Herzlich willkommen zum Informatik-Studium

an der RWTH AACHEN !

Vorlesung Programmierung WS 2000/01

Studiengang Diplom-Informatik

- **Grundstudium**
  - 1. bis 4. Semester
  - ca. 80 Semesterwochenstunden

Informatik I Programmierung Datenstrukturen & Algorithmen	**Informatik II** Rechnerstrukturen Systemprogrammierung	**Informatik III** Berechenbarkeit & Komplexität Automatentheorie & Formale Sprachen
Mathematik I Analysis Differenzialgl. & Numerik Wahrscheinlichkeitsth. & Statistik	**Mathematik II** Lineare Algebra Diskrete Strukturen Mathematische Logik	
Anwendungsfach	**Proseminar**	**Elekt. Grundlagen HW-Praktikum**	**Software-Praktikum**

Teilnehmerkreis

- **Diplom-Informatik**
- **Technische Redaktion mit Schwerpunkt Informatik**
- **Mathematik mit Nebenfach Informatik**
- **Lehramt Mathematik**
- **Lehramt Informatik**

Inhalt, Ziele

Die VL Programmierung

- **In der Informatik unterscheiden wir zwei komplementäre Aspekte:**
  - Beschäftigung mit dem Computer als einer Maschine, die aus ***Hardware***-Teilen zusammengesetzt ist,
  - Beschäftigung mit dem Computer als einer Maschine, auf der Programme (***Software***) ablaufen.
- **Diese Veranstaltung behandelt den *Softwareaspekt*:**
  - Was ist ein Programm?
  - Was sind grundlegende Programmierkonzepte
  - Wie konstruiert (entwickelt) man ein Programm?
  - Welche Programmier-Paradigmen kennen wir?
- **Weitere Veranstaltungen zu Software im Grundstudium**
  - Datenstrukturen und Algorithmen (2. Semester)
  - Systemprogrammierung (3. Semester)
  - Programmierpraktikum im Grundstudium (3. oder 4. Semester)

Inhalt, Ziele

Ziel der LV "Programmierung"

- **Entwicklung von Programmen:**
  - Was ist ein ***Programm***?
  - Was ist eine ***Programmiersprache***?
  - Was sind elementare ***Programmierkonzepte***?
  - Wie können wir ein Programm schrittweise aus ***Komponenten*** entwickeln?
  - Wie bringen wir es auf den ***Rechner***?
  - Welche unterschiedlichen Arten von ***Programmiermodellen*** gibt es?
- **Erlernen grundlegender Programmiertechniken**
  - Software-Ingenieur ⟺ Künstler
- **Dazu bedienen wir uns**
  - der imperativen Programmiersprache ***Modula-3***
  - der logischen Programmiersprache ***Prolog***
  - und der funktionalen Programmiersprache ***LISP***

Inhalt, Ziele

Weitere Ziele

- **Verstehen, was Informatik ist**
  - Begriffsklärung
  - Was sind die ***Hauptaufgaben*** der Informatik
- **Fähigkeit erwerben, in Gruppen zu arbeiten**
  - ***Teamfähigkeit*** und ***Kommunikation*** sind wichtig
  - An der Hochschule und besonders später im Arbeitsleben
- **Sich an die Arbeitsweise an der Hochschule zu gewöhnen**
  - Lehrveranstaltungen sind ***Angebote***
  - Sie entscheiden, was Sie von diesen Angeboten wahrnehmen wollen
  - Lernen, ***selbständig*** zu Lernen

Inhalt, Ziele

Wie wollen wir das erreichen?

- **"Programmierung" ist eine dreiteilige Lehrveranstaltung**
  - Vorlesung
  - Diskussion
  - Gruppenübung
- **Weshalb Vorlesung?**
  - kompakte Vermittlung von Wissen und Erfahrung
  - Möglichkeit zur Abstimmung und Rückkopplung
- **Weshalb Übung?**
  - das Gelernte überprüfen und vertiefen
  - in der Gruppe arbeiten
  - etwas und sich selbst darstellen lernen
  - Handwerkszeug handhaben lernen
  - Hilfe zur Selbsthilfe

Nutzen Sie diese Angebote!
Es ist ihre Lehrveranstaltung

Orientierung

Was läuft wo?

- **In der Vorlesung:**
  - Einführung in die Programmierung
  - Begriffe, Konzepte und Beispiele
  - Ergänzende und vertiefende Themen (Exkurse)
- **In der Diskussion:**
  - Vorstellen von Themen, die für alle relevant sind (z.B. Arbeiten mit einer Programmierumgebung)
  - Fragen zur Vorlesung
- **In der Gruppenübung**
  - Diskussion von speziellen Aspekten der Übungsaufgaben
  - Präsentation von Lösungen

Orientierung

Spezialist und Novize

- **Viele haben bereits Erfahrungen ("Spezialisten")**
  - im Umgang mit Rechnern
  - im Programmieren mit einer Programmiersprache
  - aus eigenem Interesse am Thema oder durch Kurse am Gymnasium
- **Andere haben keine oder nur wenig Vorkenntnisse ("Novizen")**
  - alles ist neu und ungewohnt
  - Angst, schlechter als die "Spezialisten" zu sein

→ Vorkurs Informatik

Diese Angst ist unbegründet !

Wir erarbeiten systematisch das Gebiet der Programmierung!

Orientierung

Wer macht was?

- **Vorlesung**
  - Prof. H. Lichter
  - Raum: E2 - 6210
  - Sprechstunde:
    - Mi, 11:00 - 12:00
- **Gruppenübung**
  - viele studentische Hilfskräfte als Tutoren
- **Diskussion**
  - A. Nowack E1-4114a
    T. von der Maßen E2-6207
    M. Schnizler E2-6207
  - Di, 15:45-17:15, Raum: Ro

Sprechstunde: Mo, 10:45-11:45
Sprechstunde: Mi, 13:00-14:00
Sprechstunde: Do, 13:00-14:00

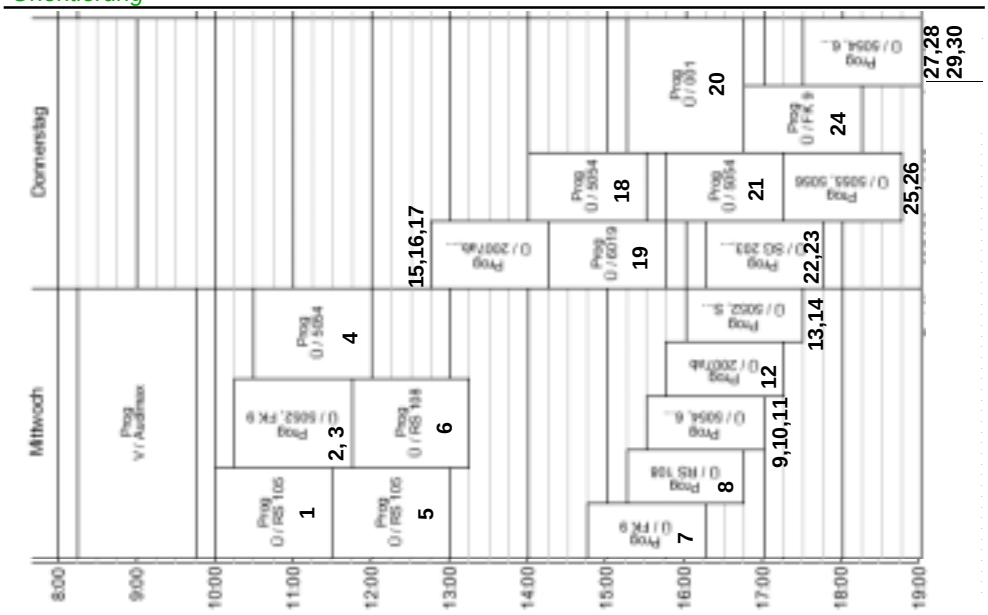
Orientierung

Termine

- **Vorlesung**
  - Mittwoch: 08:15 - 09:45 Audimax
  - Freitag: 08:15 - 09:45 Audimax
- **Diskussion**
  - Dienstag: 15:45 - 17:15 Ro — Beginn: 31. Oktober
- **Gruppenübungen**
  - Mittwoch: 14 Übungsgruppen — Beginn: 8. November
    Donnerstag: 16 Übungsgruppen — Beginn: 2. November
- **Zusatzübung für Studierende Technische Redaktion**
  - Dienstag: 08:15 - 09:45 AH IV — Beginn: 31. Oktober

Orientierung

Termine der Übungsgruppen



Orientierung

Wie zu was anmelden?

- **Anmeldung zu den Gruppenübungen**
- **Listen für die einzelnen Übungsgruppen hängen aus:**
  - Gebäude E2, Eingangsbereich Mies-van-der-Rohe-Str.
  - ab heute
  - bis ***Montag 23.10.00, 17:00***
- **Bitte die Listen gleichmäßig ausfüllen !**
  - Bitte nur den vorgesehenen Platz verwenden
- **Übungen werden in Kleingruppen bearbeitet und abgegeben:**
  - mindestens zwei, maximal drei Personen pro Kleingruppe, "Einpersonengruppen" werden nicht akzeptiert!
  - Keine Ausländergruppen
  - Frauengruppen

Orientierung

Literatur zur Vorlesung

- **Diese Vorlesung stützt sich in großen Teilen auf die folgenden Materialien:**
  - Hans-Jürgen Appelrath, Jochen Ludewig: "***Skriptum Informatik - eine konventionelle Einführung***", B.G. Teubner Stuttgart,1995 .
  - Robert W. Sebesta: "***Concepts of Programming Languages***". Benjamin Cummings, 2. Auflage, 1993.
- **Modula-3**
  - László Böszörményi, Carsten Weich: "***Programmieren mit Modula-3 - Eine Einführung in stilvolle Programmierung*** ", Springer Verlag,1995.
  - Samuel P. Harbison, "***Modula-3***", Prentice Hall, 1992
- **PROLOG, LISP**
  - L. Sterling, E. Shapiro: "***The Art of PROLOG***", MIT Press, 1994.
  - Berthold K. Horn, Patrick H. Winston, "***LISP***", Addison-Wesley, 1997.

Orientierung

Unterlagen zur Vorlesung

- **Stehen im "world wide web" zur Verfügung**
  - ***http://programmierung.informatik.rwth-aachen.de***
    - Neuigkeiten,
    - Folien der Vorlesungen,
    - Übungsblätter,
    - Lösungen

Orientierung

Was erwarten wir von Ihnen?

- **Dies ist *Ihre* Lehrveranstaltung!**
- **Mitarbeit**
  - die LV ist keine ***Beschäftigungstherapie***
  - ohne Arbeit keine ***bleibenden Ergebnisse***
  - ohne ***aktive Mitarbeit*** keine Erkenntnis
- **Gruppenarbeit**
  - ohne Gruppenarbeit keine ***Qualifikation*** zur Softwareentwicklung
  - ohne Gruppenarbeit wird das Studium ***zäh bis erfolglos***
- **Rückkopplung:**
  - über Inhalte
  - über Formen
  - über Probleme

Orientierung

Prüfung !

- **Die Diplom-Prüfungsordnung (DPO) regelt, welche Prüfungen Sie ablegen müssen.**
- **Vordiplomprüfung**
  - Fachprüfung "Informatik I"
    - LV "Programmierung“ 1. Semester
    - LV "Datenstrukturen“ 2. Semester
- **Zulassung:**
  - Übungsschein "Programmierung"
    - Voraussetzung zur Vordiplomprüfung

Diesen Übungsschein sollten Sie in dieser Veranstaltung erwerben!

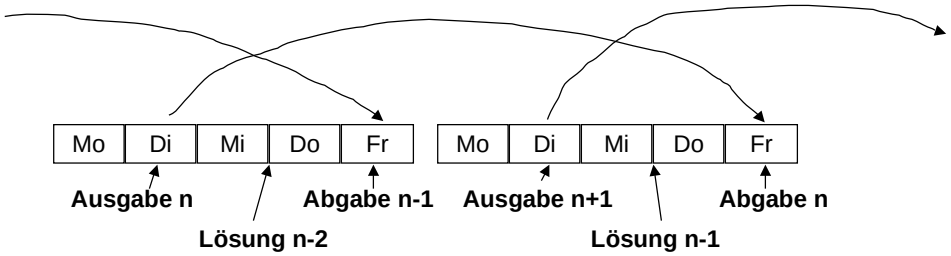
Orientierung

Prüfungsmodalitäten

- **Wer erhält einen Übungsschein?**
  - Alle die, die die Übungen erfolgreich bearbeitet und
  - die am Semesterende angebotene ***Übungsscheinklausur*** bestanden haben.
- **Erfolgreiches Bearbeiten der Übungen**
  - ***50*** % der erzielbaren Punkte der ***Übungsblätter 1 - 6***
  - ***50*** % der erzielbaren Punkte der ***Übungsblätter 7 - 13***
  - ***Vorrechnen*** **einer Lösung in den Übungsgruppen**
- **Übungsscheinklausur**
  - Dauer: 2 Stunden
  - Termin: Fr. 16. Februar 2001
  - Zeit: 16:00 - 18:00
  - Ort: Viele Hörsäle (wird noch bekannt gegeben)
  - Die Klausur geht über den ganzen Stoff der Vorlesung.

Orientierung

Übungsbetrieb



- **Ausgabe der Übungsblätter**
  - Dienstags (im www und auf Papier)
- **Abgabe der Übungen**
  - Freitags (Gebäude E2, Erdgeschoss **bis 14:00**), teilweise auch elektronisch
- **Erstes Übungsblatt**
  - *Ausgabe: Dienstag, 24. Oktober*
  - *Abgabe: Freitag, 3. November*

!

Orientierung

Informationen zum Rechnerbetrieb - 1

- **Sie benötigen Zugang zu den Rechnern, um**
  - auf die "online" zur Verfügung gestellten Informationen zugreifen zu können
  - Programmieraufgaben lösen zu können
- **Rechner werden im sogenannten "Rechnerpool Informatik" zur Verfügung gestellt**
  - ca. 75 Rechner (Linux/Solaris/NT) Gebäude E1/E2
- **Öffnungszeiten**
  - Mo 9:00 - 19:00
  - Die - Do 9:00 - 21:00
  - Fr 9:00 - 18:00
  - Es werden Anträge für Benutzerkennungen ausgegeben.
    Einige haben diese bereits aus dem Vorkurs.

Orientierung

Informationen zum Rechnerbetrieb - 2

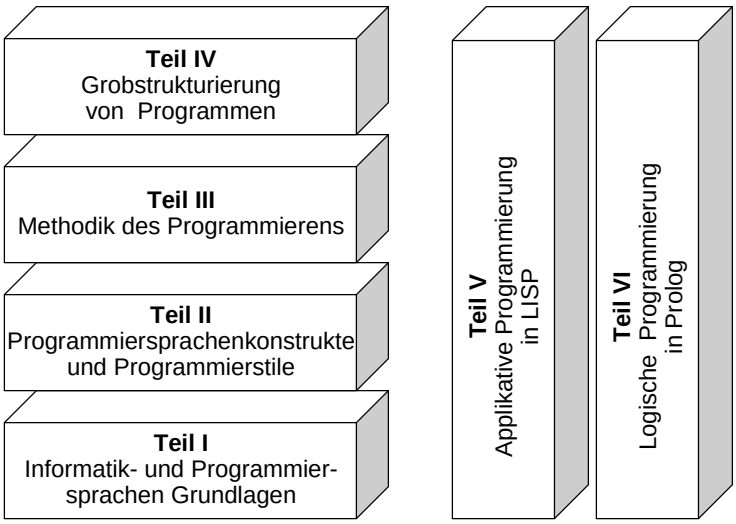
- **Folgende Zeiten sind für Hörer der Veranstaltung "Programmierung" reserviert:**

Montag:	**10.00 - 12.00, 15.00 - 19.00**
Dienstag:	**17.30 - 21.00**
Mittwoch:	**13.00 - 21.00**
Donnerstag:	**12.00 - 21.00**
Freitag:	**13.00 - 18.00**

Wie erhalten Sie eine Benutzerkennung

- **Benutzerkennungen für die Rechner des Informatik-Pools werden folgendermaßen vergeben:**
  - Melden Sie sich vorher bitte bei ***Rechnerberatung***.
  - Im ***Raum 4U15 (lila Raum)*** stehen speziell gekennzeichnete Rechner zur Verfügung, um einen Antrag für eine Benutzerkennung zu erstellen.
  - Wenn Sie alle Daten eingegeben haben, wird der Antrag bei der Rechnerberatung ausgedruckt.
  - Sie müssen diesen dann dort nur noch ***unterschreiben***!
  - Ihre Kennung ist am ***nächsten Tag*** verfügbar.

Gliederung der VL Programmierung



Gliederung

- **Teil I: Informatik- und Programmiersprachen-Grundlagen**
  - Historisches
  - Informatik als Wissenschaft, wichtige Begriffe
  - Sprache, Syntax und Semantik von Sprachen, Syntaxnotationen
- **Teil II: Programmiersprachenkonstrukte und Stile**
  - Funktionale Programmierung
  - Imperative Programmierung
  - Kontrollstrukturen
  - Datentypen
- **Teil III: Methodik des Programmierens**
  - Vorgehensweise bei der Programmierung
  - Notationen für die Dokumentation
  - Test von Programmen
  - Effizienzüberlegungen

Gliederung

- **Teil IV: Grobstrukturierung von Programmen**
  - Modularisierung und Module
  - vordefinierte Bausteine
  - Datenabstraktion
  - Objektorientierung
- **Teil V: Applikative Programmierung mit LISP**
  - Listen für Daten und Programme
  - Wertzuweisungen, -ermittlung, -interpretation
  - Listenverarbeitung
  - Systemfunktionen, Ausdrücke
- **Teil VI: Logische Programmierung mit Prolog**
  - Das logische Programmiermodell
  - Bestandteile und Aufbau eines Prolog-Programms
  - Unifikation und Resolution

Informatik-Grundlagen

- Klärung „Informatik“
- Geschichte der Informatik
- Algorithmus
- Software, Programm, Programmentwicklung
- Von-Neumann-Rechner

Was ist Informatik?

Klärung Informatik

- **Der Begriff "Informatik"**
  - ein ***Kunstwort*** , das zu Beginn der 60er Jahre zur Bezeichnung einer sich neu entwickelnden Disziplin geschaffen wurde. Es setzt sich aus Bestandteilen der beiden Worte ***Information*** und ***Mathematik*** zusammen. Darin kommt zum Ausdruck, dass Informatik die Wissenschaft von der Informationsverarbeitung ist, und eine große Nähe zur Mathematik hat. Heute wird der Begriff z.T. sehr anwendungsnah gebraucht und Synonym zur ‘Informationstechnik‘ verwendet.
- **Informatik ist die Wissenschaft**
  - von der ***systematischen*** Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von ***Computern*** (vgl. DUDEN Informatik, 1993)
- **Informatik versteht sich als Wissenschaft**
  - der Analyse, Konzeption und Realisierung von Systemen, die aus miteinander und mit ihrer Umwelt kommunizierenden Akteuren bestehen (vgl. Studienführer Informatik, RWTH Aachen, 1998)

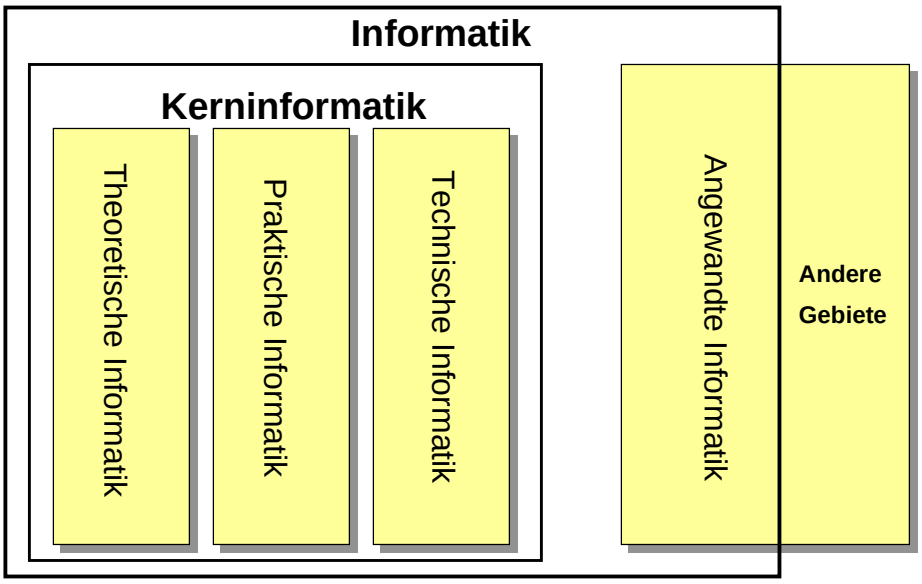
Klärung Informatik

Hauptaufgaben der Informatik

- **Hauptaufgabe der Informatik ist die Entwicklung**
  - ***formaler, maschinell ausführbarer Verfahren*** zur Lösung von Informationsverarbeitungsproblemen, die häufig als Teilprobleme komplexer Kommunikations- oder Organisationsprobleme auftreten.
- **Die Forderung der Durchführbarkeit mittels einer Maschine (i.a. eines Digitalrechners) bedingt,**
  - dass die zu verarbeitenden Informationen als ***maschinell verarbeitbare Daten*** dargestellt werden, und
  - dass die Lösungsverfahren bis ***ins Detail*** formal beschrieben werden.

Klärung Informatik

Gebiete der Informatik - 1



Klärung Informatik

Gebiete der Informatik - 2

- **Theoretische Informatik**
  - ***Formale Modelle*** zur Beschreibung und Untersuchung von Algorithmen, Computern, etc.
  - Teilgebiete: Formale Sprachen, Automatentheorie, Komplexitätstheorie etc.
- **Technische Informatik**
  - Funktioneller ***Aufbau von Computern***, Entwurf und Entwicklung von Rechnern, Geräten und Schaltungen
  - Teilgebiete: Rechnerarchitektur, VLSI-Entwurf etc.
- **Praktische Informatik**
  - ***Prinzipien und Techniken*** der Fundierung und Realisierung in Software (großer Programmsysteme!)
  - Teilgebiete: Softwaretechnik, Informationssysteme, Compilerbau, Betriebssysteme, Künstliche Intelligenz, Kommunikation/verteilte Systeme, Parallele Systeme, etc.

Klärung Informatik

Gebiete der Informatik - 3

- **Angewandte Informatik**
  - ***Anwendung*** der Methoden der ***Kerninformatik*** in anderen Wissenschaften
    - Entwicklung ***spezieller*** Verfahren und Darstellungstechniken
  - Bindestrich-Informatik:
    - Wirtschaftsinformatik, Medizin-Informatik, Bioinformatik, Rechts-Informatik
  - Grenzen zwischen ***Praktischer Informatik*** und ***Angewandter Informatik*** sind zum Teil fließend.
- **Das, was man unter Informatik versteht, kann man in solchen Definitionen jedoch nicht *endgültig* fassen. Schließlich ändert sich die Beschreibung der Definitionen einer Wissenschaft, wie in anderen Fällen auch, über die Zeit.**

Geschichte der Informatik

Geschichte - 1

- **Altertum–Mittelalter:**
  - Verwendung des ***Abakus*** (Brett mit verschiebbaren Kugeln) als Hilfsmittel für die vier Grundrechenarten.
- **9. JH.:**
  - Der arabische Mathematiker und Astronom ***Ibn Musa Al-Chwarismi*** schreibt das Lehrbuch "Kitab al jabr w' almuqabala" ("Regeln der Wiedereinsetzung und Reduktion"). Das Wort "***Algorithmus***" geht auf seinen Namen zurück.
- **1547: Adam Riese (1492–1559)**
  - veröffentlicht ein Rechenbuch, in dem er die ***Rechengesetze*** des aus Indien stammenden ***Dezimalsystems*** (5. Jh.n. Chr.) beschreibt. Im 17. Jahrhundert setzt sich das Dezimalsystem in Europa durch.
- **Wilhelm Schickard (1592–1635)**
  - konstruiert für seinen Freund Kepler (1571–1630) eine ***Maschine***, die addieren, subtrahieren, multiplizieren und dividieren kann. Sie bleibt unbeachtet.
- **1641: Blaise Pascal (1623–1662)**
  - konstruiert eine Maschine, mit der man ***sechsstellige Zahlen*** addieren kann.

