Die Verifikation beinhaltet den Nachweis, daß die Spezifikation des Programmes mit der Semantik des Programmes übereinstimmt.

- Welche Quantoren gibt es ?
  - ➔ Existenzquantor ∃
    Das Prädikat gilt für mindestens ein (beliebiges) Element der zugrundeliegenden Menge.
    $\exists x \in Z:P(X)$ "Es gibt ein x aus der Menge der ganzen Zahlen, so daß P(x) gilt"
  - ➔ Allquantor ∀
    Das Prädikat gilt für alle Elemente der zugrundeliegenden Menge.
    $\forall x \in Z:P(X)$ "Für alle x aus der Menge der ganzen Zahlen ist P(x) wahr"
- Was versteht man unter einer freien bzw. gebundenen Variablen?
  - ➔ Wenn eine Variable x in einer prädikatenlogischen Formel verwendet wird, ohne daß bei der Festlegung des Wertbereichs für x ein Quantor vor x verwendet wird, spricht man von einer freien Variablen, ansonsten von einer gebundenen Variablen

- Programmformel {P} S {Q}
  - ➔ Ist vor Ausführung der Anweisungsfolge S das Prädikat P wahr und nach Ausführung der Anweisungsfolge das Prädikat Q wahr, dann ist {P} S {Q} eine gültige Programmformel.
  - ➔ Hierbei wird nicht unterschieden ob S terminiert oder nicht. Terminiert S nicht, so ist jedes Prädikat als Vor- und Nachbedingung wahr.
- Definition: "Totale Korrektheit"
  - ➔ Ein Programm S heißt total korrekt bezüglich der Vorbedingung P und der Nachbedingung Q, falls gilt: Für jede Eingabe, welche die Vorbedingung P erfüllt endet die Ausführung von S und die Ausgabe erfüllt die Nachbedingung Q.
- Definition: "Partielle Korrektheit"
  - ➔ Ein Programm S heißt partiell korrekt bezüglich der Vorbedingung P und der Nachbedingung Q, falls gilt: Für jede Eingabe, welche die Vorbedingung P erfüllt und für welche die Ausführung von S endet, erfüllt die Ausgabe die Nachbedingung Q.

- Hoare-Regeln
  - ➔ werden im Inferenzschema angegeben, oben stehen mehrere Prämissen, falls diese gültig sind, kann man die unten stehende Programmformel daraus folgern.
  - ➔ 1. Konsequenzregel : Regel der schwächeren Nachbedingung
  - ➔ 2. Konsequenzregel : Regel der stärkeren Vorbedingung
  - ➔ Null-Regel : Spiegelt die Semantik der leeren Anweisung wider. Für jedes Prädikat P ist {P} {P} eine gültige Programmformel.
  - ➔ 1. Bedingungsregel / Konditionsregel: if then else
  - ➔ 2. Bedingungsregel / Konditionsregel: if then
  - ➔ Zusammensetzungsregel: besagt, daß eine Anweisungsfolge S durch begin und end eingeschlossen werden kann.

- ➔ Sequenzregel / Konkatenationsregel / Kompositionsregel
- ➔ Zuweisungsregel: Wird in der Vorbedingung P jedes X durch expr ersetzt und ist dieses veränderte Prädikat eine gültige Vorbedingung, dann ist die Nachbedingung P nach Ausführung der Zuweisung X:= expr auch gültig.
- ➔ while-Regel: Die Bedingung B wird immer vor Ausführung der Schleifeniteration geprüft
- ➔ repeat-Regel: Die Bedingung B wird nach Ausführung des Schleifenrumpfes geprüft, das heißt der Schleifenrumpf wird mindestens einmal abgearbeitet.
- ➔ Man kann von jeder Hoare-Regel eine gültige Abwandlung bilden.

• Wie beweist man die Gültigkeit einfacher Programmformeln?
- ➔ mit den Hoare-Regeln

• Was ist eine Schleifeninvariante (while)?
- ➔ Eine Schleifeninvariante ist ein Prädikat, daß sowohl vor als auch nach jedem Schleifendurchlauf gilt, also auch nach Beendigung der Schleife.
- ➔ Die Schleifeninvariante gilt vor der Iteration.
- ➔ Gilt die Schleifeninvariante vor einer Schleifeniteration, so auch nach der Schleifeniteration.
- ➔ Die Schleifeninvariante P wird so gewählt, daß die Vorbedingung P und die Nachbedingung ($P \wedge \neg B$) die Wirkung der Schleife möglichst genau festlegen, also die Vorstellung, die der Programmierer beim Entwickeln der Schleife hatte, widerspiegelt.