Geschichte der Informatik

Geschichte - 2

- **1674: Gottfried Wilhelm Leibniz (1646–1716)**
  - konstruiert eine ***Rechenmaschine*** mit Staffelwalzen für die vier ***Grundrechenarten***. In diesem Zusammenhang befasst er sich auch mit der binären Darstellung von Zahlen.
- **1774: Philipp Matthäus Hahn (1739–1790)**
  - entwickelte eine mechanische Rechenmaschine, die ***erstmals zuverlässig*** arbeitet.
- **Ab 1818:**
  - Rechenmaschinen nach dem Vorbild der Leibnizschen Maschine werden serienmäßig hergestellt und dabei ständig weiterentwickelt.
- **1838: Charles Babbage (1792–1871)**
  - plant eine Maschine, die "***Analytical Engine***", bei der die Reihenfolge der einzelnen Rechenoperationen durch nacheinander eingegebene ***Lochkarten*** gesteuert wird.
- **1886: Hermann Hollerith (1860–1929)**
  - entwickelt in den USA elektrisch arbeitende ***Zählmaschinen für Lochkarten***, mit denen die statistischen Auswertungen der Volkszählungen vorgenommen werden.

Geschichte der Informatik

Geschichte - 3

- **1934: Konrad Zuse (1910–1995)**
  - beginnt mit der Planung einer ***programmgesteuerten Rechenmaschine***. Sie verwendet das binäre Zahlensystem.
- **1937: Die mechanische Anlage *Z 1* von Zuse ist fertig.**
- **1941: Die elektromechanische Anlage Z 3 von Zuse ist fertig.**
  - Dies ist der ***erste funktionsfähige programmgesteuerte Rechenautomat***. Das Programm wurde mit ***Lochstreifen*** eingegeben. Die Anlage verfügt über 2000 Relais und eine Speicherkapazität von 64 Worten a 22 Bit. Multiplikationszeit: etwa 3 s.
- **1944: Howard H. Aiken (1900–1973)**
  - erstellt in Zusammenarbeit mit der Harvard-University und der Firma IBM die teilweise programmgesteuerte Rechenanlage MARK I. Additionszeit 1/3 s, Multiplikationszeit: 6 s.
- **1946: J. P. Eckert und J. W. Mauchly**
  - stellen die ENIAC (Electronic Numerical Integrator and Automatic Calculator) fertig. Dies ist der erste ***voll elektronische Rechner*** (18.000 Elektronenröhren). Multiplikationszeit: 3 ms.

Geschichte der Informatik

Geschichte - 4

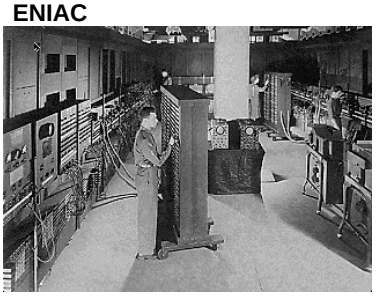
- **1946–1952:**
  - Auf der Grundlage der Ideen **John v. Neumanns** (1903–1957) (Einzelprozessor, Programm und Daten im gleichen Speicher; Von-Neumann-Rechner) und seiner Kollegen am Institute of Advanced Study at Princeton (H.H.Goldstine, A.W.Burks) werden weitere Computer in Universitätslabors entwickelt (“Pionierzeit”).
- **1949: M.V. Wilkes (University of Manchester)**
  - stellt mit der EDSAC (Electronic Delay Storage Automatic Calculator) den **ersten universellen Digitalrechner** (gespeichertes Programm) fertig.
- **Ab 1950:**
  - **Industrielle** Rechnerentwicklung und -produktion.

Historische Rechner

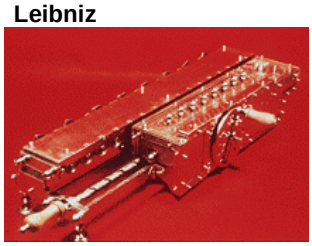
Geschichte der Informatik

Schickard

ENIAC



Leibniz



MARK1

Informelle Definition

Algorithmus

- **Ein Algorithmus ist ein Verfahren, welches**
  - in einem ***endlichen*** Text niedergelegt werden muss
  - ***effektiv*** ausführbar ist,
  - Elementaroperationen enthält, die durch die jeweilige Situation ***eindeutig*** bestimmt sind
  - ***Ein- und Ausgabe*** ermöglicht
  - durch eine (mechanisch oder elektronisch arbeitende) ***Maschine ausgeführt*** werden kann.
- **Anzahl und Ausführungszeit der Elementaroperationen sind beschränkt.**
- **Ein Algorithmus (Programm) wird durch eine Maschine schrittweise ausgeführt**
  - Die ausführende Instanz muss die Vorschrift ***interpretieren*** und ***korrekt*** ausführen.
  - Ein Algorithmus ***terminiert***, wenn er nach endlich vielen Schritten abbricht.

Algorithmus

Euklidischer Algorithmus - 1

Euklidischer Algorithmus

- Problem:
  - Man bestimme zu je zwei natürlichen Zahlen n und m den größten gemeinsamen Teiler ggt(n,m)

Algorithmus

Wiederhole:

❶ WENN n < m ist, DANN vertausche man n und m

❷ WENN m = 0 ist, DANN ist n der ggt(n,m) und man beende den Algorithmus

❸ WENN m≠ 0 ist, DANN bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

Algorithmus

Euklidischer Algorithmus - 2

n = 6
m = 10

Ablaufverfolgung (Trace)

für Überprüfung von Programmen

	n	m
	6	10
Erster Durchlauf		
da 6 < 10, so setzt man n = 10 und m = 6	10	6
da 6 ≠ 0 ist, gehen zu Schritt 3	10	6
r = 4. Man erhält also n = 6 und m = 4	6	4
Zweiter Durchlauf		
Da 6 ≥ 4 ist, gehe zu Schritt 2	6	4
Da 4 ≠ 0 ist, gehe zu Schritt 3	6	4
r = 2. Man erhält n = 4 und m = 2	4	2
Dritter Durchlauf		
Da 4 ≥ 2 ist, gehe zu Schritt 2	4	2
Da 2 ≠ 0 ist, gehe zu Schritt 3	4	2
r = 0. Man erhält n = 2 und m = 0	2	0
Abbruch		
Da 2 ≥ 0 ist, gehe zu Schritt 2	2	0
Da m = 0 ist, Programmende	2	0

Ergebnis: ggt(10, 6) =2

Algorithmus

Eigenschaften - 1

- **Abstraktion**
  - ein Algorithmus löst i. a. eine ***Klasse von Problemstellungen*** (z.B. Suchen eines Musters in einer Zeichenkette)
- **Finitheit**
  - statisch finit: ein Algorithmus besitzt eine ***endliche Länge***
  - dynamisch finit: während der Abarbeitung darf nur ***endlich viel Speicherplatz*** belegt werden
- **Terminierung**
  - terminierend: nach ***endlich vielen Schritten*** liegt ein Resultat vor
  - sonst ***nicht-terminierend*** (z.B. Steuerungsalgorithmen)
- **Determinismus**
  - deterministisch: zu jedem Zeitpunkt besteht ***höchstens eine*** Möglichkeit der Fortsetzung
  - nicht-deterministisch: an mindestens einer Stelle gibt es eine ***Wahlmöglichkeit*** für die Fortsetzung

Algorithmus

Eigenschaften - 2

- **Determiniertheit**
  - determiniert: bei ***gleichen Eingaben*** und Startbedingungen wird das ***gleiche Ergebnis*** erzielt
  - nicht-determiniert: es werden ***unterschiedliche Ergebnisse*** erzielt (z.B. Anwendung von heuristischen Methoden, syst. Probieren)
- **Bemerkung**
  - ein terminierender, deterministischer Algorithmus ist immer determiniert
  - ein terminierender, nicht-deterministischer Algorithmus kann determiniert oder nicht-determiniert sein.

Algorithmus

Terminierung - Nichtterminierung

- **Sei**
  - power : $N^+ \times N \rightarrow N$

$$\text{power}(a, b) = \begin{cases} 1 & \text{falls } b = 0 \\ a * \text{power}(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert*** im Definitionsbereich.

  - Wir weiten den Definitionsbereich aus:

$$\text{power2} : Z \times Z \rightarrow Z$$

$$\text{power2}(a, b) = \begin{cases} 0 & \text{falls } a = 0 \text{ und } b > 0 \\ 1 & \text{falls } a \neq 0 \text{ und } b = 0 \\ a * \text{power2}(a, b-1) & \text{sonst} \end{cases}$$

Dieser Algorithmus ***terminiert nicht*** für power2 (0,0) und power2 (a,b) mit $b < 0$ und $a \neq 0$)

Algorithmus

Anmerkung zur Nichtterminierung

Während wir die Nichtterminierung bei der ***manuellen Auswertung*** leicht feststellen können, fällt dies auf dem Rechner sehr viel schwerer. Solange der Rechner ***vor sich hin*** rechnet, können wir den Unterschied zwischen einem ***nicht-terminierenden*** und einem ***sehr langwierigen Algorithmus*** nicht feststellen.

Algorithmus

Fragen

- **Wie kann man aus einer Lösungsidee einen Algorithmus konstruieren?**
  - "schrittweise Programmentwicklung"
- **Wie kann man Algorithmen darstellen?**
  - "Flussdiagramme", Programme
- **Wie beweist man, dass ein Algorithmus tatsächlich das tut, was er tun soll?**
  - Verifikation: partielle Korrektheit
  - Termination
- **Wie "gut" ist ein Algorithmus?**
  - Speicherverbrauch, benötigte Zeit
  - Aufwandsabschätzungen

Algorithmus

Typische Problemklassen

- **Sortieralgorithmen**
  - Ordnen von Elementen
- **Suchalgorithmen**
  - Auffinden von Elementen
- **Algorithmen zur Verarbeitung von Zeichenfolgen**
  - Mustererkennung, Verschlüsselung, Komprimierung
- **Geometrische Algorithmen**
  - z.B. Schnittmenge geometrischer Objekte
- **Algorithmen für Graphen**
  - Suchen im Graph, kürzester Weg
- **Mathematische Algorithmen**
  - Rechnen mit Polynomen und Matrizen
- **Viele Anwendungsalgorithmen:** Kontrolle, Steuerung, Simulation

Standard-Algorithmen der Informatik

Software, Programm, Programmentwicklung

Überblick

- **Neben der Entwicklung der Hardware (Rechner)**
  - wurden seit 1955 ***Programmiersprachen*** entwickelt, um Algorithmen zu formulieren, damit sie von einem Rechner ausgeführt werden können.
- **Rechner "realisieren" Algorithmen,**
  - durch ***schrittweise Abarbeitung*** von ***Programmen***, die in einer Programmiersprache geschrieben sind.
- **Um einen Rechner zu programmieren,**
  - muss die ***Syntax*** und ***Semantik*** der verwendeten Programmiersprache bekannt sein.
- **In diesem Zusammenhang spricht man häufig auch von Software**
  - "***Software***" und "***Programm***" sind aber nicht dasselbe

Software, Programm, Programmentwicklung

Definition: Software

- **1. Definition: Software**
  - Informatik-Duden: Gesamtheit ***aller Programme***, die auf einer Rechenanlage eingesetzt werden können
    - ***Systemsoftware***: Programme die für den korrekten Ablauf einer Rechenanlage notwendig sind
    - ***Anwendungssoftware***: dient zur Lösung von Benutzerproblemen
- **2. Definition: Software**
  - IEEE Standard Glossary of Software Engineering Terminoloy
    - "Computer ***programs***, ***procedures***, and possibly associated ***documentation*** and ***data*** pertaining to the operation of a computer system."

Testfälle, Handbuch, Installations-anweisung etc.

Software, Programm, Programmentwicklung

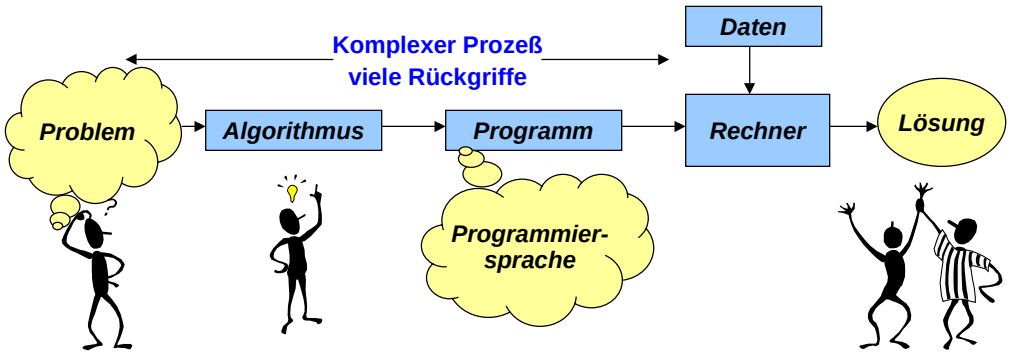
Definition: Programm

- **1. Definition: Programm in einer Programmiersprache**
  - Formulierung eines Algorithmus und der dazugehörigen Datenbereiche in einer Programmiersprache.
  - Ein Programm
    - ist ***exakt*** (formal) definiert
    - nimmt Bezug auf eine bestimmte Darstellung der ***Daten***
    - ist auf einer Rechenanlage ***ausführbar***
- **2. Definition: Programm**
  - IEEE Standard Glossary of Software Engineering Terminoloy
    - "A combination of ***computer instructions*** and ***data definitions*** that enable computer hardware to ***perform*** computational or control functions".

Software, Programm, Programmentwicklung

Programmentwicklung

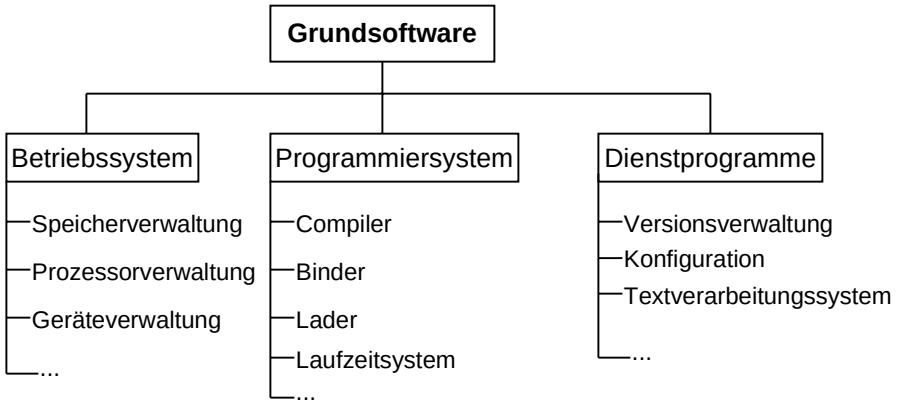
- **Unter dem Begriff Programmieren versteht man**
  - das **Lösen von Problemen** unter Zuhilfenahme eines **Rechners**



- **Programmieren ≠ Software-Entwicklung**
- **Programmieren ist *ein Teil* der Software-Entwicklung**

Software, Programm, Programmentwicklung

Grundsoftware (Systemsoftware)



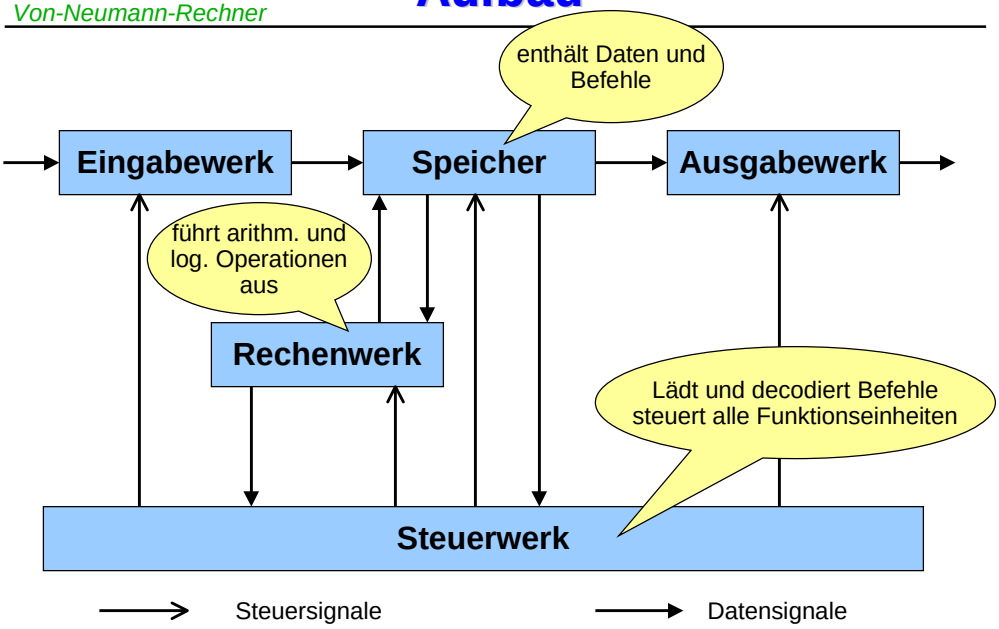
Von-Neumann-Rechner

Überblick

- **Definiert die wesentlichen Elemente eines Universalrechners**
  - Rechner soll nicht für eine ***bestimmte*** Problemklasse konstruiert sein
  - Zur Lösung des Problems muss ein ***Programm*** eingegeben und in den ***Speicher*** abgelegt werden
- **1946 von John von Neumann als Konzept für die EDVAC vorgeschlagen**
- **Fast alle heutigen Rechner basieren darauf und sind Weiterentwicklungen davon**
  - Nicht-von-Neumann-Rechner sind Gegenstand der Forschung
- **Diese Architektur prägt viele Programmiersprachen (imperative Programmiersprachen)**

Aufbau

Von-Neumann-Rechner



Prinzipien - 1

Von-Neumann-Rechner

- Der Rechner besteht aus *Steuerwerk, Rechenwerk, Speicher, Eingabewerk* und Ausgabewerk.
- Die Struktur des von-Neumann-Rechners ist *unabhängig* von den zu bearbeitenden Problemen. Zur Lösung eines Problems muss von außen das *Programm* eingegeben und im Speicher abgelegt werden.
- Programme, Daten, Zwischen- und Endergebnisse werden in *demselben Speicher* abgelegt.
- Der Speicher ist in *gleichgroße Zellen* unterteilt, die fortlaufend durchnumeriert sind. Über die Nummer (*Adresse*) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden.
- Aufeinanderfolgende Befehle eines Programms werden in *aufeinanderfolgenden* Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
- Durch *Sprungbefehle* kann von der Bearbeitung der Befehle in der gespeicherten Reihenfolge abgewichen werden.

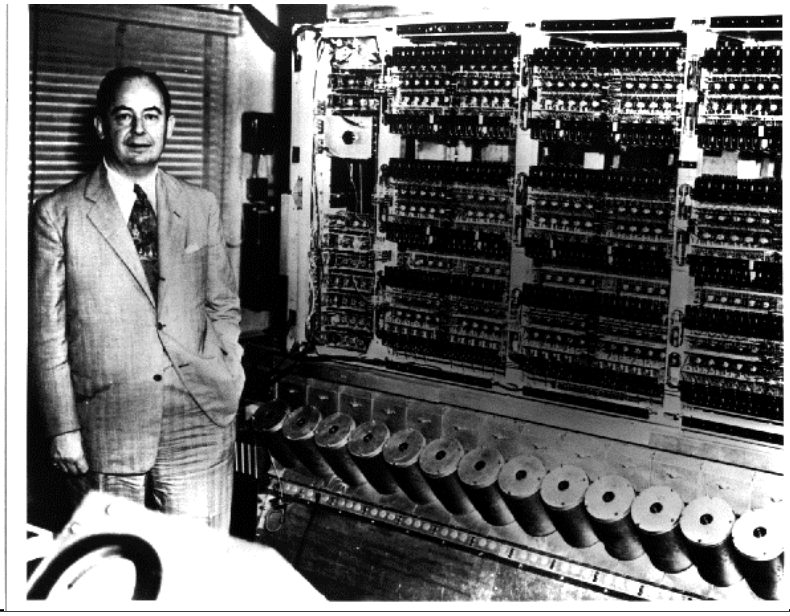
Prinzipien - 2

Von-Neumann-Rechner

- Es gibt zumindest
  - *arithmetische* Befehle wie Addieren, Multiplizieren usw.;
  - *logische* Befehle wie Vergleiche, logisches Nicht, Und, Oder usw.;
  - *Transportbefehle*, z.B. vom Speicher zum Rechenwerk und für die Ein-/Ausgabe;
  - bedingte *Sprünge*.
  - Weitere Befehle wie Schieben, Unterbrechen, Warten usw. kommen hinzu.
- Alle Daten (Befehle, Adressen usw.) werden *binär codiert*. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (*Decodierung*).

Historische Rechner - J. v. Neumann

Von-Neumann-Rechner



Was haben wir gelernt

- **Einordnung der Informatik als Wissenschaft**
  - Aufgabe der Informatik
  - Einteilung der Informatik
- **Wo kommt die Informatik her**
  - Entwicklung der Rechenmaschinen
- **Was versteht man unter einem Algorithmus**
  - Eigenschaften von Algorithmen
- **Was versteht man unter den zentralen Begriffen**
  - Software
  - Programm
  - Programmieren
- **Grober Aufbau eines von-Neumann-Rechners**

Glossar

- **Informatik:**
  - Begriff, Einteilung, Einordnung, Aufgaben, Geschichte
- **Algorithmus:**
  - Definition, Eigenschaften, Klassen von Algorithmen
- **Software**
- **Grundsoftware**
  - Betriebssystem, Programmiersystem, Dienst- und Hilfsprogramme
- **Programm**
- **Programmieren, Programmentwicklung**
- **Von-Neumann-Rechner:**
  - Speicher, Rechenwerk, Steuerwerk. Ein-/Ausgabeeinheit, Befehlsgruppen

Programmiersprachen-Grundlagen

- Syntax (und Semantik)
- Grammatik-Notationen
- Programmiersprachen Übersicht

Das erste M3-Programm

```
MODULE Willkommen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 16.08.98
   Letzte Aenderung: 20.08.98
*)

IMPORT SIO;
BEGIN
  SIO.PutText("Willkommen zum Studium in Aachen.");
END Willkommen.
```

Syntax und Semantik

Programmiersprachen - Definition

- **Programmiersprachen sind *künstliche Sprachen* (keine natürlichen Sprachen), deren Syntax und Semantik genau festgelegt ist.**
- **Syntax:**
  - Die ***Syntax*** einer Programmiersprache S ist die Definition aller in S ***zulässigen Aussagen***, die in einer Sprache formuliert werden können (Wörter, hier Programme).
- **Semantik:**
  - Die ***Semantik*** einer Sprache S ist die Definition der den zulässigen Aussagen zugeordneten ***Bedeutungen***.
  - Syntaktische ***falsche*** Aussagen haben ***keine*** Semantik
  - Aber auch syntaktisch korrekte Aussagen haben nicht immer eine Semantik (z.B. ein Programm, in dem durch 0 dividiert wird)
  - statische, dynamische Semantik
- **Pragmatik**
  - menschliche, ökonomische

Syntax und Semantik

Beispiel: Syntax, Semantik

- **Natürliche Zahlen**
  - (ohne die Null) dargestellt im Dezimalsystem in arabischen Ziffern bilden eine einfache künstliche Sprache
  - ***Syntax***:
    - jede Zahl ist eine Sequenz von Ziffern (0,1, .. , 9), wobei die erste Ziffer nicht 0 ist
  - ***Semantik***:
    - der Wert einer Zahl ist definiert als der Wert ihrer letzten Ziffer, vermehrt um den zehnfachen Wert der links davon stehenden Zahl, falls diese vorhanden ist (***rekursive*** Definition)
- **Beispiel**
  - syntaktisch ***korrekt*** ist: 367 Semantik: 7+10* (36)
7+10*(6+10*(3))
7+10*(6+10*3)
  - syntaktisch ***falsch*** ist: 007 keine Semantik

Syntax und Semantik

Alphabet

Alphabet

- Ein Alphabet (Zeichenvorrat) ist eine ***nichtleere endliche*** Menge von ***unterscheidbaren*** Zeichen ("Buchstaben", Symbolen)

  A = {a_1, a_2, a_3, …} mit einer ***Ordnungsrelation*** ≤ ($a_1 \le a_2 \le a_3$ ….)

- **Beispiel**:
  - das lateinische Alphabet (a, b, c,.. z)
  - der ASCII-Code (bestehend aus 128 Zeichen)
  - hier Latin-1-Zeichensatz ISO

Wort über einem Alphabet

- ***endliche Folge*** von Buchstaben, die auch ***leer*** sein kann (ε leere Wort)
- A* bezeichnet die ***Menge aller Wörter*** über dem Alphabet A (inkl. dem leeren Wort)

Syntax und Semantik

Formale Sprache

Definition:

- Sei A ein Alphabet. Eine (formale) Sprache (über A) ist ***eine beliebige Teilmenge von A****.