• Regeln für eine geeignete repeat-Schleifeninvariante:
- ➔ Die Schleifeninvariante gilt nach der ersten Ausführung des Schleifenrumpfes.
- ➔ Gilt die Schleifeninvariante vor einer Schleifeniteration, so auch nach der Schleifeniteration.
- ➔ Die Schleifeninvariante Q wird so gewählt, daß die Vorbedingung P und die Nachbedingung ($Q \wedge B$) die Wirkung der Schleife möglichst genau festlegen, also die Vorstellung, die der Programmierer beim Entwickeln der Schleife hatte, widerspiegelt.

• Wie beweist man die partielle Korrektheit eines Programmes?
- ➔ Mit Hilfe der Hoare-Regeln. Mit der Zuweisungsregel wird ausgehend von der Nachbedingung des Programms für die einzelnen Anweisungen die jeweilige Vorbedingung ermittelt, bis man bei der Vorbedingung des Programms angelangt ist. Enthält das Programm Schleifen, so muß man hierfür die Schleifeninvariante bestimmen. Bei geschachtelten Schleifen geht man von innen nach außen vor.

• Wie beweist man die totale Korrektheit eines Programmes?
- ➔ Mit der Methode der schwächsten Vorbedingung kann man die totale Korrektheit in einem Schritt beweisen. Dieses Verfahren ist jedoch sehr aufwendig und deshalb nicht zu empfehlen.
- ➔ Wenn man bereits die partielle Korrektheit des Programmes bewiesen hat, muß man nur noch zeigen, daß das Programm auch terminiert, dann hat man auch die totale Korrektheit bewiesen.

KE7

- Terminierung eines Programms für eine bestimmte Eingabe
  - ➔ Eigenschaft, daß das Programm für diese Eingabe nach endlich vielen Schritten beendet ist.
- Terminierung eines Programms bezüglich einer Vorbedingung P
  - ➔ Eigenschaft, daß das Programm für alle Eingaben terminiert, welche die Vorbedingung erfüllen.
- Definition der Terminierungsfunktion τ für die while-Schleife: "{P} while B do S {P$\wedge\neg$B}"
  - ➔ τ: $Z^r \rightarrow Z$, mit Z: Menge der ganzen Zahlen und folgenden Eigenschaften:
    - w1) r: Anzahl der im Schleifenrumpf vorhandenen Variablen. $\tau(v_1,..,v_r)= t \in Z$
    - w2) Jede Ausführung von S dekrementiert den Funktionswert der Terminierungsfunktion bzw. solange P$\wedge$B erfüllt ist dekrementiert die Ausführung von S den Funktionswert, d.h. gilt P$\wedge$B, so folgt: $\{\tau(v_1,..,v_r)= t\}$ S $\{\tau(v_1,..,v_r)< t\}$
    - w3) Es gibt ein $t^* \in Z$ als untere Schranke der Terminierungsfunktion. P$\wedge$B => $\tau(v_1,..,v_r) \geq t^*$ gilt immer vor Beendigung der Schleife. Danach kann $\tau(v_1,..,v_r)< t^*$ gelten, es muß aber nicht so sein, das ist abhängig davon, wie man t^* gewählt hat.
  - ➔ Die Angabe einer Terminierungsfunktion ist eine hinreichende Bedingung für die Terminierung. Das heißt wenn es eine Terminierungsfunktion gibt, dann terminiert die Schleife auch; kann man keine Terminierungsfunktion finden, so kann man nichts über die Terminierung der Schleife aussagen, insbesondere die Umkehrung gilt nicht.
  - ➔ τ kann monoton wachsend sein, dann muß aber auch t^* eine obere Schranke sein.
  - ➔ τ kann eine partielle Funktion sein. Eigentlich ist es sogar immer so, denn es wird immer nur ein bestimmter Bereich der Integerwerte abgedeckt.

- Wie geht man vor, wenn man eine Terminierungsfunktion für ein einfaches terminierendes Programm angeben will ?
  - ➔ Man orientiert sich an den Programmvariablen, die im Schleifenrumpf verändert werden. Alle im Schleifenrumpf vorkommenden Variablen sind die Eingabeparameter der Terminierungsfunktion τ. Oft gibt es eine Variable, die in jedem Schleifendurchlauf um irgendeinen Wert dekrementiert wird, eine solche Variable eignet sich gut als Funktionswert, denn dann sind bereits w1 und w2 erfüllt.