Beispiele:

- A_1 = { 0,1}, A_1* = {ε , 0, 1, 01, 10, 10, 000, 100 , … }
- L = {0, 1, 10, 11, 100, 101, .. } ⊂ A_1*, die Menge der Binärdarstellungen natürlicher Zahlen (mit Null, ohne führende Nullen)

- A_2 = {(,), +, -, *, /, a}, A_2* = {ε, (), (+-a), (a*a), … }
- die Sprache der korrekt geklammerten Ausdrücke EXPR ⊂ A_2* : EXPR = { (((a))), (a+ a), (a -a)*a+ a /(a+ a) -a, .. }

Da solche Sprachen i.d.R. unendlich sind, benötigt man eine endliche Beschreibungsvorschrift

- ***Grammatik***, die die Sprache erzeugt
- ***Automat***, der die Sprache erkennt

Grammatik-Notationen

Grammatik - informell - 1

- **Definiert *Regeln*, die festlegen, welche Wörter über einem *Alphabet* zur Sprache gehören und welche nicht.**
- **Beispiel: Grammatik für "Hund-Katze-Sätze"**

1	<Satz>	->	<Subjekt> <Prädikat> <Objekt>
2	<Subjekt>	->	<Artikel> <Attribut> <Substantiv>
3	<Artikel>	->	ε
4	<Artikel>	->	der
5	<Artikel>	->	die
6	<Artikel>	->	das
7	<Attribut>	->	ε
8	<Attribut>	->	<Adjektiv>
9	<Attribut>	->	<Adjektiv> <Attribut>
10	<Adjektiv>	->	kleine
11	<Adjektiv>	->	bissige
12	<Adjektiv>	->	große
13	<Substantiv>	->	Hund
14	<Substantiv>	->	Katze
15	<Prädikat>	->	jagt
16	<Objekt>	->	<Artikel> <Attribut> <Substantiv>

Grammatik-Notationen

Grammatik - informal - 2

- **Grammatik für "Hund-Katze-Sätze"**
  - durch diese Grammatik können z.B. die folgenden Sätze gebildet (abgeleitet) werden
    - "der kleine bissige Hund jagt die große Katze"
    - "die kleine Katze jagt der bissige Hund"
    - "das große Katze jagt der kleine große bissige kleine …. Katze"
  - folgende "Sätze" werden nicht durch diese Grammatik gebildet
    - "die Katze der Hund"
    - "Katze und Hund"
    - "der Hund jagt die Katze die jagt Hund"

Grammatik-Notationen

Grammatik - Definition - 1

- **Definition:**
  - Eine Grammatik G für eine Sprache L ist definiert durch
  - ein ***Viertupel (N, T, P, S)***
- **N: Menge der *Nichtterminalsymbole***
  - sind Zeichen oder Zeichenfolgen für syntaktische ***Abstraktionen***
  - Beispiel: <Subjekt>, <Objekt>
  - kommen nicht in den Wörtern der Sprache vor
  - werden durch Anwendung der ***Produktionsregeln*** solange ersetzt, bis nur noch Terminalsymbole übrig sind
- **T: Menge der *Terminalsymbole***
  - sind Zeichen oder Zeichenfolgen des Alphabets, aus denen die Wörter der Sprache bestehen
  - Beispiel: kleine , Katze

Grammatik-Notationen

Grammatik - Definition - 2

- **P: Menge von *Produktionsregeln***
  - definieren, wie aus bekannten Konstrukten ***neue Konstrukte*** geschaffen werden
  - Die Anwendung einer Regel bedeutet, daß in der bereits erzeugten Satzform der Teil, der der linken Seite der Regel entspricht, durch die rechte Seite ***ersetzt*** wird
  - **Beispiel**:
    - der kleine bissige Hund <Prädikat> <Objekt>
    - der kleine bissige Hund *jagt* <Objekt>

- **S: das *Startsymbol***
  - ist ein spezielles Nichtterminalsymbol, aus dem ***alle Wörter*** der Sprache mit Hilfe der Grammatik erzeugt werden
  - **Beispiel**: <Satz>
- **Sprache:** Jedes durch Anwendung der Regeln erzeugbare Wort, ***das nur aus Terminalsymbolen besteht***, gehört zu der von der Grammatik erzeugten Sprache L(G)

Grammatik-Notationen

Grammatik - Definition - 3

- **Es gilt:**
  - $N \cap T = \varnothing$
  - $V = N \cup T$ (Gesamtalphabet, Vokabular)
  - sei $p \in P$: $(\alpha \rightarrow \beta)$, $\alpha \in V^* N V^*$, $\beta \in V^*$

Terminale und Nichtterminale sind verschieden

Auf der linken Seite einer Produktion steht wenigstens ein Nichtterminal

- **Ableitung**
  - Ableitungsprozeß ist eine Relation " " auf V^*
  - Für $u,v, \beta \in V^*$ und $\alpha \in V^* N V^*$ gilt
    - $u\alpha v \quad u\beta v$ genau dann, wenn $(\alpha \rightarrow \beta) \in P$
- **Die von einer Grammatik *erzeugte Sprache* ist definiert als:**
  - $L(G) = \{ w \mid w \in T^*, S \overset{*}{} w\}$

w ist herleitbar aus dem Startsymbol

zwei Grammatiken heißen ***äquivalent***, wenn sie dieselbe Sprache erzeugen

Grammatik-Notationen

Typen von Produktionen

- **Je nach Gestalt der in P *zugelassenen Produktionen* definiert Chomsky 4 Typen von Grammatiken**

Produktion	Typ	Eigenschaften	CH-Typ
$(\alpha \rightarrow \beta)$	allgemein	$\alpha, \beta \in V^*$ beliebig	Typ-0
$(\alpha \rightarrow \varepsilon)$	ε -Produktion	$\alpha \in V^*$, $r = \varepsilon$	
$(\alpha \rightarrow \beta)$	beschränkt	$\alpha, \beta \in V^*$, $1 <= \lvert\alpha\rvert <= \lvert\beta\rvert$	Typ-1
$(uAv \rightarrow u\beta v)$	kontextsensitiv	$A \in N$, $u, v, \beta \in V^*$, $\beta \neq \varepsilon$	Typ-1
$(A \rightarrow \beta)$	kontextfrei	$A \in N$, $\beta \in V^*$	Typ-2
$(A \rightarrow Bx)$	linkslinear	$A, B \in N$, $x \in T$	Typ-3
$(A \rightarrow xB)$	rechtslinear		Typ-3
$(A \rightarrow x)$	terminierend	$A \in N$, $x \in T$	

Grammatik-Notationen

Chomsky-Grammatiken

- **Die Chomsky-Grammatiken bilden eine Hierarchie**
  - d.h. die Menge der von Typ-n-Grammatiken erzeugten Sprachen umfaßt die Menge der Sprachen, die von Typ n+1 Grammatiken erzeugt werden
- **Typ-0-Grammatik**
  - Gestalt der Produktionen ist nicht eingeschränkt
  - Alle Sprachen, die überhaupt mit endlichen Regelsystemen erzeugt werden können
- **Typ-1 oder kontextsensitive Grammatik**
  - Produktionen sind beschränkt oder kontextsensitiv
- **Typ-2 oder *kontextfreie* Grammatik**
  - Produktionen sind kontextfrei (d.h. die linke Seite einer Produktion ist immer ein Nichtterminal)
- **Typ-3 oder reguläre Grammatik**
  - Produktionen sind terminierend, links- , rechtslinear

Grammatik-Notationen

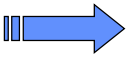
Kontextfreie Grammatik

- **Wichtigste Klasse zur formalen Beschreibung der Syntax von Programmiersprachen.**
- **Es ist möglich, Automaten zu bauen, die Wörter einer kontextfreien Sprache erkennen**
  - ***Wortproblem***
    - Kann für eine kf. Grammatik G und ein Wort w $\in$ T* festgestellt werden, ob w von G erzeugt wird oder nicht.
  - ***Analyseproblem***
    - Gibt es einen Algorithmus, der zu einer kf. Grammatik G und einem Wort w $\in$ T* die syntaktische Struktur von w bestimmt, oder aber feststellt, daß w nicht in L(G) liegt
    - ***Parser*** (Zerteilungsalgorithmus): Syntaxanalyse
- **Notationen zur Darstellung kontextfreier Grammatiken**
  - ***Syntaxdiagramme***
  - ***Extended Backus-Naur-Form* (EBNF)**

Grammatik-Notationen

Grammatik - Beispiel

- **kf. Grammatik, die korrekt geklammerte arithmetische Ausdrücke mit Operatoren *, + erzeugt**
- G1 = ({E,T,F}, {(,), a, +, *}, P, E) mit
  - P = { ❶ E -> T,
    ❷ E -> E + T,
    ❸ T -> F,
    ❹ T -> T * F,
    ❺ F -> a,
    ❻ F -> (E) }
- **a * (a + a) ∈ L(G1)**
  - denn a*(a+a) läßt sich aus E folgendermaßen ableiten
  - dabei wurde immer das am weitesten links stehende Nichtterminal ersetzt (***Linksableitung***)



		Angewandte Regel
E	=> T	❶
	=> T * F	❹
	=> F * F	❸
	=> a * F	❺
	=> a * (E)	❻
	=> a * (E + T)	❷
	=> a * (T + T)	❶
	=> a * (F + T)	❸
	=> a * (a + T)	❺
	=> a * (a + F)	❸
	=> a * (a + a)	❺

Grammatik-Notationen

Diskussion Beispiel - 1

- **Betrachtet man die Erzeugung von arithmetischen Ausdrücken genauer:**
  - Erzeugungsprozeß ist rückwärts betrachtet ein Prozeß der ***sukzessiven Zusammenfassung*** von Teilausdrücken: Reduktion
  - schließlich wird der gesamte Ausdruck auf das Startsymbol ***zurückgeführt***
- **Dabei wird z.B. folgendes beachtet**
  - Vorrang der Klammerstruktur
  - Vorrang der Multiplikation von der Addition
  - Zusammenfassen von links nach rechts bei gleichrangigen Operatoren
- **Beispiel zeigt**
  - daß die Syntax bereits auf grundlegende Eigenschaften der Semantik ***abgestimmt*** sein kann!

Grammatik-Notationen

Diskussion Beispiel - 2

- **Folgende kf. Grammatik erzeugt dieselbe Sprache wie G1**

  G2 = ({E}, {(,), a, +, *}, P, E) mit

  P = { ❶ E -> E + E,
  ❷ E -> E * E,
  ❸ E -> (E),
  ❹ E -> a }

- **Bemerkung**
  - bei G2 fehlt die bei G1 festgestellte Abstimmung der Syntax, d.h. des Erzeugungsprozesses auf die Regeln der Auswertung arithmetischer Ausdrücke
  - G1 kann als Modell für die Definition von Ausdrücken in höheren Programmiersprachen angesehen werden (keine Auswertungs-reihenfolge, Prioritätsfestlegung).

Grammatik-Notationen

Grammatik - Beispiel

- **Sei G = (N, T, P, S) mit**
  - N = {A, B}
  - T = {a, b, c, d}
  - P = { (A -> aBbc),
    (B -> aBb),
    (aBb -> d) }
  - S = A
  - G ist keine kontextfreie Grammatik, da die dritte Produktionsregel auf der linken Seite mehr als nur das Nichtterminalsymbol enthält.

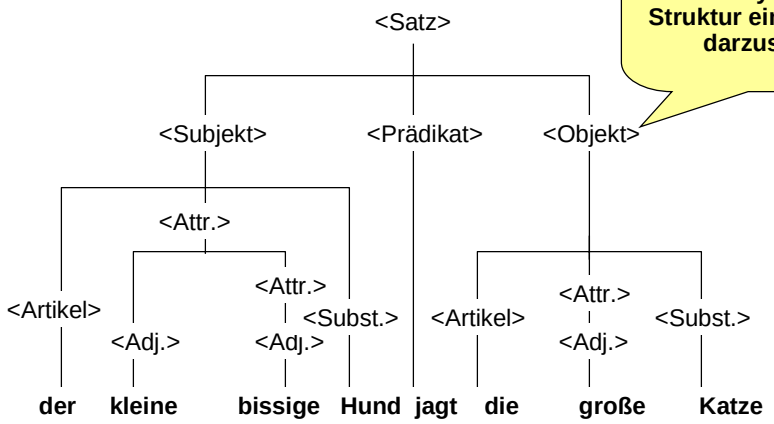
- **Ersetzt man in G die Produktionen P durch P', dann ist G' kontextfrei. Es gilt L(G) = L(G')**
  - P' = { (A -> Bc),
    (B -> aBb),
    (B -> d) }

Grammatik-Notationen

Ableitungsbaum -Strukturbaum

- **Vaterknoten = linke Seite einer Regel**
- **Söhne = rechte Seite einer Regel**

Mittel, um die syntaktische Struktur eines Wortes darzustellen



Grammatik-Notationen

Syntaxdarstellung - EBNF - 1

- **EBNF**
  - Extended Backus-Naur-Form
  - ***Meta-Sprache*** zur Beschreibung der Syntax formaler Sprachen
  - BNF erstmals benutzt zur Definition der Sprache ***Algol-60***
  - ***Metasymbole*** von EBNF sind
    - = „definiert als“
    - (...|...) genau eine Alternative aus der Klammer muß stehen
    - [...] Inhalt der Klammer kann stehen oder nicht
    - { ... } Inhalt der Klammer kann n-fach stehen, n >= 0
    - . Ende der Produktion
    - Terminalsymbole werden in " " eingeschlossen

Grammatik-Notationen

Syntaxdarstellung - EBNF - 2

- **Unsere einfache Grammatik für "Hund-Katze-Sätze" sieht in EBNF folgendermaßen aus:**

Satz	=	Subjekt Prädikat Objekt.
Subjekt	=	Artikel Attribut Substantiv.
Artikel	=	[("**der**" | "**die**" | "**das**")].
Attribut	=	[(Adjektiv | Adjektiv Attribut)].
Adjektiv	=	("**kleine**" | "**bissige**" | "**große**").
Substantiv	=	("**Hund**" | "**Katze**").
Prädikat	=	"**jagt**".
Objekt	=	Artikel Attribut Substantiv.

Grammatik-Notationen

Syntaxdarstellung - EBNF - 2

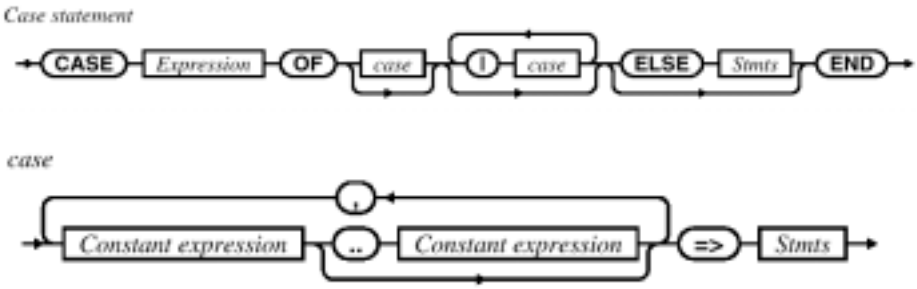
CaseStatement = „**CASE**“ Expression „**OF**“
[Case] { „**|**“ Case }
[„**ELSE**“ Stmts] „**END**“ .

```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 |'/'  => resultat := a / b;
 ELSE (* Fehler *)
END;
```

Grammatik-Notationen

Syntaxdiagramme - Beispiel

- **Syntaxdiagramme**
  - beschreiben Produktionen ***grafisch***
  - Nichtterminalsymbole sind Rechtecke
  - Terminalsymbole sind Langrunde



Programmier-sprachen: Allgemeines

Syntaxebenen

- **lexikalische Syntax**
- **kontextfreie Syntax**

} **Vgl. Modula-Syntax**

- **kontextsensitive Syntax (umgangssprachlich)**

```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 |'/'  => resultat := a / b;
 ELSE (* Fehler *)
END;
```

Programmier-sprachen: Allgemeines

Lexikalische Einheiten

- Bezeichner
- Begrenzer
- Wortsymbole
- Literale
- Kommentare

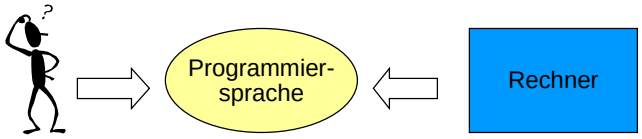
Eventl. Trennzeichen zwischen lexikalischen Einheiten

```
CASE  operator OF
  '+'  => resultat := a + b;
 |'-'  => resultat := a - b;
 |'*'  => resultat := a * b;
 |'/'  => resultat := a / b;
 ELSE (* Fehler *)
END;
```


Programmier-sprachen: Allgemeines

Programmiersprachen

- **Die Programmiersprache bildet die *Schnittstelle* zwischen Mensch und Rechner**



Beide haben unterschiedliche Anforderungen

- **Mensch**
  - Erlernbarkeit
  - Lesbarkeit
  - Ausdrucksstärke

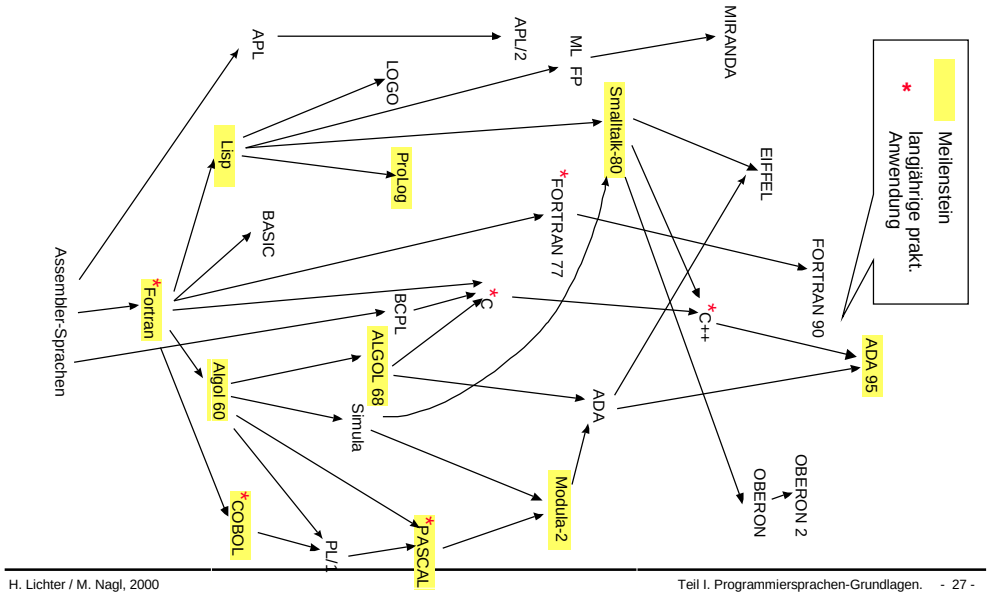
Problem-orientierte Sprachen

- **Rechner**
  - einfaches Übersetzen in Maschinensprache
  - effizienter Code soll generiert werden können

Maschinen-sprachen

Programmier-sprachen: Allgemeines

Entwicklung



Programmier-sprachen: Allgemeines

Werkstoff

- **Programmiersprachen sind der *Werkstoff* des Informatikers**
- **Das, was wir erzeugen (Programme)**
  - ist ***immateriell***
  - wir bauen es aus dem Werkstoff "Programmiersprache"
- **Ein Informatiker**
  - sollte die ***Qualität*** und ***Eignung*** verschiedener Werkstoffe (Programmiersprachen) kennen
  - Welche Sprache ist wofür geeignet?
- **Analogie!**
  - Ein Bauingenieur verwendet andere Werkstoffe, wenn ein
    - Hochhaus (Stahl, Beton etc.)
    - oder ein Einfamilienhaus (Ziegelsteine, Holz, Beton, etc.)
  - gebaut werden soll

Was haben wir gelernt

- Wie wissen was Programmiersprachen sind
- Wir kennen den Begriff "Formale Sprache"
- Wir haben gesehen, daß eine Grammatik eine Sprache erzeugt
- Wir kennen EBNF und Syntaxdiagramme
- Wir sehen Programmiersprachen als Werkstoff des Informatikers

Glossar

- Programmiersprache
- Syntax, Syntaxebenen
- Semantik
- Alphabet
- Formale Sprache
- Grammatik
  - kontextfreie Grammatik
- Syntaxdarstellung: EBNF, Syntaxdiagramm
- Ableitungsbaum, Syntaxbaum
- lexikalische Einheiten

Aufbau vom M3 Programmen

- **Was ist Modula-3**
- **Modulkonzept**
- **Aufbau von Modula-3-Programmen**
- **Vom Programmtext zum ausführbaren Programm**

Warum Modula-3

Die Programmiersprache Modula-3

- **Modula-3**
  - erlaubt Programmierkonzepte ***elegant*** zu formulieren
  - zeigt die zentralen Konzepte der ***imperativen*** und ***objektorientierten*** Programmierung
  - ist ***leicht*** erlernbar
  - ist im Sinne der ***software-technischen Qualität*** von Programmen entwickelt worden
  - fördert einen "***guten***" Programmierstil
- **Wichtig ist (im Sinne der VL)**
  - nicht die Programmiersprache Modula-3 selbst
  - sondern die ***Programmierkonzepte***
  - Die erlernten Konzepte werden Sie später in anderen Sprachen in anderer Form wiederfinden.

Warum Modula-3

Modula-3

As Sam Harbison writes in his book Modula-3,

Modula-3 is a member of the ***Pascal*** family of languages. Designed in the late 1980s at Digital Equipment Corporation and Olivetti, Modula-3 corrects many of the ***deficiencies*** of Pascal and Modula-2 for ***practical software engineering***. In particular, Modula-3 keeps the ***simplicity*** of type safety of the earlier languages, while providing ***new facilities*** for exception handling, concurrency, object-oriented programming, and automatic garbage collection. Modula-3 is both a practical implementation language for large software projects and an ***excellent teaching language***.

- **Modula = Modular Language**
  - Modul ist ein wichtiges Sprach- und Programmierkonzept
- **Modula-3 ist**
  - eine ***imperative*** (prozedurale) Programmiersprache
  - eine ***strukturierte*** Programmiersprache
  - eine ***objektorientierte*** Programmiersprache

Das erste M3-Programm

```
MODULE Willkommen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Autor           : Horst Lichter, RWTH Aachen
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 16.08.98
   Letzte Aenderung: 20.08.98
*)

IMPORT SIO;
BEGIN
  SIO.PutText("Willkommen zum Studium in Aachen.");
END Willkommen.
```

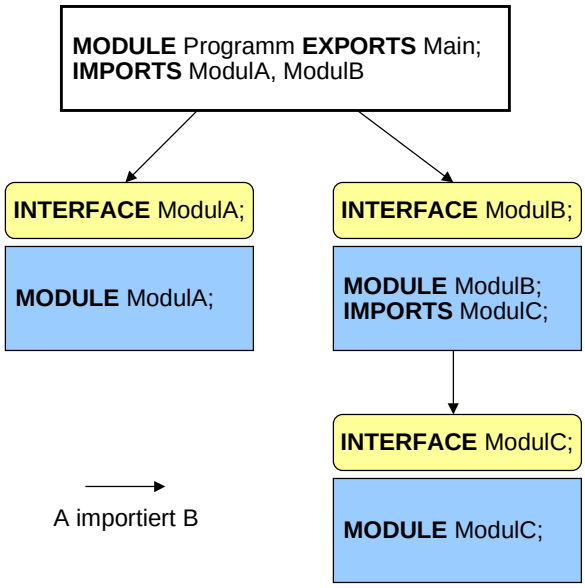
Aufbau von M3

Modula-3 Programme

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul.***
- **Was ist ein Modul?**
  - Ein Modul ist ein Programmteil, das ***sinnvoll*** zusammengehörende Elemente enthält.
  - Ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Schnittstelle*** (interface)
      - definiert, was ein Modul exportiert
    - ***Implementierung***
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Hauptmodul**
  - exportiert die vordefinierte ***leere*** Schnittstelle `Main`

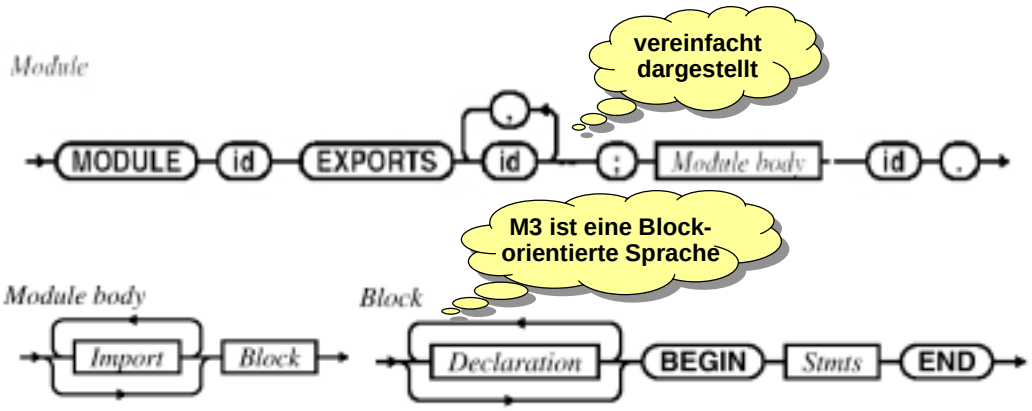
Modul-Hierarchie

- Ein Modul kann Elemente anderer Module ***benutzen.***
  - Ein Modul ***importiert*** dazu andere Module.
  - Von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar.

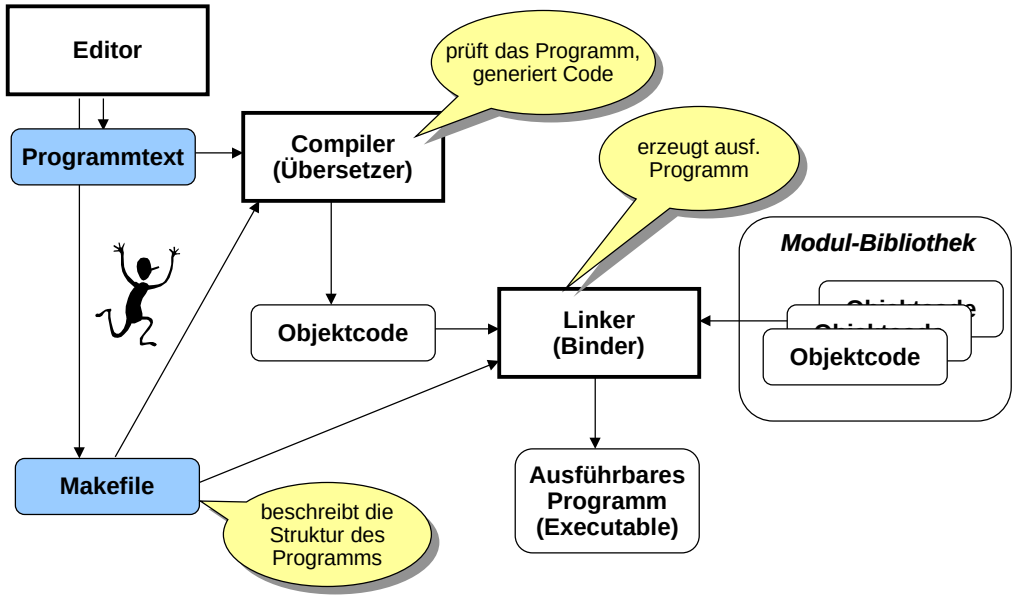


Aufbau eines Moduls

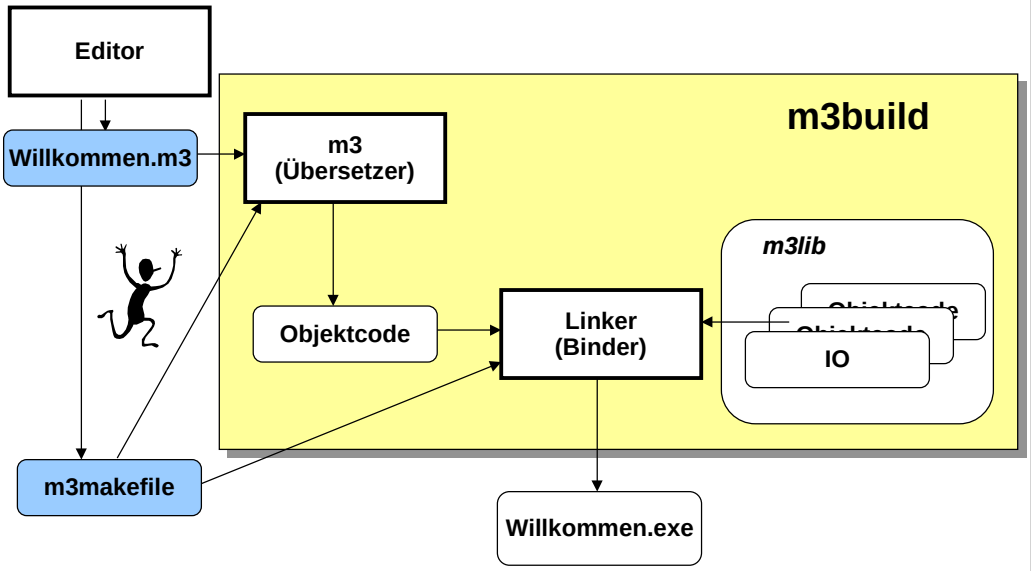
- **Ein Modul**
  - ***exportiert*** Elemente und ***importiert*** andere Module
  - enthält einen ***Block***
- **Ein Block**
  - enthält ***Deklarationen*** und ***Anweisungen***



Vom Programmtext zum ausf. Programm



Edit-Compile-Link-Run bei Modula-3



Vorteile modularer Programme

- **Vorteile modularer Programme**
  - Module können von ***unterschiedlichen*** Personen entwickelt und gepflegt werden.
    - Software-Entwicklung ist Team-Arbeit
  - Module können einzeln ***getestet*** werden.
    - Test großer Programme ist extrem aufwendig
  - Module können geordnet zum Gesamtsystem ***integriert*** werden
  - Eine Implementierung eines Moduls kann leichter durch eine neue Implementierung ***ersetzt*** werden.
    - z.B. durch eine effizientere Implementierung
  - Module können in verschiedenen Programmen ***wiederverwendet*** werden (Modul-Bibliothek).
    - Dies senkt die Kosten für die Entwicklung

Glossar

- **Modul**
  - Schnittstelle
  - Implementierung
  - Hauptmodul
- **Block**
  - Blockorientierte Programmiersprachen
- **Programmentwicklungswerkzeuge**
  - Übersetzer
  - Binder
- **Datei-Arten**
  - Quelltext-Datei
  - Objektcode
  - ausführbare Programmdatei

Einführung in Modula-3
Funktionale Programme

- **Funktionale Programmierung**
- **Funktionale Programme in Modula-3**
- **Funktionen, Parameter**
- **Einfache Datentypen**
- **Rekursion**

Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
  - Es existiert keine Möglichkeit, Zwischenergebnisse zu speichern
- **Was benötigt man dazu**
  - Daten / ***Datentypen***
  - ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Anmerkung:**
  - Es gibt rein funktionale Programmiersprachen MIRANDA, LISP
  - Viele sog. Hochsprachen erlauben eine gewisse Art der funktionalen Programmierung (Modula-3).

Funktionen in Modula-3

- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereich)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift***
  - ist frei von ***Seiteneffekten***

Signatur

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;
```

Berechnungsvorschrift **Name** **formale Eingabeparameter** **Ergebnistyp**

Aufruf einer Funktion

```
MODULE QuadratM EXPORTS Main;
(* Dieses Programm berechnet das Quadrat einer Zahl
   Autor           : Horst Lichter
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 20.08.98    Letzte Aenderung: 20.08.98
*)

IMPORT SIO;

PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (SIO.GetReal()));
END QuadratM.
```

Deklaration und ***Definition*** der Funktion

Aufruf der Funktion mit einem ***aktuellen*** Parameter;

Das ***Ergebnis*** der Funktion wird an die Prozedur PutReal ***übergeben***

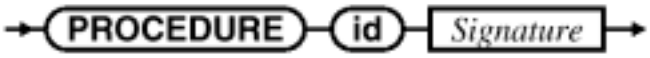
Formale & aktuelle Parameter

- **Parameter**
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - haben eine ***Typ***
  - dadurch werden Funktionen ***flexible einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - Beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden.
  - Diese werden dann im Rumpf ***verwendet***.

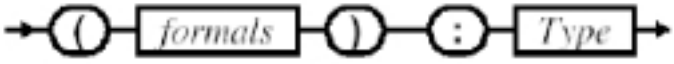
Syntax von M3-Funktionen - 1

Declaration

Procedure heading

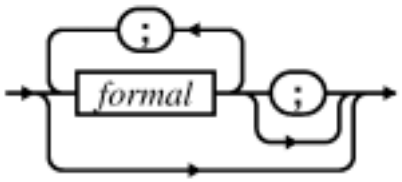


Signature

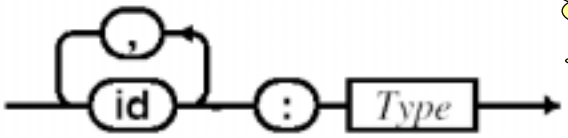


Syntax von M3-Funktionen - 2

formals



formal



Vereinfachte Darstellung !

Formale & aktuelle Parameter

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

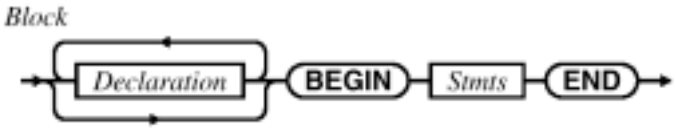
BEGIN
  SIO.PutReal (Quadrat (2.5));
END QuadratM.
```

Binden der aktuellen Werte (Parameter) an die formalen Parameter (Stellvertreter)

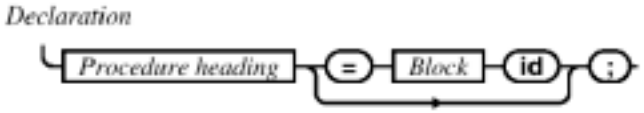
Vorschau: ...

Deklarationen (vorläufig)

- **Wir wissen bereits:**
  - ***Blöcke*** können ***Deklarationen*** enthalten
  - und bestehen aus ***Anweisungen*** (Statements)



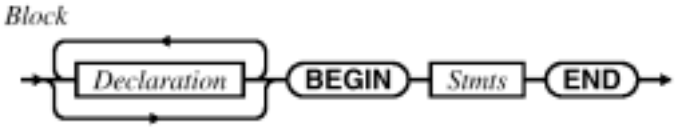
- **Deklarationen:**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***



Vorschau: ...

Anweisungen (vorläufig)

- **Anweisungen:**
  - sind in Blöcken enthalten und werden durch ";" getrennt
  - atomare Bausteine für Modula-3 Programme



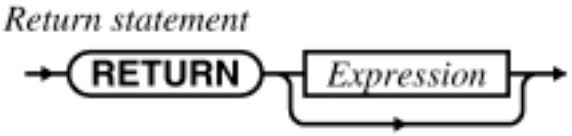
  - Die Folge der Anweisungen eines Blocks wird bei Ausführung in der ***Reihenfolge der Aufschreibung*** abgearbeitet

- **Beispiele für Anweisungen:**
  - RETURN-Anweisung
  - CALL-Anweisung (Funktionsaufruf)

Ausdrücke

Vorschau: ...

- **Ausdrücke**
  - bezeichnen Elemente, die ***ausgewertet*** werden
  - liefern einen Wert (Ergebnis)
- **Beispiele**
  - arithmetische Ausdrücke
    - x * x
  - logische Ausdrücke
    - x AND y
- **Viele Anweisungen können Ausdrücke enthalten**



Datentypen (vorläufig)

Vorschau: ...

- **Typbegriff**
  - Im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung.
- **Definition**
  - Unter einem ***Datentyp*** versteht man die ***Zusammenfassung*** von ***Wertebereich*** und ***Operationen*** zu einer ***Einheit.***
  - Ein Typ hat eine Struktur und Literale (Aggregate).
- **Man unterscheidet:**
  - ***einfache*** (skalare) Datentypen
  - ***zusammengesetzte*** Datentypen
    Konstruktionsmittel: Datentypkonstruktoren
  - ***vordefinierte*** Datentypen
  - ***selbstdefinierte*** Datentypen

Vordefinierte Datentypen

Vordefinierte Datentypen in Modula-3

- **Einfache, skalare Datentypen**
  - **Ganze Zahlen (diskret)**
    - darunter fallen die Typen **`INTEGER`** und **`CARDINAL`**
  - **Zeichen (diskret)**
    - Werte eines bestimmten Zeichenvorrates; z.B. definiert der ASCII-Zeichenvorrat 128 Zeichen
  - **Wahrheitswerte (diskret)**
    - Werte sind {wahr, falsch}, Literale **`TRUE`**, **`FALSE`**
  - **Gleitkommazahlen**
    - reelle Zahlen, **`REAL`**, **`LONGREAL`**, **`EXTENDED`**
- **Zusammengesetzte Datentypen**
  - **Texte**
    - sind eine Folge von Zeichen

Vordefinierte Datentypen

Ganzahlige Datentypen

- **Typen: `INTEGER` und `CARDINAL`**
  - **Integer-Zahlen** sind ***ganzzahlige*** Werte innerhalb der Unter- und Obergrenze des jeweiligen Rechners
  - **Cardinal-Zahlen** sind sind ***nicht-negative*** ganzzahlige Werte, d.h. zwischen 0 und der Obergrenze des jeweiligen Rechners
- **Wertebereich**
  - auf einen 32-Bit-Rechner:
    - **`INTEGER`** [-2147483648 .. 2147483647] oder [-2^{31} .. $2^{31}-1$]
    - 1 Bit für das Vorzeichen, 31Bit für die Zahlendarstellung
  - Zahlen sind geordnet (***Ordinaltyp***)
- **Operationen**
  - arithmetische Operationen (**`+`**, **`-`**, **`*`**, **`DIV`**, **`MOD`**)
  - Vergleichsoperationen (**`=`**, **`#`**, **`<`**, **`>`**, **`<=`**, **`>=`**)
  - vordefinierte Funktionen (**`FIRST(type)`**, **`LAST(type)`**, **`INC(z)`**, **`DEC(z)`**, **`ABS(z)`**)

Vordefinierte Datentypen

Beispiel

```
MODULE Zahlen EXPORTS Main;
(* Dieses Programm zeigt dem Umgang mit ganzen Zahlen *)

IMPORT SIO;

PROCEDURE Modulo (x,y: INTEGER): INTEGER =
(* MOD ist def. : x MOD y = x - y*(x DIV y) *)
BEGIN
  RETURN ( x - y*(x DIV y) );
END Modulo;

BEGIN
  (* Ausgabe der Unter- und Obergrenze von INTEGER *)
  SIO.PutInt (FIRST(INTEGER)); SIO.Nl();
  SIO.PutInt (LAST(INTEGER));  SIO.Nl();

  SIO.PutInt (20 DIV 6); SIO.Nl();     (* = 3 *)
  SIO.PutInt (20 MOD 6); SIO.Nl();     (* = 2 *)
  SIO.PutInt (Modulo(20,6));           (* = 2 *)
END Zahlen.
```


Vordefinierte Datentypen

Gleitpunktdatentypen

- **Typen: REAL, LONGREAL, EXTENDED**
  - repräsentieren die darstellbaren reellen Zahlen
- **Darstellung**
  - Wertebereich ist ***beschränkt*** (im Unterschied zur Mathematik)
  - Genauigkeit der Darstellung ist ***beschränkt***
    - Bspl: wir haben 4 Stellen zur Verfügung
      größte Zahl: 9999
      kleinste Zahl: 0.001
    - Probleme: die Zahlen 0.0005 und 0.000089 sind nicht zu unterscheiden (es wird gerundet)
  - Rechnen mit Gleitkommazahlen ist immer **fehlerbehaftet**!
    - "Numerik" liefert Techniken und Algorithmen, um Fehler zu beherrschen

Vordefinierte Datentypen

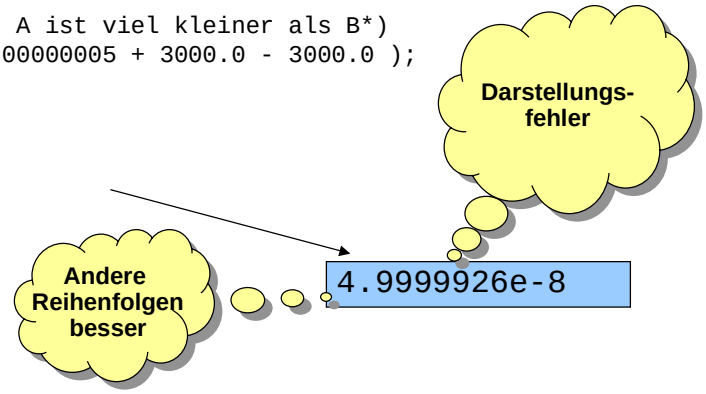
Beispiel für Gleitkommazahlen

```
MODULE Gleitkomma EXPORTS Main;
(* Dieses Programm zeigt Rundungsprobleme bei Gleitkommazahlen *)

IMPORT SIO;


BEGIN
  (* A + B - B mit A ist viel kleiner als B*)
  SIO.PutReal ( 0.00000005 + 3000.0 - 3000.0 );

END Gleitkomma.
```



Vordefinierte Datentypen

Zeichentyp CHAR

- **Typ CHAR**
  - **CHAR** (character) bezeichnet eine ***endliche***, ***geordnete Menge*** von Zeichen
  - **CHAR** ist ein ***Ordinaltyp***
- **Wertebereich**
  - Latin-1, viele Rechner benutzen den ASCII-Zeichensatz
  - Zeichenliterale: 'A' 'z' '1'
  - Spezialzeichen: `\n` Zeilenvorschub `\t` Tabulator `\\` Backslash
    `\'` Apostroph `\f` Seitenumbruch
    `\r` Wagenrücklauf `\"` Anführungszeichen
- **Operationen**
  - Vergleichsoperationen (**=**, #, **<**, **>**, **<=**, **>=**)
  - Vordefinierte Funktionen **`FIRST(CHAR), LAST(CHAR), INC(z), DEC(z), ORD(z), VAL(i)`**

Vordefinierte Datentypen

Beispiel - Zeichen

```
MODULE KleinGross EXPORTS Main;
(* Dieses Programm wandelt einen Klein- in einen Grossbuchstaben um *)

IMPORT SIO;

PROCEDURE Offset (): INTEGER =
BEGIN
  RETURN (ORD('A') - ORD('a'));
END Offset;


BEGIN
  (* Kleinbuchstaben einlesen und umwandeln *)
  SIO.PutChar ( VAL(ORD(SIO.GetChar()) + Offset(), CHAR) );

END KleinGross.
```


Vordefinierte Datentypen

Texte (Zeichenketten)

- **Typ TEXT**
  - repräsentiert eine ***beliebig lange Folge*** von Zeichen (kann auch leer sein)
  - in vielen Sprachen nicht explizit vorhanden
- **Wertebereich**
  - Textliterale werden in " " notiert
  - z.B. "Das ist ein Text mit Zeilenvorschub\n"
"\"Dieser Text beginnt und endet mit einem Hochkomma\""
- **Operationen**
  - Konkatenation : &
    - Bspl: "Heute " & "ist " & "Freitag." ⇨ "Heute ist Freitag."
  - Schnittstelle des Moduls "Text"
    - Equal, Length, Empty, FindChar ….

Vordefinierte Datentypen

Wahrheitswerte

- **Typ BOOLEAN**
  - repräsentiert die beiden vordefinierten Wahrheitswerte
- **Wertebereich**
  - wahr, Literal **TRUE**
  - falsch, Literal **FALSE**
- **Operationen**
  - Komplement (**NOT**), Oder (**OR**), Und (**AND**)

p	q	NOT q	p OR q	p AND q
1	1	0	1	1
1	0	1	1	0
0	1	0	1	0
0	0	1	0	0

1 wahr
0 falsch

Vordefinierte Datentypen

Beispiel für Wahrheitswerte

```
MODULE BooleanM EXPORTS Main;
(* Dieses Programm berechnet die Wahrheitstabelle NOT, OR, AND *)
IMPORT SIO, Fmt;

PROCEDURE NotOrAnd ( a, b : BOOLEAN) : TEXT =
BEGIN
  RETURN (Fmt.Bool(a) & " " & Fmt.Bool(b) & " : " &
          Fmt.Bool( NOT(b))       & " " &
          Fmt.Bool( a OR b)       & " " &
          Fmt.Bool(a AND b) );
END NotOrAnd;

BEGIN
  (* Ausgabe der Wertetabelle *)
  SIO.PutLine(NotOrAnd(FALSE, FALSE));
  SIO.PutText(NotOrAnd(TRUE, FALSE));
  SIO.PutLine(NotOrAnd(FALSE, TRUE));
  SIO.PutLine(NotOrAnd(TRUE, TRUE));

END BooleanM.
```

produzierte Ausgabe

```
FALSE FALSE : TRUE  FALSE FALSE
TRUE  FALSE : TRUE  TRUE  FALSE
FALSE TRUE  : FALSE TRUE  FALSE
TRUE  TRUE  : FALSE TRUE  TRUE
```

Vernetzung von Funktionen

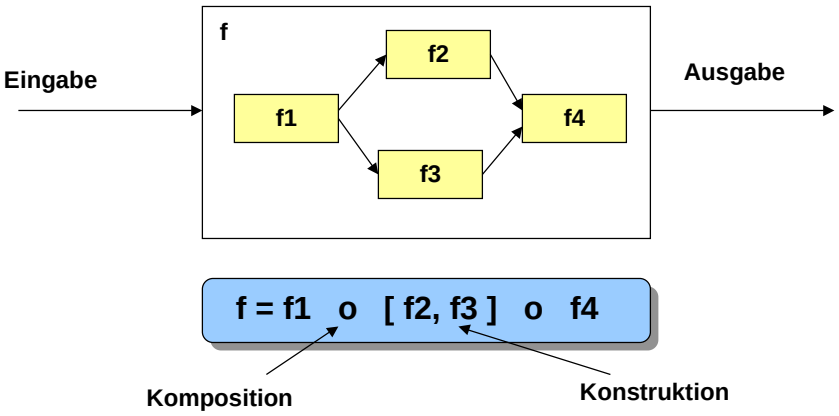
Funktionskomposition

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, ...] : x = (f : x, g : x, h : x, ...)
  - **Bedingung**
    - if t then f else g : x =

$$\begin{cases} f : x & \text{, falls } t : x = \text{TRUE} \\ g : x & \text{, falls } t : x = \text{FALSE} \\ ? & \text{, sonst} \end{cases}$$

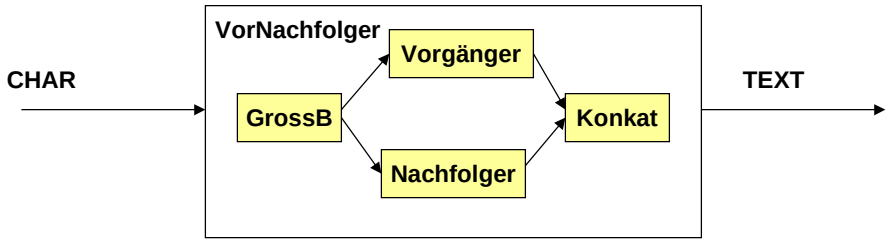
Vernetzung von Funktionen

schematisches Beispiel



Vernetzung von Funktionen

Beispielnetz



VorNachfolger = GrossB o [Vorgänger, Nachfolger] o Konkat

Beispiel:

Eingabe: 'h'

Ausgabe: "GI"

Vernetzung von Funktionen

Beispiel: Detaillierung

```
MODULE VorNachfolger EXPORTS Main;
IMPORT SIO;

PROCEDURE GrossB(c : CHAR) : CHAR =
...
PROCEDURE Vorgaenger (c : CHAR) : CHAR =
...
PROCEDURE Nachfolger (c : CHAR) : CHAR =
...
PROCEDURE AddOrd (c, d : CHAR) : CARDINAL =
. . .
PROCEDURE Konkat (c, d : CHAR) : TEXT =
...

BEGIN
  (* VorNachfolger = GrossB o   [ Vorgänger, Nachfolger ]   o   Konkat *)
  SIO.PutText ( Konkat ( Vorgaenger( GrossB( SIO.GetChar() )) ,
                         Nachfolger( GrossB( SIO.GetChar() ))
                       )
              );

END VorNachfolger.
```

Da **keine Variablen** (Zwischenspeicher) verwendet werden, muß der Wert **zweimal** eingelesen werden !

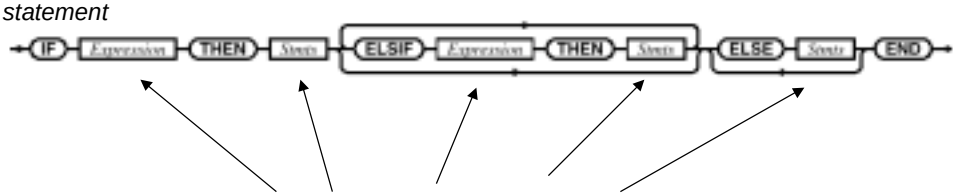
Funktion GrossB muß **zweimal** ausgeführt werden, da Modula-3 die **Konstruktion** von Funktionen nicht unterstützt.