- Wie kann man für ein nicht terminierendes Programm begründen, warum es nicht terminiert
  - ➔ Es gibt keinen direkten Beweis dafür, daß ein Programm nicht terminiert
  - ➔ Man kann jedoch für spezielle Eingaben zeigen, daß die Schleife nicht terminiert, da w3 nicht erfüllt ist.

- Argumente gegen die Programmverifikation
  - ➔ sehr komplex, aufwendig, je nach Programmgröße teilweise undurchführbar
  - ➔ Die Beweise sind dann meist noch komplexer als die Programme und auch in den Beweisen können dann Fehler auftreten.
- Argumente für die Programmverifikation
  - ➔ Das Beweisen von Programmen ist einfacher, wenn die Programme gut strukturiert sind, das heißt wenn sie aus mehreren Modulen zusammengesetzt sind. Sind erst einmal die einzelnen Funktionen und Prozeduren der Module bewiesen, kann man auch die komplexen Programme beweisen.
  - ➔ Teilweise kann man Programmbeweiser verwenden.
  - ➔ Wichtig für Weltraum- und Regelungstechnik, da dort auftretende Fehler schwere Auswirkungen haben können, nicht nur auf die Kosten.
- Zusammenhang Terminierung und Korrektheit
  - ➔ Hoare-Regeln => partielle Korrektheit
  - ➔ partielle Korrektheit + Terminierung => totale Korrektheit

- Was versteht man unter Testen ?
  - ➔ Testen ist ein gezieltes Suchen nach Fehlern und keine Fehlerbeseitigung.
  - ➔ Eingabe einer Auswahl möglichst charakteristischer oder repräsentativer Eingaben und Kontrolle der Ausgaben auf die Erfüllung der Nachbedingung.

- Faustregeln zum Testen:
  - ➔ Jedes Programm ist fehlerhaft, es sei denn man hat sich vom Gegenteil überzeugt.
  - ➔ Der Programmtest sollte möglichst nicht vom Programmautor durchgeführt werden.

- Welche Testverfahren gibt es und was sind ihre Merkmale?
  - ➔ Es gibt statische und dynamische Testverfahren, die beide nochmals untergliedert werden
  - ➔ Merkmale von statischen Testverfahren:
    - ▫ einfach durchzuführen und effektiv
    - ▫ Das Programm wird nicht ausgeführt
    - ▫ Die Korrektheit bzgl. der Spezifikation wird nicht bewiesen.
    - ▫ Beispiel : Review
      - Das Programm wird von einer Gruppe von Testern mit einer bestimmten Zielsetzung strukturiert gelesen
      - erkannt wird nicht nur fehlerhaftes Verhalten, sondern die Fehlerursache selbst
    - ▫ Beispiel : statische Programmanalysatoren
      - ergänzen bzw. verbessern die Programme automatisch
      - sind als Ergänzung zum Compiler denkbar
  - ➔ Merkmale von dynamischen Testverfahren
    - ▫ Das Programm wird mit konkreten Testdaten ausgeführt.
    - ▫ Das Programm wird in der realen Umgebung getestet
    - ▫ Es sind Stichprobenverfahren, welche die Korrektheit eines Programms nicht beweisen können
    - ▫ Sie werden in der Praxis am meisten verwendet

➔ Weitere Unterteilung der dynamischen Testverfahren
- strukturorientierter Test ⇔ White-Box-Test
  - Die Testfälle werden kontroll- oder datenflußbezogen abgeleitet.
  - Die Testfälle basieren auf der internen Struktur des Programmes, diese muß bekannt sein.
  - Die Spezifikation wird nicht berücksichtigt.
  - Es gibt auch statische strukturorientierte Testverfahren.
- funktionaler Test ⇔ Black-Box-Test
  - Die Programmspezifikation dient zur Herleitung der Testfälle, deshalb ist eine sehr präzise Spezifikation erforderlich
  - Die Testfälle werden meist 2-stufig mit Hilfe von Äquivalenzklassen definiert
  - Wie das Programm verwirklicht wurde ist egal
- diversifizierender Test ⇔ Grey-Box-Test
  - kann sowohl strukturorientierte als auch funktionale Ansätze beinhalten
  - ist also eine Mischung aus White- und Black-Box-Test
  - Die Ergebnisse verschiedener Programmversionen werden miteinander verglichen

- Datenflußorientierte Verfahren
  ➔ basieren auf dem Fluß der Daten durch das Programm
  ➔ Suche nach Fehlern, die durch falsche Benutzung von Daten entstehen, zum Beispiel
    - deklarierte aber nicht benutzte Variablen
    - benutzte aber nicht deklarierte Variablen