Bedingungen in funkt. Programmen

Funktionale Programme

- **Idee:**
  - In Abhängigkeit von ***Bedingungen*** soll eine ***alternative*** Programmkomponente ausgeführt werden
  - Bedingungen müssen einen Wert vom Typ ***BOOLEAN*** liefern
  - Modula-3 bietet dazu u.a. die ***IF-Anweisung*** an

If statement



Bei funktionalen Programmen stehen hier ***ausschließlich*** Funktionsaufrufe!

Bedingungen in funkt. Programmen

Bedingung - IF-THEN-ELSE

- **Typische Formen von Bedingungen:**

```
IF bf THEN
  f1;
ELSE
  f2;
END;
```

Alternative "Entweder-Oder" (zweiseitig bedingte Anweisung)

```
IF bf THEN
  f1;
END;
```

einseitig bedingte Anweisung

```
IF    bf1 THEN
  f1;
ELSIF bf2 THEN
  f2;
ELSIF bf3 THEN
  f3;
...
ELSE
  fn;
END;
```

mehrseitig bedingte Anweisung

Die ***Bedingungen*** (bfi) werden der Reihe nach ausgewertet, bis eine ***WAHR*** ist. Dann wird die entsprechende Anweisung (Funktionsaufruf) ausgeführt.

Ist ***keine*** Bedingung wahr, dann wird der ***ELSE-Zweig*** ausgeführt

Bedingungen in funkt. Programmen

Beispiel - 1

- **Aufgabe:**
  - Eingabe ist eine ganze Zahl
  - Stelle fest, ob diese 1-, 2-, 3-, mindestens 4-stellig oder negativ ist!

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF IstEinstellig(i)     THEN RETURN ("einstellig") END;
  IF IstZweistellig(i)    THEN RETURN ("zweistellig") END;
  IF IstDreistellig(i)    THEN RETURN ("dreistellig") END;

  IF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```


Bedingungen in funkt. Programmen

Beispiel - 2

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF    IstEinstellig(i)     THEN RETURN ("einstellig")
  ELSIF IstZweistellig(i)    THEN RETURN ("zweistellig")
  ELSIF IstDreistellig(i)    THEN RETURN ("dreistellig")
  ELSIF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE                            RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  (* Aufruf der Hauptfunktion *)
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```

ELSIF-Konstruktionen machen Bedingungen semantisch klarer

Bedingungen in funkt. Programmen

Beispiel - 3

```
MODULE Stelligkeit EXPORTS Main;
(* Dieses Programm berechnet die Stelligkeit von Zahlen *)
IMPORT IO;

PROCEDURE IstEinstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=0) AND (i<10) );
END IstEinstellig;

PROCEDURE IstZweistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=10) AND (i<100) );
END IstZweistellig;

PROCEDURE IstDreistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=100) AND (i<1000) );
END IstDreistellig;

PROCEDURE IstMinVierstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=1000) );
END IstMinVierstellig;
```


Rekursion

Rekursion

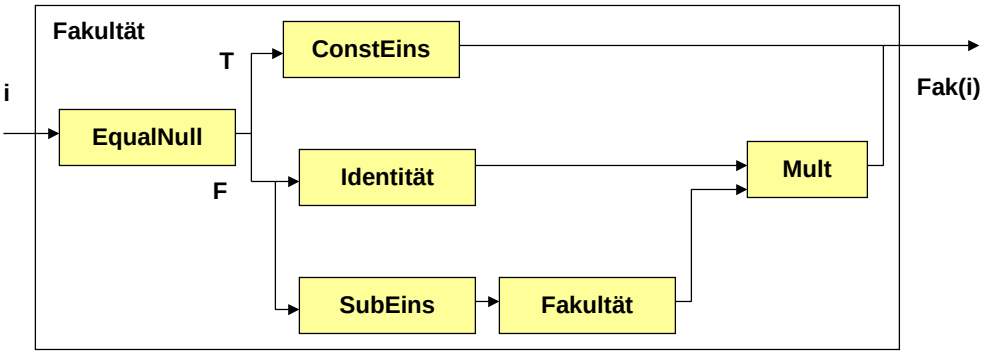
- **Idee:**
  - Allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens "***durch Rückführung auf sich selbst***".
- **Rekursive Funktion**
  - Darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen.***
- **Beispiel: Matrioschka-Puppen**

Anmerkungen

- **Ziel:**
  - Es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen.
  - Dies führt immer zu einem ***Laufzeitfehler.***
- **Konsequenz**
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung.***
    - Rekursionsabbruch: in definierten Fällen wird der rekursive Aufruf nicht ausgeführt
  - Muster:
    -
```
IF … THEN
  rekursiver Aufruf
ELSE
  Rekusionsabbruch
```
- **Bemerkung**
  - Rekursion führt zu ***prägnaten, knappen*** und ***eleganten*** Algorithmen für bestimmte Problemstellungen.

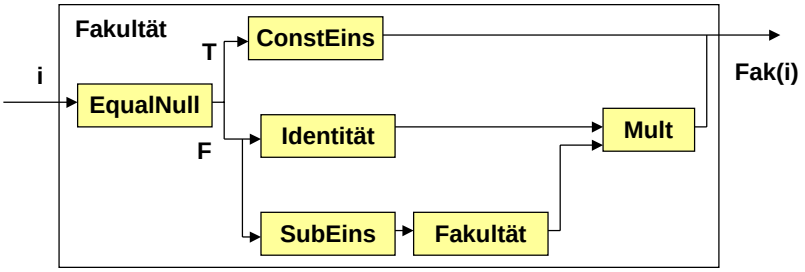
Beispiel Fakultät - 1

- **Fakultät is definiert:**
  - fak : n 1 , falls n = 0
    n * fak(n-1) n ∈ N
- **Funktionsnetz für Fakultät**



Rekursion

Beispiel Fakultät - 2




```
PROCEDURE Fakultaet (i : INTEGER) : INTEGER =
BEGIN
  IF EqualNull(i)
  THEN RETURN(ConstEins(i))
  ELSE RETURN (Mult(Identitaet(i), Fakultaet(SubEins(i))));
  END;
END Fakultaet ;
...
  SIO.PutInt(Fakultaet(SIO.GetInt()));
```


Rekursion

Beispiel Fakultät - 3

```
PROCEDURE EqualNull ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( i = 0 );
END EqualNull;

PROCEDURE ConstEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( 1 );
END ConstEins;

PROCEDURE Mult ( i, j : INTEGER) : INTEGER =
BEGIN
  RETURN ( i * j );
END Mult;

PROCEDURE Identitaet ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i );
END Identitaet;

PROCEDURE SubEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i - 1 );
END SubEins ;
```

Diese Funktionen sind nur im Sinne der **reinen funktionalen** Programmierung notwendig.

Sie können in der Definition der Funktion "Fakultaet" durch entsprechende **Ausdrücke** ersetzt werden!

Rekursion

Beispiel Fakultät - 4

```
MODULE FakultaetM1 EXPORTS Main;
(* Dieses Programm berechnet die Fakultaetsfunktion *)

IMPORT SIO;


PROCEDURE Fakultaet ( i : INTEGER) : INTEGER =
BEGIN
  IF i = 0
  THEN RETURN 1
  ELSE RETURN (i * Fakultaet(i-1));
  END;
END Fakultaet ;


BEGIN
  SIO.PutInt(Fakultaet(SIO.GetInt()));
END FakultaetM1.
```


Rekursion

Fibonacci-Funktion (rekursiv)

- **Wachstum einer Kaninchen-Population**
  - Wieviele Kaninchen-Pärchen gibt es nach n Jahren
    - Jahr 1 : 1 Pärchen
    - Jedes Pärchen hat ab dem zweiten Jahr je ein Pärchen Nachwuchs
  -

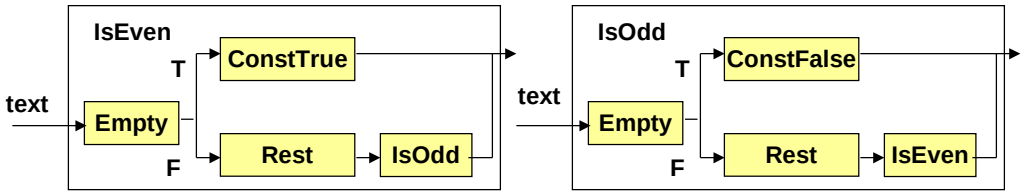
Jahr	1	2	3	4	5	6	7	8	9
Anzahl	1	1	2	3	5	8	13	21	34

```
PROCEDURE Fibonacci (arg: INTEGER): INTEGER =
BEGIN
  IF arg <= 2 THEN
    RETURN 1
  ELSE
    RETURN (Fibonacci (arg -1) + Fibonacci (arg - 2))
  END;
END Fibonacci ;
```

Rekursion

Indirekte Rekursion

- **Definition:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Beispiel:**
  - Die beiden Funktionen (IsEven, IsOdd) sind ***indirekt*** rekursiv.
  - Sie stellen fest, ob eine Zeichenfolge eine gerade oder ungerade Anzahl von Zeichen enthält.



Rekursion

Beispiel - Indirekte Rekursion

```
MODULE EvenOdd EXPORTS Main;
(* Beispiel fuer die indirekte Rekursion *)
IMPORT SIO, Text;

PROCEDURE IsEven (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN TRUE;
  ELSE RETURN (IsOdd (Text.Sub(str,1)));
  END;
END IsEven ;

PROCEDURE IsOdd (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN FALSE;
  ELSE RETURN (IsEven (Text.Sub(str, 1)));
  END;
END IsOdd ;

BEGIN
  SIO.PutBool ( IsEven (SIO.GetWord()));     SIO.Nl();
  SIO.PutBool ( IsOdd  (SIO.GetWord()));
END EvenOdd.
```

Rekursion

Funktional, Rekursion und Iteration

- **Funktionale Algorithmen**
  - kennen keine ***Variablen*** zur Zwischenspeicherung von Ergebnissen.
- **Nichtfunktionale Algorithmen**
  - speichern Zwischenergebnisse in ***Variablen*** ab.
- **Rekursive Algorithmen**
  - lösen ein Problem, in dem sie sich ***selbst wieder aufrufen*** (direkt oder indirekt).
- **Iterative Algorithmen**
  - besitzen Abschnitte, die bei der Ausführung ***mehrmals*** durchlaufen werden
  - Jeder rekursive Algorithmus kann in einen iterativen umgewandelt werden
    - allgemein (siehe Compiler)
    - problemspezifisch

Was haben wir gelernt!

- **Funktionale Programmierung**
  - Funktion, Parameter
  - Vernetzung von Funktionen
- **Anweisungen und Ausdrücke**
- **Datentyp**
  - einfache, zusammengesetzte, vordefinierte, selbstdefinierte
  - einfache (skalare) vordefinierte Datentypen: INTEGER, CARDINAL, REAL, LONGREAL, CHAR, BOOLEAN
- **Fallunterscheidungen: bedingte Anweisungen**
- **Konzept der Rekursion**
  - rekursive Funktionen
  - indirekt rekursive Funktionen
  - Realisierung durch Laufzeitkeller

Glossar

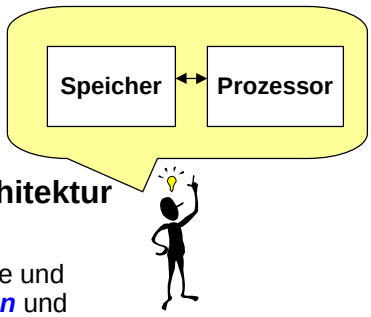
- **Funktionale Programmierung**
- **Funktion, Funktionskopf, -rumpf**
- **Funktionalform**
- **Parameter**
  - Formal- , Aktual-
- **Seiteneffektfreiheit**
- **Rekursion**
  - direkt, indirekt
- **Datentyp**
  - einfacher, elementarer Typ
  - Ordinaltyp
- **Speicherverwaltung nach Kellerprinzip (Laufzeitkeller), statische Speicherverwaltung**
- **Aktivierungsblock**

Imperative Programmierung

- Konzepte der imperativen Programmierung
- Variable und Wertzuweisung
- Prozeduren
- rekursive Prozeduren
- Parameterübergabe
- Gültigkeitsbereich und Lebensdauer

Modell der imperativen Programmierung

- **Synomym:**
  - Befehls-orientierte Programmierung

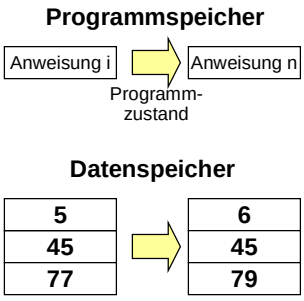


- **Geprägt durch die von-Neumann-Architektur**
  - Die CPU führt ***Maschinenbefehle*** aus
  - Deshalb müssen über den sog. Bus Befehle und Daten vom Speicher in die CPU ***übertragen*** und die Ergebnisse ***rückübertragen*** werden.
- **Mit imperativen Programmiersprachen setzen wir Entwürfe um:**
  - ***Aktionen*** fassen Folgen von Maschinenbefehlen zusammen,
  - ***Variable*** abstrahieren vom physischen Speicherplatz.

Konzepte der imperativen Programmierung

Semantik eines imperativen Progamms

- **Wesentliches Merkmal eines Programms**
  - ***Zustand*** der Daten im Speicher
  - Stand des Befehlszählers
- **Semantik eines Befehls**
  - Übergang von ZUSTAND1 -> ZUSTAND2
- **Programmierer beschreibt einen Prozeß von Zustandsübergängen**
  - durch elementare Anweisungen
  - durch den Kontrollfluß (Programmpfad)
- ***Variable*** **und** ***Wertzuweisung*** **modellieren Zustand und Zustands-übergang im Datenspeicher**



Konzepte der imperativen Programmierung

Imperatives Programmieren

- **Aufgaben des Programmierers**
  - Planung der ***Speicherbelegung***
    - Welche Daten braucht mein Programm?
  - Planung der ***Unterprogramme***
    - Aus welchen Funktionen und Prozeduren soll das Programm bestehen?
    - Wie sollen diese die Werte der Daten verändern?
  - Planung des ***Kontrollflusses***
    - In welcher Reihenfolge sollen die Operationen ausgeführt werden?
  - Planung des ***Datenflusses***
    - Welche Daten müssen von welchen Operationen an andere übergeben werden?

Deklaration von Daten

Deklaration von Unterprogrammen

Anweisungen

Reihenfolge in Programmpfad bzw. Parameterübergabe

Konzepte der imperativen Programmierung

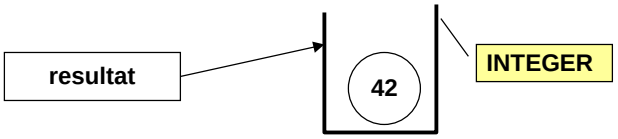
Objekte und Aktionen

- **Wir unterscheiden bei Datenobjekten:**
  - ***Konstante***: Objekte, deren Wert während der Ausführung des Algorithmus unverändert bleibt.
  - ***Variable***: Objekte, deren Wert sich während der Ausführung des Algorithmus verändern kann.
  - Für beide gilt, daß ihre Werte einen ***Typ*** haben. Diese sind zunächst die elementaren Datentypen.
- **Wir unterscheiden bei Aktionen:**
  - Veränderung des Werts einer Variablen durch ***Zuweisung***.
  - Festlegung der nächsten Aktion durch den sog. ***Kontrollfluß*** (Ablaufsteuerung).

Variable und Wertzuweisung

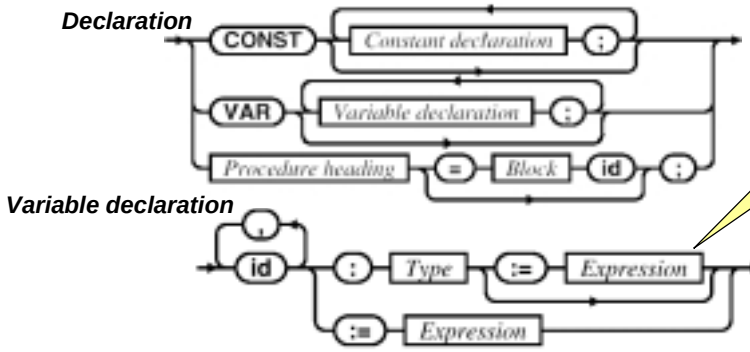
Charakterisierung

- **Variable**
  - Speicherplatz mit seinem Wert
  - besitzt einen **Namen**, unter dem man ihn ansprechen kann
  - bei Sprachen mit Typsystem muß jede Variable einem **Datentyp** zugeordnet sein
  - Datentyp legt fest, welche **Struktur** und **Werte** eine Variable besitzt/ annehmen kann und wie Werte im Programm als Literale/Aggregate hingeschrieben werden
  - Ferner legt ein Datentyp fest, welche **Operationen** ausgeführt werden dürfen
  - Variablen werden im Deklarationsteil von Programmeinheiten (Blöcke, Module) vereinbart (deklariert)
  - Variable kann als Behälter betrachtet werden: hat Wert und Typ



Variable und Wertzuweisung

Deklarationen (erweitert)



Initialisierung der Variablen

Syntaktisch korrekt

Laufzeitfehler?

```
VAR r : INTEGER := 4;
    x : INTEGER := (4 * 8 -2);

    b := FALSE;
    c := TRUE;
    d := (b = c);

    y : REAL;
```

```
VAR x  : INTEGER := 4;
    xx : INTEGER := (2 * x);
    y  : INTEGER := (4 DIV y);
```


Variable und Wertzuweisung

Wertzuweisung

- **Wertzuweisung**
  - dient dazu, den Wert einer Variablen zu verändern
  - Syntax (vereinfacht): **Variable := Ausdruck**
  - Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variablen zugewiesen
    - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
    - LH value, RH value
  - Als Ausdruck auf der rechten Seite verwenden wir meist arithmetische und Boolesche Ausdrücke, Vergleiche und Zeichen bzw. Zeichenketten.
  - ***Typkompatibilität***:
    Der Typ des Bezeichners muß zum Typ des Ausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Variable und Wertzuweisung

Beispiele: Wertzuweisung

- **Deklaration**
  - `VAR x, y, z: CARDINAL;`
- **einige (korrekte und inkorrekte) Wertzuweisungen für x:**
  -
```
x :=  0;
x := MAX (CARDINAL);
x :=  y;   (* sicher richtig *)
```
  -
```
x := -1;
x := MAX (CARDINAL)+1;
x := -y-1; (* sicher falsch  *)
```
  -
```
x := y+z; x := z-y;(* richtig oder falsch,  *)
                        (* je nach Wert von y, z *)
```


Variable und Wertzuweisung

Beispiel: Wertzuweisung

```
MODULE Vertausche1 EXPORTS Main;
(* Vertauscht zwei eingebenene Werte *)

IMPORT SIO;

VAR x, y, h : INTEGER;

BEGIN
  x := SIO.GetInt();
  y := SIO.GetInt();

  h     := x;
  x     := y;
  y     := h;

  SIO.PutText("x = "); SIO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); SIO.PutInt(y);
END Vertausche1.
```

Deklaration der Variablen

Initialisierung der Variablen

Variable und Wertzuweisung

Diskussion der RETURN-Anweisung

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
  VAR min: INTEGER;
BEGIN
  IF m <= n THEN
    min := m;
  ELSE
    min := n
  END;
  RETURN min;
END Minimum ;
```

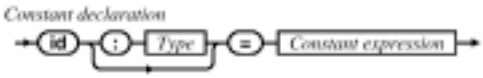
```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
BEGIN
  IF m <= n THEN
    RETURN m
  ELSE
    RETURN n
  END;
END Minimum ;
```

- **Anmerkung:**
  - mehrere RETURN-Anweisungen machen eine Funktion unübersichtlich
- **Empfehlung**
  - Code-Effizienz gegen Lesbarkeit abwägen!

Variable und Wertzuweisung

Symbolische Konstanten

- **Verwendung von Zahlen-Konstanten ("Literalen") in Ausdrücken führt zu Problemen bei der Wartung!**
- **Konstante**



  - Bezeichner mit einem ***festen*** Wert
  - hat einen Datentyp
  - muß deklariert werden
  - überall, wo der Konstantenbezeichner auftritt, wird der Konstantenwert eingesetzt
  - Nach der Deklaration kann ihr ***kein Wert zugewiesen*** werden
- **Beispiel:**

```
CONST PI = 3.141;
VAR umfang, radius : REAL;

…

umfang := 2 * PI * radius
```

```
CONST
  Arbeitstage   = 5;
  Arbeitszeit   = 8;
  Wochenstunden = Arbeitstage *
                  Arbeitszeit;
```

Prozeduren

Der Prozedurbegriff

- **Prozedur ist ein zentraler Begriff der prozeduralen Programmierung.**
- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.
  - Die Parameter erlauben die Prozedur mehrfach zu nutzen

Prozeduren

Algorithmische Abstraktion

- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.
- **Abstraktion wird hier sowohl als *Vorgang* als auch als *Ergebnis des Vorgangs* verstanden:**
  - Im Vorgang der algorithmischen Abstraktion sehen wir von der konkreten Anweisungsfolge ab und bringen diese auf *"**einen Begriff**"*.
  - Das Ergebnis ist eine Entwurfs- und ***Programmeinheit*** – die Prozedur.

Prozeduren

Beispiel: Abstraktionsprozeß

```
Programm Telefonieren
  Hörer_abheben;
  Telefonnummer_wählen;
  Gespräch_führen;
  Hörer_auflegen;
ENDE Telefonieren.
```

Algorithmische Abstraktion durch **Prozeduren**: Statt einer Anweisungsfolge wird ein **Name** verwendet.

Ergebnis
Programmeinheit

```
Prozedur Telefonnummer_wählen
 IF Telefonnummer_gespeichert
THEN
   Kurzwahltaste_drücken
 ELSE
   Telefonnummer_eintippen
 END
ENDE Telefonnummer_wählen.
```


Prozeduren

Kennzeichen von Prozeduren

- **Um den Prozedurbegriff verstehen zu können, benötigen wir drei Begriffe:**
- **Parametrisierung:**
  - Der Mechanismus zum ***Datenaustausch*** zwischen Prozedur und Umgebung.
- **Sichtbarkeit:**
  - Der Programmbereich, in dem ***Namen bekannt*** sind.
- **Lebensdauer:**
  - Der ***Zeitraum***, in dem die Werte von Programmobjekten zugegriffen werden können.

Prozeduren

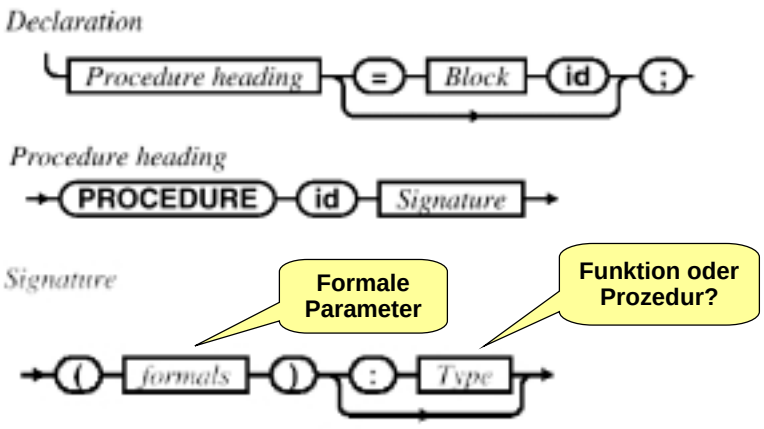
Prozeduren vs. Funktionen

- **Wurden bisher zur Ausgabe verwendet**
  - SIO.PutText ("Hallo")
- **Sind Funktionen "ähnlich"**
  - werden mit PROCEDURE eingeleitet, können auch rekursiv sein
- **Unterschied zu Funktionen**
  - Prozeduren liefern ***kein*** Ergebnis im Sinne eines Funktionsergebnisses!
  - Konsequenz:
    - Prozeduren besitzen keinen ***Ergebnistyp***
    - Aufruf einer Prozedur ist kein Ausdruck, sondern eine Anweisung
- **Zweck einer Prozedur**
  - ***Zusammenfassen*** einer "Funktionalität" (im Sinne der Lokalität)
  - ***Verändern*** der ihr übergebenen Parameter
  - Durchführung einer ***Nebenwirkung*** (z.B. Ausgabe einer Meldung)

Prozeduren

Prozedurdeklaration

- **Syntax:**



- **Beispiele:**

```
PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL) = ...