- Definition "Kontrollflußgraph"
  ➔ Ein Kontrollflußgraph ist ein gerichteter Graph mit G=(N, E, nstart, nfinal).
    - N: Knotenmenge
    - E: Menge der gerichteten Kanten
    - Knoten : Anweisungen des Programms
    - Die gerichteten Kanten (Zweige) geben den statischen Kontrollfluß wieder
    - Block: nichtleere Folge von Knoten
    - Pfad: Folge aus Knoten und Blöcken vom Start bis zum Endknoten

- Beispiele für kontrollflußorientierte Testverfahren
  ➔ Anweisungsüberdeckung (C_0-Test)
    - Die Testfälle müssen so gewählt werden, daß alle Anweisungen mindestens einmal ausgeführt werden, d.h. alle Knoten des Kontrollflußgraphen müssen mindestens einmal besucht werden.
    - Anweisungsüberdeckungsgrad = Anzahl besuchter Knoten / Anzahl aller Knoten
    - Es können nur nicht ausführbare Programmteile entdeckt werden.
    - geringe Anzahl von Eingabedaten, einfach
    - mangelhafte Testqualität, es werden nur etwa 20 % der Fehler gefunden

➔ Zweigüberdeckung (C_1-Test)
- alle Verzweigungen durch Selektion oder Iteration müssen mindestens einmal ausgewählt worden sein, d.h. alle Kanten des Kontrollflußgraphen müssen mindestens 1mal besucht werden
- beinhaltet die Anweisungsüberdeckung
- Zweigüberdeckungsgrad = Anzahl besuchter Zweige / Anzahl aller Zweige
- systematisches Erkennen von nicht-ausführbaren Programmteilen
- minimales Testkriterium, darf bei keinem Test fehlen
- ca. 25% der Fehler werden erkannt

➔ Pfadüberdeckung
- Bei einer vollständigen Pfadüberdeckung muß jeder Pfad, das heißt jeder Weg im Kontrollflußgraphen vom Startknoten bis zum Endknoten besucht werden. Dieser Test wäre dann umfassend. Wegen der sehr vielen bzw. oft unzählbaren Testfälle ist dieser Test jedoch unpraktikabel bis undurchführbar
- Der Boundary-Interior-Pfadtest ist eine abgeschwächte Version der vollständigen Pfadüberdeckung. Es ist ein gezieltes Testen von Schleifen. Hierbei werden folgende Testklassen unterschieden:
  - alle Pfade, die der Schleifenbedingung nicht genügen, also den Schleifenrumpf keinmal bearbeiten
  - alle Pfade, die genau einmal durch die Schleife gehen (Grenze = Boundary)
  - alle Pfade, die eine Schleifenwiederholung beinhalten, also zweimal den Schleifenrumpf bearbeiten (interior)
  - Bei geschachtelten Schleifen geht man von innen nach außen vor
- ca. 50% der Fehler werden erkannt

➔ Bedingungsüberdeckung
- Bedingungen in Schleifen und anderen Auswahlkonstrukten dienen zur Definition der Testfälle
- einfache Bedingungsüberdeckung (C_2-Test):
  - alle atomaren Prädikate müssen einmal den Wert True und einmal den Wert False annehmen
- Mehrfach-Bedingungsüberdeckung (C_3-Test):
  - erfordert die Überprüfung aller Wertkombinationen
- minimale Mehrfachbedingungsüberdeckung
  - Jedes Prädikat, egal ob atomar oder nicht wird zu beiden Wahrheitswerten ausgewertet.

- Definition "Testfall"
  ➔ Unter einem Testfall versteht man eine aus der Spezifikation oder dem Programm abgeleitete Menge von Eingabedaten einschließlich der zugehörigen Ergebnisse. Also eine Teilmenge mit der das Programm tatsächlich getestet wird.
- Definition "Testdaten"
  ➔ Testdaten sind Elemente aus der Testfallmenge.

- Beispiele für funktionale Testverfahren
  - ➔ Bilden von Äquivalenzklassen
    - werden auf Basis der Definitionsbereiche der Eingabe- bzw. Ausgabemenge gebildet
    - es ist eine bestimmte Auswahlstrategie zur Ermittlung der Testdaten erforderlich. Testfälle = Äquivalenzklasse + Auswahlstrategie
    - oft werden die Äquivalenzklassen intern noch weiter aufgeteilt
    - alle Werte einer Äquivalenzklasse verursachen ein bestimmtes identisches funktionales Verhalten.
    - für Ausgabeäquivalenzklassen müssen zuerst noch die zulässigen Eingaben ermittelt werden.