PROCEDURE Minimum(n,m : INTEGER): INTEGER = ...
```

Prozeduren

Prozeduraufruf: Syntax

- **Beim Aufruf einer Prozedur werden die aktuellen Parameter an die formalen übergeben.**
- **Zur Übersetzungszeit wird überprüft:**
  - Der Name im Aufruf muß ***gleich*** dem Prozedurnamen sein.
  - Die ***Anzahl*** der aktuellen Parameter muß gleich der Anzahl der formalen sein.
  - Die Bindung der jeweiligen Parameter wird entsprechend ihrer ***Position*** im Aufruf und in der Prozedurdeklaration vorgenommen.
  - Die aktuellen Parameter müssen ***typkompatibel*** zu den formalen Parametern sein (d.h. meist typgleich).

```
PROCEDURE Minimum ( m, n : INTEGER) : INTEGER = …

res := Minimum (x, y); (* korrekter Aufruf)
res := Mini (x, y);
res := Minimum (x, y, z);
```

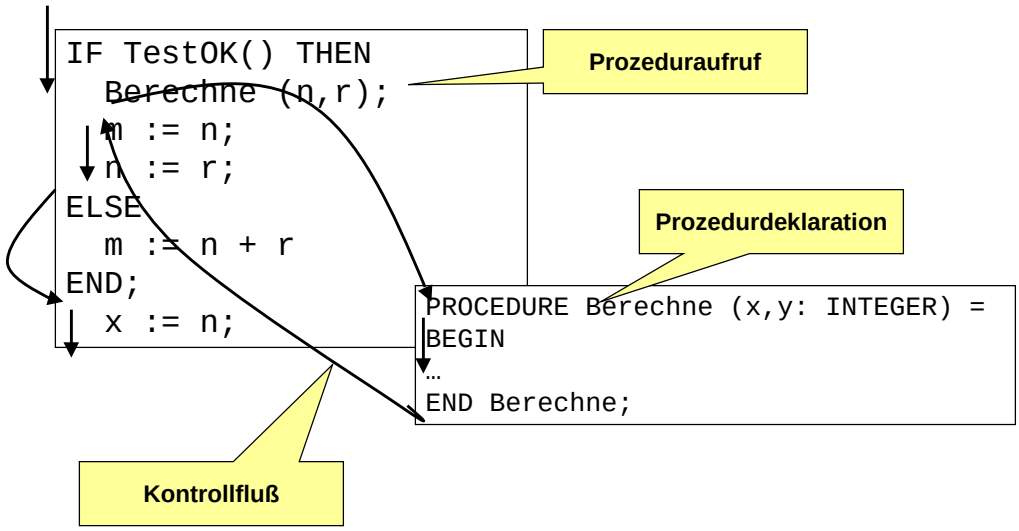

Prozeduren

Prozeduraufruf: Semantik

- **Der Prozeduraufruf ist die explizite Anweisung,**
  - daß die Prozedur ***ausgeführt*** werden soll.
- **Eine Prozedur ist *aktiv*,**
  - nachdem sie gerufen wurde und in der Ausführung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- **Für den Prozeduraufruf in imperativen Sprachen ist charakteristisch:**
  - Beim Aufruf wechselt die ***Kontrolle*** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die aktuellen Parameter an die formalen ***gebunden***.
  - Prozeduren können wieder Prozeduren aufrufen. Dabei wird der Aufrufer unterbrochen (suspended), so daß die Kontrolle stets bei einer Prozedur ist.
  - Nach der Ausführung der Prozedur kehrt die Kontrolle zum Rufer zurück; die Ausführung wird mit der ***Anweisung nach dem Aufruf*** fortgesetzt.

Prozeduren

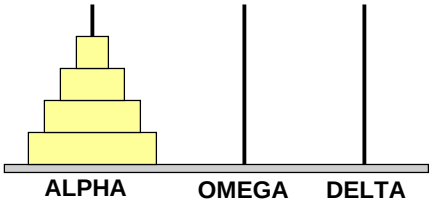
Beispiel: Prozeduraufruf



Rekursive Prozeduren

Rekursion: Aufgabe

- **Aufgabe: Türme von Hanoi**
  - bewege die Scheiben des Turms von ALPHA nach OMEGA
  - es darf immer ***nur eine Scheibe*** bewegt werden
  - niemals darf eine Scheibe auf eine kleinere bewegt werden



- **Lösungsstrategie (Divide & Conquer)**
  - allgemeine Lösung für einen Turm der Höhe h von ALPHA nach OMEGA
    - h = 0 gar nichts machen
    - h > 0 1. Turm der Höhe h-1 von ALPHA nach DELTA über OMEGA
      2. Scheibe von ALPHA nach OMEGA legen
      3. Turm der Höhe h-1 von DELTA nach OMEGA über ALPHA

Rekursive Prozeduren

Rekursion: Programmtext

```
MODULE Hanoi EXPORTS Main;
(* Ausgabe der Zugfolge fuer Tuerme von Hanoi *)

IMPORT SIO;

PROCEDURE DruckeZug (hoehe: CARDINAL; von, nach : TEXT)  =
BEGIN
  SIO.PutText ("Scheibe "); SIO.PutInt (hoehe);
  SIO.PutText (" von " & von & " nach " & nach); SIO.Nl();
END DruckeZug ;

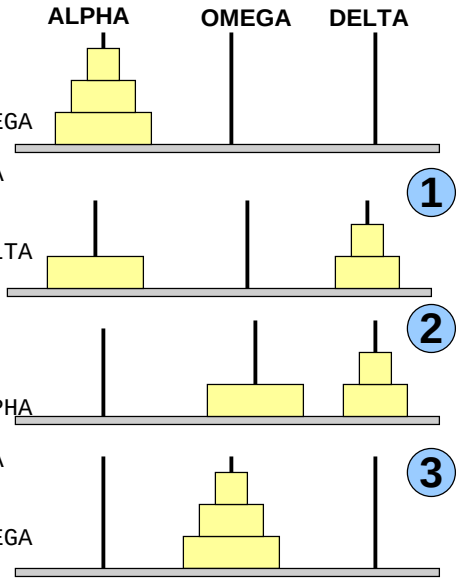
PROCEDURE BewegeTurm ( hoehe : CARDINAL; von, nach, ueber: TEXT)  =
BEGIN
  IF hoehe > 0 THEN
    BewegeTurm (hoehe-1, von, ueber, nach);
    DruckeZug  (hoehe, von, nach);
    BewegeTurm (hoehe-1, ueber, nach, von);
  END;
END BewegeTurm;

BEGIN
  BewegeTurm(SIO.GetInt(), "ALPHA", "OMEGA", "DELTA" );
END Hanoi.
```


Rekursive Prozeduren

Rekursion: Laufzeitgeschehen

```
3 ALPHA OMEGA DELTA
  ❶ 2 ALPHA DELTA OMEGA
    ❶ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
    ❷ Scheibe 2 von ALPHA nach DELTA
    ❸ 1 OMEGA DELTA ALPHA
      ❶ 0 OMEGA ALPHA DELTA
      ❷ Scheibe 1 von OMEGA nach DELTA
      ❸ 0 ALPHA DELTA OMEGA
  ❷ Scheibe 3 von ALPHA nach OMEGA
  ❸ 2 DELTA OMEGA ALPHA
    ❶ 1 DELTA ALPHA OMEGA
      ❶ 0 DELTA OMEGA ALPHA
      ❷ Scheibe 1 von DELTA nach ALPHA
      ❸ 0 OMEGA ALPHA DELTA
    ❷ Scheibe 2 von DELTA nach OMEGA
    ❸ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
```



Parameter-übergabe

Parameter

- **Bisher**
  - Funktionen besitzen ausnahmslos ***Eingabeparameter***
  - Wert dieser Parameter kann nicht ***geändert*** werden
- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
    - vor dem Aufruf wird der aktuelle Parameter ausgewertet und dem formalen Parameter zugewiesen (***call-by-value***)
  - ***Ausgabeparameter***
    - dienen dazu, Ergebnisse einer Prozedur an den Aufrufer zurückzugeben
    - Wert ist zum Zeitpunkt des Aufrufs undefiniert (***call-by-reference***)
  - ***Ein- / Ausgabeparameter (Transienten)***
    - vereinen Eigenschaften beider Arten
- **Modula-3 nutzt dazu zwei Parameterübergabemechanismen**

Parameter-übergabe

Übergabemechanismen: Call by Value

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
    - realisieren Eingangsparameter
    - Der aktuelle Parameter muß ein ***Ausdruck*** sein (Spezialfall: Variable, d.h. Bezeichner für ein Objekt).
    - Beim Aufruf der Prozedur wird ein dem Typ des formalen Parameters entsprechendes ***lokales Objekt*** angelegt. Ist der aktuelle Parameter eine Variable, so entsteht dabei eine ***Kopie*** des Parameter-Objekts.
    - In jedem Falle wird der Wert des aktuellen Parameters ***berechnet*** und dem formalen Parameter (-Objekt) zugewiesen.
    - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
    - Schlüsselwort ***VALUE*** zeigt einen Wertparameter an
    - steht kein Schlüsselwort, ist es ***per default*** ein Wertparameter

Parameter-übergabe

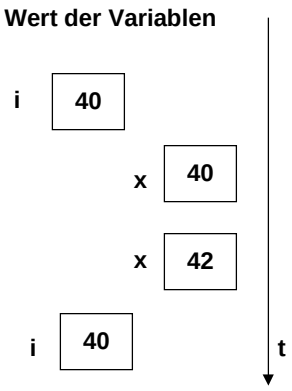
Beispiel: Call by Value

```
PROCEDURE Proc1 (VALUE x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Nichts passiert")
ELSE
  SIO.PutLine ("kein Call by Value")
END;
```



Parameter-übergabe

Übergabemechanismen: Call by Reference

- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
    - realisieren Ausgangs und Ein- / Ausgangsparameter
    - Der aktuelle Parameter muß ein ***Bezeichner für ein Objekt*** sein.
    - Beim Aufruf wird der formale Parameter durch einen ***Verweis*** auf den aktuellen Parameter ersetzt.
      D.h. der formale Parameter wird als lokaler Bezeichner für das aktuelle Parameterobjekt ***substituiert***.
    - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.
    - Veränderungen des formalen Parameters in der Prozedur haben auf den aktuellen Parameter Auswirkung, d.h. Objekte im Namensraum des Rufers können ***verändert*** werden. Auf diese Weise können von einer Prozedur ***Ergebnisse*** zurückgegeben werden.
    - Schlüsselwort ***VAR*** zeigt einen Variablenparameter an

Parameter-übergabe

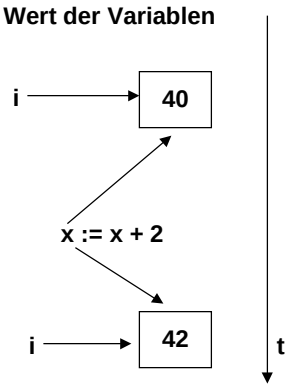
Beispiel: Call by Reference

```
PROCEDURE Proc1 (VAR x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

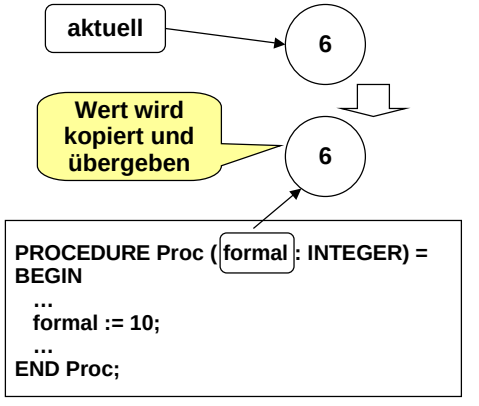
IF i = 40 THEN
  SIO.PutLine ("Call by Value")
ELSE
  SIO.PutLine ("Call by Reference")
END;
```

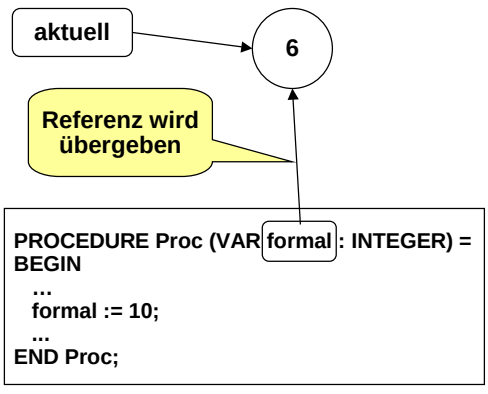


Parameter-übergabe

Call-by-value <-> Call-by-reference

```
VAR aktuell : INTEGER
…
aktuell := 6;
Proc (aktuell);
```





Parameter-übergabe

Beispiel: Wert- Variablenparameter

```
MODULE Parameter EXPORTS Main;
IMPORT IO, SIO;

VAR aktuell: INTEGER;

PROCEDURE Proc1 (VALUE formal : INTEGER) =
BEGIN
  formal := 10;
END Proc1;

PROCEDURE Proc2 (VAR formal : INTEGER) =
BEGIN
  formal := 10;
END Proc2;

BEGIN
  aktuell := 6;        Proc1 (aktuell);
  SIO.PutText("aktuell (Proc1) = "); IO.PutInt(aktuell); SIO.Nl();
  aktuell := 6;        Proc2 (aktuell);
  SIO.PutText("aktuell (Proc2) = "); IO.PutInt(aktuell);
END Parameter.
```

Wertparameter

Variablen-parameter

aktuell (Proc1) = 6
aktuell (Proc2) = 10

Ausgabe des Programms

Parameter-übergabe

Beispiel: Variablenparameter

```
MODULE Vertausche2 EXPORTS Main;
IMPORT IO, SIO;

PROCEDURE Vertausche (VAR w1, w2 : INTEGER) =
VAR hilfe : INTEGER := w1;
BEGIN
  w1    := w2;
  w2    := hilfe;
END Vertausche;

VAR x, y : INTEGER;

BEGIN
  x := IO.GetInt();
  y := IO.GetInt();
  Vertausche (x, y);
  SIO.PutText("x = "); IO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); IO.PutInt(y);
END Vertausche2.
```

Vertauscht die Werte der beiden Parameter

Parameter-übergabe

Beispiel: Ausgabeparameter

- **Ausgabeparameter**
  - dient dazu, einen Wert an den Aufrufer zurückzugeben

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR  ergebnis : REAL;

PROCEDURE Quadrat ( VALUE x : REAL; VAR wert : REAL )=
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Ausgabeparameter oder Ein- Ausgabeparameter ?

Wert ist beim Aufruf undefiniert

Parameter-übergabe

Datenaustausch: Beispiel - 1

In einer Reihe von Meßwerten soll der **laufende Mittelwert** berechnet werden. Benötigte Objekte und Aktionen:

```
Wert, Mittelwert, Summe : REAL;
Anzahl : INTEGER;
(* BerechneMWert
      Addiere neuen Wert und Summe,
      Erhöhe Anzahl um 1,
      Mittelwert := Summe / Anzahl *)
```

Austausch über Parameter:

```
PROCEDURE BerechneMWert (     wert     : REAL;
                          VAR anzahl    : INTEGER;
                          VAR summe, mw : REAL) =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  mw := summe / anzahl;
END BerechneMWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;
BerechneMWert (wert,anzahl,summe,mw);
BerechneMWert (2*wert,anzahl,summe,mw);
```

Parameter-übergabe

Datenaustausch: Beispiel - 2

Datenaustausch über Funktionsergebnis:

```
PROCEDURE MWert (    wert  : REAL;
                 VAR anzahl: INTEGER;
                 VAR summe : REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl:= anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;

SIO.PutReal (MWert(wert,anzahl,summe));
...
SIO.PutReal (MWert(2*wert,anzahl,summe));
```


Parameter-übergabe

Datenaustausch: Beispiel - 3

Datenaustausch über Funktionsergebnis:

```
VAR summe: REAL; anzahl: INTEGER;

PROCEDURE MWert (wert:REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0;

SIO.PutReal (MWert(4.0);
...
SIO.PutReal (MWert(10.0));
```

Parameter-übergabe

Diskussion der Beispiele - 1

- **Beispiel 1:**
  - Die Verwendung von VAR-Parametern in einer Prozedur zur Rückgabe von Ergebnissen ist in der imperativen Programmierung üblich.
  - Sie führt oft zu ***Verständnisproblemen***, da sowohl in der Deklaration als auch in der Verwendung klar sein muß, was das eigentliche Ergebnisobjekt (hier: der Mittelwert) ist.
- **Beispiel 2:**
  - Die Verwendung einer Funktion zur Berechnung genau eines Wertes ist dann sauber, wenn alle anderen Parameter als ***Wertparameter*** verwendet werden.
  - Die Modellierung einer Zustandsveränderung (hier: Summe und Anzahl) außerhalb der Funktion mit Hilfe von VAR-Parametern ist ein ***Seiteneffekt***, der die ***Lokalität*** der Funktion zerstört.

Parameter-übergabe

Diskussion der Beispiele - 2

- **Beispiel 3:**
  - Die Verwendung von ***globalen*** Variablen, die in einer Prozedur oder Funktion als Seiteneffekt verändert werden, ist die ***schlechteste*** Lösung.
  - Zustandsveränderungen über globale Variablen, die in der Signatur der Prozedur nicht aufgeführt sind, sind unverständlich und extrem ***fehleranfällig***.
- **Funktionsprozeduren**
  - Eine Funktion kann in imperativen Sprachen nur dann sauber modelliert werden, wenn
    - alle Parameter als ***Wertparameter*** übergeben werden,
    - in der Funktion ***keine globalen Variablen*** verwendet werden.
  - In Funktionen sollte auch keine globalen Variablen ***lesend verwendet*** werden, da dies Funktionen an ihren Verwendungskontext ankoppelt.
  - Merke: Funktionen sollten "***in sich***" verständlich sein.

Parameter-übergabe

Regeln für die Parameterverwendung

- **Wertparameter sind Variablenparameter vorzuziehen**
  - Änderung an Parametern in Prozeduren ist häufig eine ***Fehlerquelle***, die schwer zu finden ist.
- **Variablenparameter sollten nur verwendet werden, wenn**
  - Prozedur-Ergebnisse ***übergeben*** werden sollen (Ausgabeparameter)
  - Kopieren des Parameters ***nicht möglich*** ist (bei gewissen Datenstrukturen)
  - Kopieren zu ***ineffizient*** ist (bei sehr großen Datenstrukturen)
- **Funktionen haben nur Wertparameter**
  - liefern ihr Ergebnis durch ihren ***Namen*** zurück
  - geht in Modula-3 nur, wenn
    - genau ein Wert zurückgegeben werden soll
  - Rückgabe durch Namen und Variablenparameter muß ***vermieden*** werden

Parameter-übergabe

Zusammenfassung: Prozeduren und Parameter

- **In imperativen Sprachen wird oft die Prozedur in den Vordergrund gestellt.**
- **Die Funktion ist gelegentlich (z.B. Modula-3) nur eingeschränkt verwendbar, kann dafür aber *Seiteneffekte* erzeugen (nicht wünschenswert).**
- **Nur durch eine saubere Definition der *Signaturen* von Routinen kann ein verständlicher und weiterverwendbarer Entwurf erreicht werden, d.h. vor allem, jeder Datenaustausch mit der Umgebung sollte *explizit* sein.**
- **Die Modellierung von Zuständen und ihrer Veränderung ist nur in Verbindung mit einem entsprechenden *Modulkonzept* softwaretechnisch sauber zu lösen.**

  Das lernen wir später!

Gültigkeitsbereich und Lebensdauer

Def. Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - ***Sichtbarkeitsbereich*** ist Teil des Gültigkeitsbereichs
  - Gültigkeitsbereich/Sichtbarkeitsbereich wird durch den Compiler überwacht
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
  - macht nur Sinn für Objekte, die Speicher belegen
    - Typen belegen keinen Speicher
- **Es ist wichtig, beide Begriffe klar zu unterscheiden!**

Gültigkeitsbereich und Lebensdauer

Gültigkeitsbereich: lokale Deklarationen

- **Regeln für die Gültigkeit von Bezeichnern;**
  - Alle in einer Prozedur oder einem Modul deklarierten Bezeichner sind in der gesamten Prozedur / im gesamten Modul gültig.
  - Das gilt auch für den textuell vor der Deklaration eines Bezeichers liegenden Bereich
  - Davon ausgenommen sind Prozedur- und Modulkopf

```
PROCEDURE Proc ( in : REAL )=

  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Hier können die Bezeichner X und C verwendet werden.

```
PROCEDURE Proc ( in : REAL )=

  CONST CC : C * C
  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Korrekte Vorwärts-referenz

Gültigkeitsbereich und Lebensdauer

Gültigkeitsbereich: Erweiterung

- **Durch IMPORT kann der Gültigkeitsbereich von Bezeichnern auf das *importierende Modul* erweitert werden.**

```
INTERFACE SIO;
...
PROCEDURE GetReal …;

PROCEDURE PutReal ...;

PROCEDURE GetText ...;

PROCEDURE PutText ...;

…
END SIO;
```

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR   ergebnis : REAL;

PROCEDURE Quadrat … =
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Qualifizierter Bezeichner muß verwendet werden.

Gültigkeitsbereich und Lebensdauer

Prozeduren als Namensraum - 1

- **Namensraum**
  - Eine Prozedur bildet gegenüber der (textuellen) Umgebung ihrer Deklaration einen eigenen ***Namensraum***, d.h. sie kann lokale Objekte benennen und verwalten.
  - Die Namen der formalen Parameter sowie die im Deklarationsteil des Prozedurrumpfs deklarierten Bezeichner sind nur ***im Prozedurrumpf gültig***, d.h. bekannt.
  - In einer Prozedur können Bezeichner der Umgebung ***neu lokal*** deklariert werden; diese Bezeichner ***verdecken*** die global deklarierten Bezeichner, die zugehörigen Objekte sind in der Prozedur ***nicht sichtbar***.
- **Lebensdauer**
  - Die Lebensdauer von prozedurlokalen Objekten entspricht dem Zeitraum, in dem der Aufruf der Prozedur abgearbeitet wird. Sie werden zu Beginn der Prozedurausführung angelegt.
  - Werte gehen ***verloren***, wenn die Ausführung der Prozedur beendet ist.

Gültigkeitsbereich und Lebensdauer

Prozeduren als Namensraum - 2

```
PROCEDURE Proc ()=

  VAR x : INTEGER;

  PROCEDURE Proc1 (x :Integer) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Proc;
```

G2

G3

G1

- **Module und Prozeduren definieren eigene Namensräume**
  - Prozeduren können verschachtelt werden
  - Hierarchie von Gültigkeits-bereichen (Namensräumen)
- **Überlappung von Gültigkeitsbereichen**
  - Liegt innerhalb des Gültigkeitsbereiches G1 von X mit der Bedeutung B1 ein weiterer Gültigkeitsbereich G2 von X mit der Bedeutung B2, so handelt es sich um zwei ***verschiedene*** Objekte
  - innerhalb von G2 gilt nur B2, alle anderen Bedeutungen sind ***unsichtbar***.

Gültigkeitsbereich und Lebensdauer

Lebensdauer - 1

- **Durch Ausführung einer Prozedur P entsteht eine *Inkarnation*.**
- **Zur Inkarnation gehören *zur Laufzeit*:**
  - ein ***Ausführungspunkt*** (also ein Zeiger auf den gerade auszuführenden oder ausgeführten Befehl)
  - ***Speicherplätze*** für alle Bezeichner von Variablen und Wertparameter
  - ***Bezüge*** auf die konkreten Variablenparameter.
- **Informationen existieren bis zum Ende der Ausführung von P.**
- **Beispiel**
  - `PROCEDURE Test (ch: CHAR; VAR x:INTEGER)=`
    `VAR y: REAL`
- **`Test ('a', z)` führt zu einer Inkarnation mit**
  - Speicherplatz für ch und y
  - unter dem lokalen Bezeichner `x` einen Bezug (einer Referenz) auf z
  - nach Abschluß werden diese Speicherplätze wieder freigegeben

Gültigkeitsbereich und Lebensdauer

Lebensdauer - 2

- **Bemerkungen:**
  - **Sei M ein Modul,**
    - dann wird ***Speicherplatz*** für die Variablen permanent für die gesamte Laufzeit des Programms reserviert.
    - Eine eigentliche Inkarnation wird nur von dem Rumpf des Moduls gebildet; dieser hat weder Variablen noch Parameter.
  - Zu irgendeinem Zeitpunkt existieren i.a. neben der Inkarnation des ablaufenden Moduls ***Inkarnationen verschiedener*** Prozeduren, bei **rekursiven** Aufrufen auch mehrere der gleichen Prozedur.
  - Nur in der ***jüngsten*** aller existierenden Inkarnationen wandert der Ausführungspunkt weiter; diese wird als erste beendet.
  - Die Lebensdauer einer Variablen ist ***identisch*** mit der Existenz der zugehörigen Prozedur-Inkarnation.
  - Zu Beginn der Lebensdauer ist der Wert einer Variablen ***undefiniert***, darf also nicht verwendet werden.

Gültigkeitsbereich und Lebensdauer

Beispiel: Lebensdauer - 1

```
MODULE LDUGB EXPORTS Main;
IMPORT SIO;
VAR i  , j  : INTEGER;

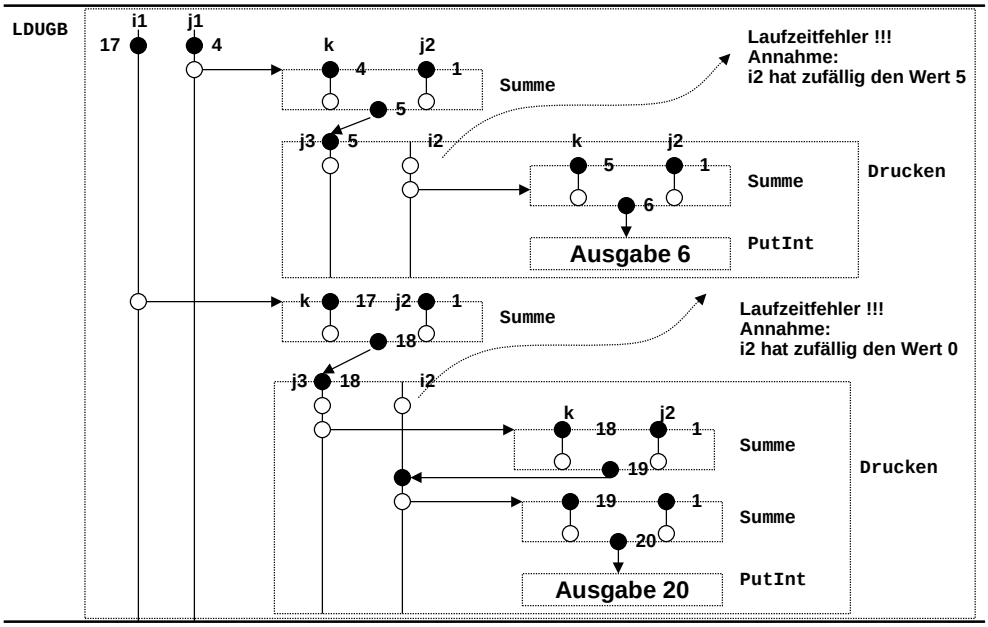
PROCEDURE Summe (k  : INTEGER) : INTEGER =
VAR j  : INTEGER;
BEGIN
  IF i  <10 THEN j  := 10 ELSE j  :=1 END;
  RETURN j  + k  ;
END Summe;

PROCEDURE Drucken (j  : INTEGER) =
VAR i  : INTEGER;
BEGIN
  IF i   #   j   THEN i  := Summe(j  ) END;
  SIO.PutInt (Summe(i  )); SIO.Nl();
END Drucken;

BEGIN
  i  := 17; j  := 4;
  Drucken (Summe(j  ));
  Drucken (Summe(i  );
END LDUGB.
```

Gültigkeitsbereich und Lebensdauer

Beispiel: Lebensdauer - 2



Gültigkeitsbereich und Lebensdauer

Terminologie

- **"Objekt" bezeichnet alles,**
  - was durch einen Bezeichner eingeführt wird (Modul, Prozedur, Konstante, Typ, Variable, Parameter),
  - keine Objekte sind demnach Operatoren oder Wortsymbole (z.B. VAR, END)
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

Gültigkeitsbereich und Lebensdauer

Beispiel: lokal, global, global relativ

```
MODULE Gueltigkeit EXPORTS Main;
  IMPORT SIO;
  VAR x : INTEGER;

  PROCEDURE Proc1 (x: INTEGER) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Gueltigkeit.
```

x ist global sichtbar im Modul

x ist lokal in der Prozedur Proc1

x ist global relativ in der Prozedur Proc2

y ist lokal in der Prozedur Proc2

Gültigkeitsbereich und Lebensdauer

Lokalität

- **Ziel**
  - Programme sollten mit ***möglichst wenig*** Aufwand korrigier- und modifizierbar sein.
- **Strategie**
  - ***hohe Lokalität*** durch enge Gültigkeitsbereiche.
  - Größtmögliche Lokalität ist daher ***vorrangiges Ziel*** einer guten Programmierung!
- **Ein Programm sollte dafür folgende Merkmale aufweisen:**
  - Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind ***überschaubar***.
  - Die Objekte sind so ***lokal*** wie möglich definiert, jeder Bezeichner hat nur eine ***einzige***, bestimmte Bedeutung.
  - Die ***Kommunikation*** zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variable.

Gültigkeitsbereich und Lebensdauer

Vorteile lokaler Variabler

- **Lokale Variablen haben softwaretechnisch einige Vorteile.**
  - Deklaration und Verwendung stehen in einem ***textlichen*** Zusammenhang. Das erhöht die Lesbarkeit.
  - Die ***unfreiwillige*** Verwendung von globalen Variablen wird vermieden, d.h. globale Variablen müssen nicht vollständig bekannt sein.
  - Der Speicherverbrauch durch Prozeduren wird ***minimiert***, da Speicher für lokale Variablen nur während ihrer Aktivierung vorgehalten werden muß.
  - Lokalität:
    - Variablen sollen ***nur in dem Kontext*** deklariert werden, wo sie bekannt sein müssen. Eine Verteilung erschwert die Änderbarkeit.
  - Kapselung:
    - Außerhalb eines Kontextes (Modul, Prozedur) sollen nur relevante Objekte sichtbar sein. Implementationsdetails werden im Inneren ***verborgen*** und sind nicht zugreifbar.

Was haben wir gelernt!

- **Modell der imperativen Programmierung**
  - Zustand, Zustandsübergang
- **Konzept der Variablen**
  - Wertzuweisung
- **Parameterübergabemechanismen**
  - Wertparameter
  - Variablenparameter
  - Wie geht man damit um
- **Gültigkeit von Bezeichnern**
- **Lebensdauer von Objekten**
- **Konzept der Lokalität**

Glossar

- **Modelle der imperativen Programmierung (von-Neumann-Sprachen)**
- **Programm- und Datenspeicherzustände und -übergänge bei imperativen Programmen**
- **imperative Programmierung: Aufgaben und Hilfsmittel der Programmiersprache**
- **Programmiersprachliche Objekte**
- **Variablenbegriff: typisiert, Initialisierung, LH-Value, RH-Value, Wertzuweisung, Veränderung bei Parameterübergabe**
- **Deklarationen: verschiedene Arten, Zweck dieser Arten**
- **Konstantenbegriff von Modula-3**
- **Prozeduren: Aufgaben, Parameter, Gültigkeit, Lebensdauer, Unterscheidung zu Funktionen**
- **Prozedurdeklaration: Prozedurkopf/Signatur, Prozedurrumpf, Formalparameterliste, lokale Deklarationen, Verwendung der Formalparameter im Rumpf**
- **Prozeduraufruf: Aktualparameterliste, Konsistenz, Aufruf und Deklaration, Semantik durch Inkarnation**
- **Parameter: Arten, Übergabemechanismen, Methodikregeln, Call-by-Value (Aufruf über den Wert), Call-by-Reference (Aufruf über die Adresse)**
- **rekursive Prozeduren (direkt und indirekt): Verwendung bei Divide & Conquer-Entwicklungsstrategie, statisches Verständnis der Rekursion, Laufzeitgeschehen bei rekursiven Prozeduren**
- **Datenaustausch über Prozeduren**
- **Namensraum von Prozeduren, Gültigkeitsbereich, Erweiterung durch Importe, lokale Variable, Vorteile im Sinne der Programmiermethodik**
- **Lebensdauer von Prozedurinkarnationen, darin enthaltener Variablen**

Kontroll-strukturen

- **Ablaufkontrolle**
- **Fallunterscheidungen**
  - IF
  - CASE
- **Wiederholungsanweisungen (Schleifen)**
  - WHILE, FOR
  - REPEAT-UNTIL, LOOP-EXIT

Ablaufkontrolle

Kontrollfluß

- **Ablaufsteuerung in der von Neumann-Maschine:**
  - Befehle stehen ***hintereinander*** im Speicher, werden vom Steuerwerk in den zentralen Prozessor geholt, dort decodiert und verarbeitet.
  - Durch ***Sprungbefehle*** kann von der Reihenfolge der gespeicherten Befehle abgewichen werden.
- **Abstraktion:**
  - Die Ausführungsreihenfolge der Aktionen eines Algorithmus entspricht zunächst der textuellen Anordnung (***Sequenz***). Davon kann aber abgewichen werden. Dazu gibt es eigene Anweisungen.
  - Ablaufsteuerung:
    - Fallunterscheidung,
    - Wiederholungen.
- **Ablaufsteuerung in imperativen Programmiersprachen:**
    - Anweisungsfolgen,
    - Fallunterscheidungen **`IF`**- und **`CASE`**-Anweisungen,
    - Schleifenformen **`WHILE`**-, **`UNTIL`**-, **`LOOP`**-Anweisungen.

Ablaufkontrolle

Kontrollstrukturen

- **Berechnungen in imperativen Programmen erfolgen durch die *Auswertung von Ausdrücken* und die *Zuweisung* von Werten zu Variablen.**
- **Zur Erhöhung der Flexibilität von Programmen sind zwei weitere Sprachmechanismen notwendig:**
  - die Steuerung des Kontrollflusses zur ***Auswahl*** zwischen unterschiedlichen Anweisungen, (Fallunterscheidungen)
  - die ***wiederholte*** Ausführung einer Folge von Anweisungen.
- **Sprachmechanismen, die dies leisten,**
  - heißen ***Kontrollstrukturen***.
- **Kontrollstrukturen**
  - zusammen mit den Anweisungsfolgen, die von ihnen kontrolliert werden, stellen den Anweisungsteil eines Programms dar.

Ablaufkontrolle

Beispiel GOTO (Sprunganweisung)

- **FORTRAN-Programm zur Bewertung von Meßdaten**

```
Beispiel GOTO in einem FORTRAN-Programm
100   format()
200   format(56h anzahl messwerte gut brauchbar schlecht unbrauchbar)
      ischl = 0
      ibr = 0
      igut = 0
      do 1 i = 1,n
      read (5,100) x
      abso = abs(x - s);
      if (abso .le. 0.01) goto 10
      if (abso .le. 0.05) goto 11
      if (abso .le. 0.2)  goto 12
      iunb = iunb + 1
      goto 1
10    igut = igut + 1
      goto 1
11    ibr = ibr + 1
      goto 1
12    ischl = ischl + 1
1     continue
      ...
```

Ablaufkontrolle

Diskussion: Kontrollstrukturen

- **Von Mitte der 60er bis Mitte der 70er wurde in der Informatik viel über *Kontrollstrukturen* diskutiert.**
- **Ein Ergebnis war:**
  - Obwohl eine einzige ***Anweisungsform*** (bedingter oder unbedingter Sprung) ausreicht, sollte genau diese Anweisung durch eine ***kleine Zahl*** von Kontrollanweisungen ***ersetzt*** werden.
- **Boehm und Jacopini haben 1966 nachgewiesen, daß alle Flußdiagramm-Programme durch zwei Kontrollstrukturen implementierbar sind:**
  - ***Auswahl*** zwischen alternativen Kontrollflüssen,
  - logisch kontrollierte ***Wiederholung***.
- **Kontrollstrukturen sollen *einen Einstieg* und *einen Ausstieg* (single entry, single exit) besitzen.**

Ablaufkontrolle

Diskussion Goto

- **Obwohl die *unbedingte Verzweigung* (Goto) ausreicht,**
  - alle anderen Kontrollstrukturen nachzubilden, führt ihre uneingeschränkte Verwendung zu unlesbaren und unzuverlässigen Programmen.
- **Hauptgrund:**
  - Durch Goto kann im Ablauf ***jede beliebige Reihenfolge*** von Anweisungen unabhängig von ihrer textlichen Anordnung erreicht werden.
  - In seinem Artikel ("Goto statement considered harmful", CACM, 1968, Vol.11, No.3, pp.147-149) schreibt E.W. Dijkstra: "The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."
- **Dies hat die Goto-Debatte entzündet,**
  - die zwar zur softwaretechnischen Ablehnung des ***uneingeschränkten Goto*** geführt hat,
  - aber nur ***wenige*** Programmiersprachen haben völlig auf dieses Konstrukt verzichtet (Modula-2, Modula-3, Java).

Ablaufkontrolle

Konzept: Sequenz

- **Sequenz von Aktionen:**
  - Eine Aktion wird ***nach der anderen*** abgearbeitet.
  - Dazu muß nur klar sein, wie zwei Aktionsbeschreibungen voneinander ***getrennt*** sind.
  - Ein Aktion ist auch "tue nichts".

- **Beispiel:**
  - Algorithmus Telefonieren:
    - `hebe den Hörer ab;`
    - `wähle die Telefonnummer;`
    - `führe das Gespräch;`
    - `lege den Hörer auf`

Trennzeichen

Fallunter-scheidung

Konzept: Bedingte Anweisung (if)

- **Fallunterscheidungen kommen vor als:**
  - Einwegauswahl (one-way selection)
  - Zweiwegauswahl (two-way-selection)
  - allgemeine bedingte Anweisung

```
IF b THEN
  stmts;
ELSE
  stmts;
END;
```

Zweiwegauswahl, zweiseitig bedingte Anweisung

```
IF b THEN
  stmts;
END;
```

Einwegauswahl, einseitig bedingte Anweisung

...

allgemeine bedingte Anweisung

- **Anwendbar**
  - Typ des Ausdrucks, der die Auswahl bestimmt, ist BOOLEAN

Fallunter-scheidung

Auswahlanweisung (case)

- **Die *Auswahlanweisung* ermöglicht die Auswahl aus einer *beliebigen* Anzahl explizit angegebener Alternativen.**
- **Designentscheidungen sind:**
  - Welche Form und welcher Typ von Ausdruck ***kontrolliert*** die Mehrfachselektion?
  - Welche Form von Anweisung kann ***ausgewählt*** werden?
  - Wie ist der ***Kontrollfluß*** innerhalb der Mehrfachselektion?
  - Wie werden Selektionswerte behandelt, für die es ***keine*** passende Anweisungsalternative gibt?
- **Programmiersprachen realisieren die Auswahl durch die**
  - CASE-Anweisung

Fallunter-scheidung

CASE-Anweisung - 1

- **Hängen die Fälle einer Fallunterscheidung nur von unterschiedlichen Werten *eines Ausdrucks* ab,**
  - können wir die in modernen imperativen Sprachen vorhandene *CASE-Anweisung* verwenden.
- **Der ELSE-Teil ist im Beispiel leer, um einen *Fehlerfall* zu vermeiden.**
- **Dies ist eine häufige, aber *schlechte* Programmiertechnik.**
- **Besser:**
  - möglichen Fehlerfall explizit behandeln.

```
CASE zweierpotenz OF

  0 => ergebnis := 1;
| 1 => ergebnis := 2;
| 2 => ergebnis := 4;
| 3 => ergebnis := 8;
| 4 => ergebnis := 16;
ELSE
END;
```

Fallunter-scheidung

CASE-Anweisung - 2

```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  ergebnis := 0;

  CASE zweierpotenz OF
     0 => ergebnis := 1;
   | 1 => ergebnis := 2;
   | 2 => ergebnis := 4;
   | 3 => ergebnis := 8;
   | 4 => ergebnis := 16;
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;

  SIO.Nl(); SIO.PutInt(ergebnis);
END CaseDemo.
```

Bemerkung:

Werden nicht alle Werte als Alternativen aufgeführt und fehlt des ELSE-Zweig, dann kann das zu *Laufzeitfehlern* führen!!

Fehlerbehandlung im ELSE-Zweig ???

Fallunter-scheidung

CASE-Anweisung - 3

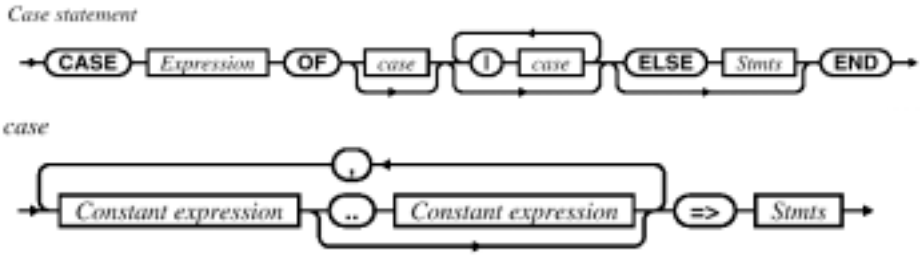
```
MODULE CaseDemo EXPORTS Main;
IMPORT SIO;
VAR zweierpotenz, ergebnis: INTEGER;

BEGIN
  zweierpotenz := SIO.GetInt();
  IF (zweierpotenz >= 0 AND zweierpotenz <= 4) THEN
    CASE zweierpotenz OF
      0 => ergebnis := 1;
    | 1 => ergebnis := 2;
    | 2 => ergebnis := 4;
    | 3 => ergebnis := 8;
    | 4 => ergebnis := 16;
    END;
    SIO.PutInt(ergebnis);
  ELSE
    SIO.PutText ("Wert nicht definiert!");
  END;
END CaseDemo.
```

Fehlersituation wird explizit vor Ausführung der CASE-Anweisung behandelt!

Fallunter-scheidung

Syntax CASE-Anweisung



Auswahlisten: Bequemlichkeit!

- **Zusatzbedingungen für die Case-Anweisung:**
  - Ergebnis von *Expression* und Ergebnis der *Constant expressions* müssen ***denselben*** Typ haben.
  - Zugelassen sind nur ***Ordinaltypen*** (z.B. BOOLEAN oder CHAR)
  - Die Werte dürfen jeweils ***nur einmal*** auftreten.

Schleifen

Konzept: Schleifen

- **Schleifen (Wiederholungsanweisungen, Iterationen) werden benötigt,**
  - um ***iterative Algorithmen*** zu formulieren.
- **Die historisch ersten Sprachkonstrukte zur Wiederholung waren**
  - für die ***Array-Bearbeitung*** gedacht,
  - da anfangs vorrangig numerische Probleme gelöst werden mußten.
- **Designentscheidungen sind:**
  - Wie wird die Wiederholung kontrolliert?
    - durch einen ***logischen Ausdruck***,
    - durch ***Abzählen***
  - Wo steht der Kontrollmechanismus im Programmtext?
    - am ***Anfang***/***Ende***,
    - ***automatisch***/***benutzerdefiniert***

Schleifen

Zählschleife (FOR-Anweisung)

- **Die FOR-Anweisung ist eine spezielle Schleifenform**
  - Anzahl der Wiederholungen ist ***im voraus*** bekannt
  - Wiederholungen werden durch Abzählen kontrolliert
- **Syntax:**

- **Anmerkungen:**
  - Die Steuerung und der Abbruch geschieht mit Hilfe der sog. ***Laufvariablen***, deren Schrittweite und Laufrichtung festgelegt werden kann.
  - In Modula-3: Die Laufvariable muß ***nicht deklariert*** werden.
  - Bei jedem Durchlauf wird die Laufvariable "***automatisch***" erhöht oder erniedrigt.
  - FOR-Schleifen werden oft bei der Bearbeitung von ***Arrays*** verwendet.

Schleifen

Beispiel: FOR-Anweisung

- **Ein Beispiel für die FOR-Schleife:**

```
(* Berechne Summe I = 1 bis 100 ueber I *)
r := 0;
FOR i := 1 TO 100 DO
  r := r + i
END;
```

- **FOR-Schleife mit Schrittweite:**

```
(* Berechne Summe I = 1 bis 100 ueber I *)
r := 0;
FOR i := 100 TO 1 BY -1 DO
  r := r + i
END;
```

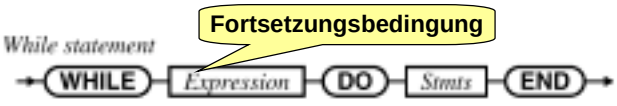
Innerhalb der FOR-Schleife muß die Laufvariable wie eine ***Konstante*** behandelt werden, darf also nicht auf der linken Seite einer Wertzuweisung oder als Referenzparameter stehen und damit ***manipulierbar*** sein.

Schleifen

Bedingte Schleife mit vorheriger Prüfung

- **WHILE-Schleife**
  - "***ablehnende Schleife***", da Prüfung vor Schleifendurchlauf gemacht wird
- **Syntax:**

```
WHILE E DO
    S;
END;
```



- **Bemerkung**
  - Ergebnis der "Expression" muß vom ***Typ BOOLEAN*** sein
  - Der Programmierer muß dafür sorgen, daß das Ergebnis von "Expression" ***irgendwann FALSE*** ist; ansonsten Endlosschleife; Programm terminiert nicht
- **Semantik (rekursiv definiert):**

```
IF E THEN
  S;
  WHILE E DO S; END;
END;
```


Schleifen

Beispiel: WHILE-Anweisung

- **Aufgabe:**
  - Man berechne den Quotienten und den Rest der ganzzahligen Division zweier positiver ganzer Zahlen a und b.

```
MODULE GanzzahlDivision EXPORTS Main;
IMPORT SIO;
VAR a, b, c: INTEGER;

BEGIN
  a := SIO.GetInt();
  b := SIO.GetInt();

  c := 0;  (*enthalte den Quotienten *)
  WHILE a >= b DO
    a := a - b;
    c := c + 1;
  END;

  SIO.PutText(" Quo :");  SIO.PutInt(c);
  SIO.PutText(" Rest :"); SIO.PutInt(a);

END GanzzahlDivision.
```

Wiederhol-bedingung

Schleifen

Bedingte Schleife mit nachfolgender Prüfung

- **REPEAT-UNTIL Anweisung (UNTIL-Schleifen)**
  - "***annehmende***" Schleife, da eine erste Ausführung stattfindet, ohne daß die Bedingung (in diesem Fall eine ***Abbruch-Bedingung***) geprüft wird
- **Syntax:**

- **Bemerkung**
  - REPEAT-UNTIL erfordert besondere Vorsicht
  - REPEAT-UNTIL ist immer dann sinnvoll, wenn der Bedingungswert erst durch die ***Anweisungen der Schleife*** entsteht
  - z.B. Eingabe mit Fehlerbehandlung

Schleifen

Beispiel: UNTIL

```
MODULE Einleseschleife EXPORTS Main;
IMPORT SIO;
VAR a: INTEGER;

BEGIN

  REPEAT
    SIO.PutText("Geben Sie eine Zahl zwischen 1 und 5 ein! ");
    SIO.Nl();
    a := SIO.GetInt();
  UNTIL (a >=1 AND a <=5);

  SIO.PutText("Ihre Eingabe war korrekt! ");

END Einleseschleife.
```

Abbruch-bedingung

Schleifen

Vergleich WHILE - UNTIL

```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t : TEXT := "";
BEGIN
  c := SIO.GetChar();
  WHILE c # Punkt DO
    t := t & Text.FromChar(c);
    c := SIO.GetChar();
  END;
  RETURN t;
END Einlesen;
```

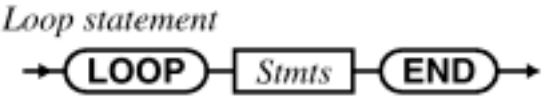
```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t : TEXT := "";
BEGIN
  c := SIO.GetChar();
  IF c # Punkt THEN
    REPEAT
      t := t & Text.FromChar(c);
      c := SIO.GetChar();
    UNTIL c = Punkt;
  END;
  RETURN t;
END Einlesen;
```

```
PROCEDURE Einlesen () : TEXT =
  CONST Punkt = '.';
  VAR c : CHAR; t, cText : TEXT := "";
BEGIN
  REPEAT
    t := t & cText;
    c := SIO.GetChar();
    cText := Text.FromChar(c);
  UNTIL c = Punkt
  RETURN t;
END Einlesen;
```


Schleifen

Endlosschleife

- **LOOP-Anweisung**
  - die Schleife wird solange durchlaufen, bis eine darin enthaltene ***EXIT-Anweisung*** ausgeführt wird
- **Syntax:**



- **Bemerkungen:**
  - in einer LOOP-Anweisung können ***mehrere*** EXIT-Anweisungen stehen
  - single-entry - single-exit wird dadurch verletzt
  - ist keine EXIT-Anweisung enthalten, so realisiert die LOOP-Anweisung eine ***nichtterminierende Schleife***

Schleifen

Beispiel: LOOP-Anweisung - 1

Algorithmus GGT

❶ Falls n < m ist, so vertausche man n und m

❷ Falls m = 0 ist, dann ist n der ggt(n,m) und man beende den Algorithmus

❸ Falls m≠ 0 ist, so bilde man den Rest r, der bei der Division von n durch m bleibt, dann ersetze man n durch m und m durch r und beginne von vorn.