  - ➔ Zufallstest
    - Methode, aus den Äquivalenzklassen Testdaten zu generieren
    - zufällige Auswahl von Elementen aus der jeweiligen Äquivalenzklasse
    - nicht-deterministisches Erzeugen von Testfällen
    - Vor- und Nachteile
      +Regellosigkeit der Testdatenerzeugung
      - stellt keines der minimalen Testkriterien sicher
      - Probleme bei der Validierung einer großen Anzahl von Ausgaben

  - ➔ Fehlerorientierte Testverfahren,Tests spezieller Werte
    - Auswahl der Testdaten aufgrund einer konkreten Erwartungshaltung bezüglich möglicher Fehler.
    - Test von Werten, die allgemein als fehlerbehaftet bekannt sind
    - ZERO-Values-Kriterium: Division durch Null. Alle Variablen müssen während der Tests einmal mit dem Wert Null belegt werden. Analog ist für die Zeiger mit dem Wert nil zu verfahren.
    - Grenzwertanalyse: Die Grenzen von Äquivalenzklassen sind in der Regel sehr fehleranfällig und sollten deshalb abgetestet werden

- Effizienz von Algorithmen
  - ➔ abhängig von der Wahl einer geeigneten Datenstruktur, dem zur Ausführung benötigten Speicherplatz und der Laufzeit
  - ➔ Parameter zur Effizienzmessung
    - Anzahl der ausgeführten Additionen und Multiplikationen
    - Anzahl der Vergleiche
    - Anzahl der Zuweisungen
    - Anzahl der Funktions- und Prozeduraufrufe
  - ➔ allein die Verwaltung der For-Schleife benötigt 3 Operationen (Zuweisung, Vergleich und Übergang zum Nachfolger)
  - ➔ Laufzeit und Speicherplatz sind auch von der Eingabegröße abhängig
  - ➔ ist von der geschickten Organisation der Daten abhängig
  - ➔ Je nach der Struktur der Eingabe unterscheidet man
    - Verhalten im besten Fall (worst case)
    - Verhalten im Mittel (average case)
    - Verhalten im schlechtesten Fall (worst case)
  - ➔ Die Effizienz wird in Größenordnungen gemessen, zur Charakterisierung verwendet man das Landausche Symbol 'O', O-Notation

- Definition: "Größenordnung"
  - ➔ man sagt, g ist von der selben Größenordnung wie f, und schreibt dabei $g \in O(f)$ oder $O(g) \subseteq O(f)$ wenn gilt:
    $f:N \rightarrow R_+$; $O(f):=\{g: N \rightarrow R_+ \mid \exists c_1,c_0 >0, \forall n(g(n) \leq c_1 * f(n)+c_0)\}$
  - ➔ $g \in O(f)$ entspricht der Abschätzung von g nach oben
  - ➔ $f \in O(g)$ entspricht der Abschätzung von g nach unten
  - ➔ Die Größenordnung ist nur für große n gültig. Sie charakterisiert das Wachstum auf lange Sicht. Man kann dabei jedoch nicht angeben wie groß die n sein müssen

- Wie kann man bestimmte Algorithmen bezüglich Ihrer Laufzeit vergleichen?
  - ➔ Man kann sie in Ihrer Größenordnung vergleichen

- O-Kalkül: Rechenregeln für O(f)
  - ➔ $O(g) \subseteq O(f) \Rightarrow g \in O(f)$
  - ➔ $O(g) \subset O(f) \Rightarrow g \in O(f) \wedge f \notin O(g)$
  - ➔ $O(g) \subseteq O(f)$: g wächst höchstens so schnell wie f, f mindestens so schnell wie g
  - ➔ $O(g) \subset O(f)$: g wächst langsamer als f, f schneller als g

- Wie ist die Reihenfolge der Größenordnung?
  - ➔ $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq ... \subseteq O((1+\varepsilon)^n) \subseteq O(2^n) \subseteq O(3^n) \subseteq ... \subseteq O(n!) \subseteq O(n^n))$
  - ➔ log n: logarithmisch
  - ➔ n: linear
  - ➔ n log n
  - ➔ $n^2, n^3, ...$: quadratisch, kubisch,... allgemein: polynomiell
  - ➔ 2^n: exponentiell

- Was ist ein praktikabler Algorithmus?
  - ➔ Ein Algorithmus dessen Laufzeit durch ein Polynom beschränkt ist, am besten O(nlogn)
  - ➔ Es gibt jedoch auch komplexe Algorithmen, die man nicht mit einer kürzeren als der polynomiellen Laufzeit lösen kann.