```
MODULE GGT EXPORTS Main;
VAR n, m, hilf, r : INTEGER;
BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  LOOP
    IF n < m THEN  (* 1 *)
      hilf := m;
      m := n;
      n := m;
    END;
    IF m = 0 THEN  (* 2 *)
      EXIT;
    ELSE           (* 3 *)
      r := n MOD m;
      n := m;
      m := r;
    END;
  END (* LOOP *);
  SIO.PutText("GGT = ");SIO.PutInt(n);
END GGT.
```


Schleifen

Beispiel: LOOP-Anweisung - 2

Bemerkungen

- LOOP-Anweisungen terminieren nicht!
- Die Ausführung der EXIT-Anweisung terminiert die LOOP-Anweisung. Die Kontrolle geht zur Anweisung ***nach dem END***.
- Bei geschachtelten LOOP-Strukturen bewirkt die EXIT-Anweisung nur das Verlassen der ***zugehörigen*** LOOP-Struktur.

Empfehlung

- Aus Gründen der besseren Lesbarkeit sollte man überall dort WHILE- und REPEAT-Schleifen zu verwenden, wo nur eine Abfrage am ***Anfang*** oder ***Ende*** der Schleife erforderlich ist!

```
MODULE GGT1 EXPORTS Main;
IMPORT SIO;
VAR n, m, r : INTEGER;

BEGIN
  n := SIO.GetInt();
  m := SIO.GetInt();
  r := n MOD m;
  LOOP
    IF r = 0 THEN
      EXIT;
    ELSE
      n := m;
      m := r;
      r := n MOD m;
    END;
  END;
  SIO.PutText ("GGT = ");
  SIO.PutInt(m);
END GGT1.
```

Schleifen

LOOP versus WHILE

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
LOOP
  IF r = 0 THEN
    EXIT;
  ELSE
    n := m;
    m := r;
    r := n MOD m;
  END;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

```
n := SIO.GetInt();
m := SIO.GetInt();

r := n MOD m;
WHILE r # 0 DO
  n := m;
  m := r;
  r := n MOD m;
END;

SIO.PutText ("GGT = ");
SIO.PutInt(m);
```

- **Hier bietet sich die WHILE-Anweisung an,**
  - da am Beginn der Schleife die Bedingung geprüft werden soll.

Schleifen

Wahl des Schleifenkonstrukts

- **REPEAT...UNTIL**
  - ist mit Vorsicht zu verwenden, denn die Anweisung in der Schleife wird ***mindestens einmal*** durchlaufen.
  - Sie ist nur sinnvoll, wenn der Bedingungswert ***erst in der Schleife entsteht***
  - kann durch Schleife ohne Prüfung realisiert werden (sollte man aber nicht!)

```
REPEAT
  SIO.PutText("Zahl zwischen 1 und 5! ");
  SIO.Nl();
  a := SIO.GetInt();
UNTIL (a >=1 AND a <=5);

LOOP
 SIO.PutText("Zahl zwischen 1 und 5! ");
 SIO.Nl();
 a := SIO.GetInt();
 IF (a >=1 AND a <=5) THEN EXIT END;
END;
```

Schleifen

Was haben wir gelernt!

- **Fallunterscheidungen**
  - einseitig bedingte IF-THEN
  - zweiseitig bedingte IF-THEN-ELSE
  - allgemeine bedingte mit ELSIF
  - Auswahlanweisung CASE
- **Wiederholungsanweisungen (Schleifen)**
  - WHILE
  - FOR
  - REPEAT-UNTIL
  - LOOP-EXIT
- **Einsatz der Wiederholungsanweisungen**
  - Die Wahl der geeigneten Ausdrucksmittel hängt vom jeweiligen Konstruktionsproblem ab.
  - Implementationsgesichtspunkte sind ggf. zu berücksichtigen.
  - Softwaretechnische Überlegungen wie Verständlichkeit und Sicherheit spielen bei der Verwendung eine zentrale Rolle.

Glossar

- **Ablaufkontrolle, Kontrollfluß, zugehörige Kontrollstrukturen für zusammengesetzte Anweisungen, Kontrollstrukturen als Konstruktoren für die Ablaufkontrolle**
- **Fallunterscheidung und Zusammenführung im Kontrollfluß**
- **Kontrollstrukturen für Fallunterscheidungen: bedingte Anweisung (einseitig, zweiseitig, mehrseitig), Auswahlanweisung (Mehrfachselektion), Steuerung des Kontrollflusses durch Boolesche Ausdrücke bzw. Aufzählen der Fälle durch Auswahllisten**
- **GOTO-Kontroverse, wohlstrukturierte Programme (Sequenz, Fallunterscheidung, Iteration), wohlstrukturierte Programme mit Escape (Exit), Umwandlung beliebiger Programme in wohlstrukturierte**
- **Schleifenformen: Zählschleife, WHILE-Schleife, UNTIL-Schleife, Endlosschleife**
- **Anwendungsfälle für verschiedene Schleifenformen**
- **Termination bei verschiedenen Schleifenformen**

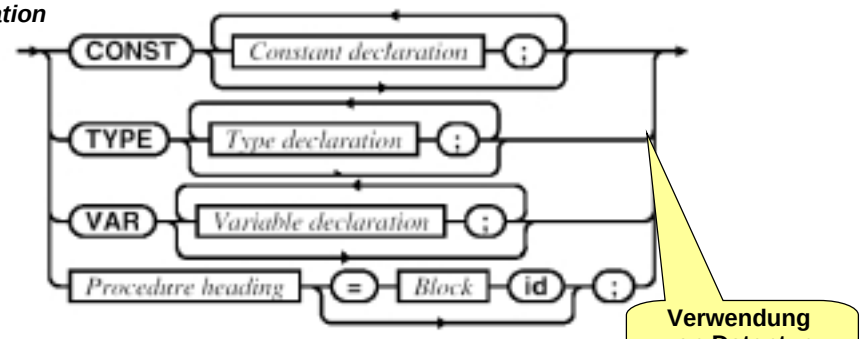
Datentypen I

- **Datentypen: Allgemeines**
- **Skalare benutzerdefinierte Datentypen**
  - Aufzählungstyp
  - Unterbereichstyp
- **Zusammengesetzte benutzerdefinierte Datentypen**
  - ARRAY-Type
  - RECORD-Type
  - SET-Type

Deklaration von Typen

- **Typdeklaration:**
  - Um das vorhandene ***Typkonzept*** erweitern zu können, sind in vielen imperativen Sprachen Typen selbst ***Programmobjekte***, die deklariert werden müssen.
  - Um Typen deklarieren zu können, benötigt man ***Datentypkonstruktoren***

Declaration



Datentypen: Allgemeines

Einteilung von Typen - 1

- **In imperativen Programmiersprachen unterscheidet man einfache und zusammengesetzte Datentypen:**
  - ***Einfache Datentypen*** erlauben ***keinen*** Zugriff auf ihre innere Struktur. Ihre Werte können unmittelbar notiert werden. Die in einer Programmiersprache vorgegebenen einfachen Datentypen heißen elementar (skalar).
  - **Zusammengesetzte Datentypen** sind aus anderen Datentypen aufgebaut. Auf ihre einzelnen Elemente kann zugegriffen werden. Letztlich werden sie auf einfache Datentypen zurückgeführt.
- **Vorgegebene und benutzerdefinierte Datentypen:**
  - **Vordefinierter Datentypen** haben einen vordeklarierten Namen und können unmittelbar zur Deklaration von Variablen verwendet werden.
  - **Benutzerdefinierte** Datentypen haben einen selbst definierten Namen und müssen deklariert werden. Sie werden mit Hilfe bereits deklarierter vorgegebener oder benutzerdefinierter Datentypen gebildet.

Datentypen: Allgemeines

Einteilung von Typen - 2

- **Statische Datentypen**
  - Größe der Typobjekte ist von vornherein ***bekannt***
  - ***Statische*** Typkonstruktoren
    - ARRAY
    - RECORD
    - SET
- **Dynamische Datentypen**
  - Größe ist während der Laufzeit ***veränderbar***
  - Typkonstruktor für ***dynamische Datentypen***
    - Zeiger
    - Pointer

Skalare benutzerdefinierte Datentypen

Merkmale benutzerdefinierter Typen

- **Benutzerdefinierte Typen**
  - erlauben die Vergabe ***anwendungsbezogener*** Namen,
    - z.B. `Zeit` statt `REAL`,
  - sind ein wesentlicher ***Abstraktionsmechanismus***, da sie die programmiersprachliche Realisierung von Datenstrukturen ***verbergen*** können,
    - später werden wir das Konzept der Abstrakten Datentypen kennenlernen
  - sind "***Baumuster***" für die Erzeugung anwendungsbezogener Datenstrukturen,
    - z.B. Struktur

```
Zugverbindung =  Abfahrt: Zeit;
                 Ankunft: Zeit;
```

  - liefern "***Sprachelemente***" für die anwendungsbezogene Modellierung von Softwaresystemen.

Skalare benutzerdefinierte Datentypen

Benutzerdefinierte einfache Typen

- **Modula-3 erlaubt,**
  - benutzerdefinierte einfache Typen unter Verwendung einen vorgegebenen elementaren Typs zu deklarieren.
  -
```
TYPE     Zeit  = REAL;
         Alter = CARDINAL;
```

Basistyp muß ein Ordinaltyp sein

- **Unterbereichstyp (subrange type)**
  - benutzerdefinierte einfache Typen können als ***Einschränkung*** des Wertebereichs eines elementaren Typs deklariert werden
  -
```
TYPE     Index = [1..10];
         Alter = [1 .. 120];
```

- **Aufzählungstyp (emumeration type)**

Ordinaltyp

  - benutzerdefinierte einfache Typen können durch ***Aufzählung*** der zulässigen Werte deklariert werden
  -
```
TYPE Ampelfarbe   = {rot, gelb, gruen};
     Parteien     = {CDU, SPD, Gruene, FDP, PDS}
```

Skalare benutzerdefinierte Datentypen

Operationen auf Aufzählungstypen

- **Aufzählungstypen**
  - werden systemintern auf nichtnegative Zahlen abgebildet
  - (z.B.: rot ->1, blau -> 2, gruen -> 3 oder 0, 1, 2).
  - Dadurch sind die Werte von Aufzählungstypen dann ***vergleichbar*** – was aber selten Sinn macht (rot < gruen).
- **Bezeichner der Werte von Aufzählungstypen können bei *mehreren* Typen auftreten.**
  - `TYPE Ampelfarbe = {rot, gelb, gruen};`
  - `TYPE Parteifarbe = {rot, gelb, gruen, schwarz};`

```
VAR a : Ampelfarbe;
    p : Parteifarbe;
    a := Ampelfarbe.gruen;
    p := Parteifarbe.gruen
```

  - Gilt nicht für viele andere imperative Sprachen!

Skalare benutzerdefinierte Datentypen

Operationen auf Unterbereichstypen

- **Regel:**
  - Für einen Unterbreichstyp sind alle die Operationen definiert, die auch für seinen ***Basistyp*** definiert sind.
- **Es ist häufig unsinnig,**
  - ***arithmetische*** Operationen unmittelbar auf Unterbereichstypen anzuwenden, auch wenn dies die Sprache zuläßt (z.B. Addition zweier Jahreszahlen).
  -
```
TYPE AeraKohl = [1982 .. 1998];
VAR wahljahr1, wahljahr2, jahr : AeraKohl;

wahljahr1 := 1982; wahljahr2 := 1986;
jahr := wahljahr1 + wahljahr2;
```

    Laufzeitfehler

  - Vorsicht bei arithmetischen Operationen auf Unterbereichstypen, dies führt oft zu ***Laufzeitfehlern***.

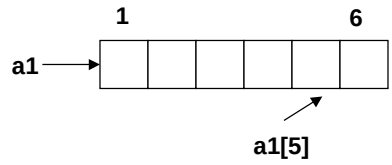
Zusammengesetzte benutzerdefinierte Datentypen

Feldtypen

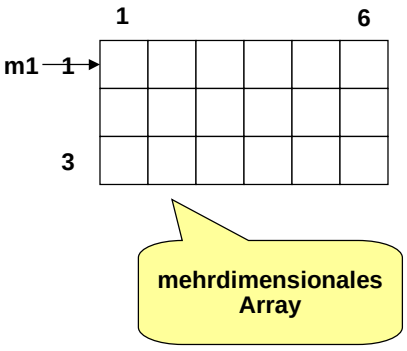
- **Definition:**
  - Nach Informatik-Duden: Feld (Reihung, engl. array): Aneinanderreihung von ***gleichartigen Elementen***, wobei auf die Komponenten mit Hilfe eines ***Indexausdrucks*** zugegriffen wird.
- **Eigenschaften von Arrays (Feldern) in Modula-3:**
  - Die Anzahl der Elemente ist ***fest*** und heißt ***Länge*** des Array.
  - Der ***Name*** einer Array-Variablen bezeichnet das gesamte Array.
  - Ein einzelnes Array-Element wird durch einen ***Index*** bzw. mehrere Indizes (im Fall mehrdimensionaler Arrays) identifiziert.
  - Zur Indizierung kann jeder ***Ordinaltyp*** verwendet werden.
    - INTEGER, CARDINAL, CHAR, BOOLEAN, Aufzählungs- und Unterbereichstypen

Zusammengesetzte benutzerdefinierte Datentypen

Beispiele: Array




```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;
VAR a1 : Vector
```




```
TYPE Spalte  = [1 .. 6];
     Zeile   = [1 .. 3];

TYPE
  Matrix = ARRAY Spalte, Zeile
           OF INTEGER;

VAR m1 : Matrix;
```

Zusammengesetzte benutzerdefinierte Datentypen

Felder und Zählschleifen

- **Beispiel:**
  - Initialisieren eines zweidimensionalen Arrays

```
TYPE Spalte  = [1 .. 6];
     Zeile   = [1 .. 3];

TYPE Matrix = ARRAY Spalte, Zeile OF INTEGER;

PROCEDURE Initialisieren (VAR m : Matrix) =
BEGIN
  FOR i := FIRST(Spalte) TO LAST(Spalte) DO
    FOR j := FIRST(Zeile) TO LAST(Zeile) DO
      m [i,j] := 0;
    END;
  END;
END Initialisieren;
```


Zusammengesetzte benutzerdefinierte Datentypen

Felder als zusammengesetzte Typen

- **Betrachten wir Arrays als zusammengesetzte Typen, dann stellen wir fest:**
  - Der ***Typkonstruktor***, der in Deklarationen benutzt wird, ist in Modula-3 (vereinfacht):

    ```
    <ArrayTyp> = ARRAY Index OF Komponenten;
    ```

  - Als ***Selektor*** für ein einzelnes Element wird die Indexangabe verwendet (indizierter Zugriff, Indizierung):

    ```
    val := a1[3] (* Wert des 3. Elements von a1 *)
    ```

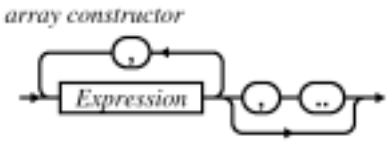
  - Üblicherweise wird die Indexangabe auch für die ***selektive Zuweisung*** (Feldkomponentenzuweisung) verwendet:

    ```
    a1[4]:= 42 (* 4. Elements von a1 wird 42 *)
    ```

Zusammengesetzte benutzerdefinierte Datentypen

Feldaggregate

- **Mithilfe eines sog. *Feldaggregats* können konstante ARRAY-Objekte erzeugt und Feldobjekte initialisiert werden:**
  - `VAR x := Array_Type { e1, .., en}`
    e1 bis en sind Ausdrücke; ihr Wert wird den Feldelementen initial zugewiesen



Feldvariableninitialisierung

```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;

VAR a1 := Vector {0, 0, 0, 0, 0, 0};
    a2 := Vector {0, ..}
```

```
TYPE Spalte = [1 .. 6];
     Reihe  = [1 .. 3];

TYPE Matrix = ARRAY Spalte , Reihe OF INTEGER;
VAR m1 := Matrix {ARRAY Reihe OF INTEGER {0, ..}, ..};
```


Zusammengesetzte benutzerdefinierte Datentypen

Operationen auf Feldobjekten

- **Zuweisung**
  - zwei ARRAY-Objekte sind ***zuweisungskompatibel***, wenn sie
    - den gleichen ***Komponententyp*** und die
    - gleiche ***Gestalt*** haben (gleiche Anzahl Elemente in jeder Dimension)
- **Vergleich**
  - zuweisungskompatible Arrays können auf ***Gleichheit*** und ***Ungleichheit*** geprüft werden.

```
TYPE Index = [1 .. 6];
     Vector = ARRAY Index OF INTEGER;
CONST A = ARRAY [11 .. 16] OF INTEGER {1,2,3,4,5,6};
VAR v : Vector;
…
  v := A;

  IF v = A THEN
```

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Array - 1

```
MODULE StundenPlan EXPORTS Main;
IMPORT SIO;

TYPE
    Tage      = {Montag, Dienstag, Mittwoch, Donnerstag, Freitag};
    Stunden   = [7..20];
    Vormittag = [8..12];
    Fächer    = {Keine, Englisch, Software_1, Mathematik};
    Plan      = ARRAY Tage, Stunden OF Fächer;

CONST
   TagNamen  = ARRAY Tage OF TEXT {"Montag", "Dienstag","Mittwoch",
                                   "Donnerstag", "Freitag"};
   FachNamen = ARRAY Fächer OF TEXT {"Keine","Englisch","Software_1","Mathematik"};

VAR stundenPlan : Plan;          (*Speichert den Stundenplan*)
```

Typdeklarationen

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Array - 2

```
BEGIN
  FOR tag:= FIRST(Tage) TO LAST(Tage) DO
    FOR stunde:= FIRST(Stunden) TO LAST(Stunden) DO
      stundenPlan[tag, stunde]:= Fächer.Keine;    (*Initialisierung auf Keine*)
    END; (*FOR stunde*)
  END; (*FOR tag*)

  FOR stunde:= 8 TO 18 DO    (*Fast den ganzen Montag Englisch*)
    stundenPlan[Tage.Montag, stunde]:= Fächer.Englisch;
  END; (*FOR stunde*)

  FOR tag:= Tage.Dienstag TO Tage.Freitag DO
    stundenPlan[tag, 10]:= Fächer.Software_1;
  END; (*FOR tag*)

  stundenPlan[Tage.Dienstag, 8]:= Fächer.Mathematik;
  stundenPlan[Tage.Freitag, 9]:= Fächer.Mathematik;

  FOR tag:= FIRST(Tage) TO LAST(TAGE) DO
    SIO.PutText(TagNamen[tag]& "\ n");
    FOR stunde:= FIRST(Vormittag) TO LAST(Vormittag) DO
      SIO.PutInt(stunde);
      SIO.PutText(":" & FachNamen[stundenPlan[tag, stunde]]);
    END; (*FOR stunde*)
    SIO.NL();
  END; (*FOR tag*)
END StundenPlan.
```

Zusammengesetzte benutzerdefinierte Datentypen

Verbundtypen

Definitionen

- Nach Informatik-Duden:
  - Record *(Verbund, Struktur, Datensatz)*: ***Zusammenfassung*** von mehreren Datentypen zu einem Datentyp. Der neue Wertebereich ist das ***kartesische Produkt*** der Wertebereiche der einzelnen Datentypen, wobei die Anordnung keine Rolle spielt.
- Nach Sebesta:
  - A record is a ***possibly heterogeneous*** aggregation of data elements in which the individual elements are identified by **names**.
- Nach Ludewig:
  - Records (Verbunde) sind ***heterogene*** kartesische Produkte und dienen zur Darstellung ***inhomogener***, aber ***zusammengehöriger*** Informationen. Typische Beispiele sind
    - Personendaten (Name, Adresse, Jahrgang, Geschlecht)
    - Meßwerte (Zeit, Gerät, Wert)
    - Strings (tatsächliche Länge, Inhalt)

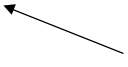
Zusammengesetzte benutzerdefinierte Datentypen

Verbunde in Modula-3

Record in Modula-3:

- Datentyp, der eine Sammlung von Elementen auch ***verschiedenen*** Typs (Elementtyp) repräsentiert.
- Der ***Name*** einer Record-Variablen bezeichnet den gesamten Record.
- Ein einzelnes Record-Element heißt auch ***Komponente*** (record field) und wird durch einen ***Namen*** (Selektorname, field identifier) bezeichnet.
- Von außen wird ein Feld über seinen Bezeichner mit der sog. ***Punktnotation*** (dot notation) angesprochen:
- `<RecordName>"."<FeldName>` z.B. `Person.Vorname`

Anrede	Vorname	Name	PersNr
Herr	Franz	Mustermann	4711
Dr.	Josef	Wanninger	4712
Frau	Susanne	Mitternacht	4713



ein Record

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: RECORD - 1

```
MODULE RecordDemo EXPORTS Main;

TYPE PersNr = [4700 .. 9999];
TYPE Anrede = {Frau, Herr, Dr};
TYPE Name   = TEXT;

TYPE Person = RECORD
                anrede   : Anrede;
                vorname  : Name;
                nachname : Name;
                persnr   : PersNr;
              END;

VAR person1 : Person;
BEGIN
  person1.anrede   := Anrede.Dr ;
  person1.vorname  := "Josef";
  person1.nachname := "Wanninger";
  person1.persnr   := 4712;
END RecordDemo.
```

Typdefinition

Typdeklaration

Faßt unterschiedliche Typen zu einer gemeinsamen Struktur zusammen.

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: RECORD - 2

```
TYPE PersNr = [4700 .. 9999];
TYPE Anrede = {Frau, Herr, Dr};

TYPE Name   = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : Pers;
              END;
VAR person1 : Person;

person1.anrede        := Anrede.Dr ;
person1.name.vorname  := "Josef";
person1.name.nachname := "Wanninger";
person1.persnr        := 4712;
```

- **Bemerkung:**
  - Ziel ist, Typen so zu konstruieren, daß sie möglichst sinnvoll ***aufeinander*** aufbauen.
  - Vorteile:
    - erleichterte Modifikation
    - Wiederverwendbarkeit
    - bessere Lesbarkeit
    - Begriffe der Anwendung können verwendet werden

Zusammengesetzte benutzerdefinierte Datentypen

Records als zusammengesetzte Typen

- **Betrachten wir Records als zusammengesetzte Typen, dann stellen wir fest:**
  - Der ***Typkonstruktor***, der in Deklarationen benutzt wird, ist in Modula-3 (vereinfacht):

    RECORD <Feldname> : <Basistyp> {;<Feldname> : <Basistyp>} END

  - Der ***Selektor*** für ein Feld eines Records, der bei der Verwendung benutzt wird, ist in Modula-3:

    <RecordBezeichner> . <FeldName>

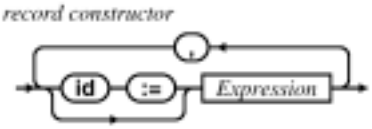
  - Eine rekursive Typdeklaration eines Records ist ***nicht möglich***:

```
Liste = RECORD Listenkopf : CHAR;
               Listenrest : Liste;
         END (* geht nicht *)
```


Zusammengesetzte benutzerdefinierte Datentypen

Verbundaggregate

- **Mit dem *Verbundaggregat* werden initialisierte RECORD-Objekte erzeugt:**
  - `VAR x := Record_Type { Bindings }`
    Bindings: analog zur Bindung der aktuellen Parameter an formale Parameter beim Prozeduraufruf




```
TYPE Name   = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
VAR n1 := Name { vorname := "Josef",  nachname := "Maier"};

TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : PersNr;
              END;

VAR p1 := Person {anrede := Anrede.Herr,
                  name   := Name { vorname := "Kai",  nachname := "Blau"},
                  persnr := 4700 };
```

Angabe der Werte per Name

Zusammengesetzte benutzerdefinierte Datentypen

Operationen auf RECORDs - 1

- **Zuweisung**
  - zwei RECORD-Objekte sind ***zuweisungskompatibel***, wenn
    - alle Komponenten den gleichen ***Namen*** und den gleichen Typ haben
    - alle Komponenten in der gleichen ***Reihenfolge*** deklariert sind
- **Vergleich**
  - zuweisungskompatible Records können auf ***Gleichheit*** und ***Ungleichheit*** geprüft werden.

```
TYPE Name1  = RECORD
                vorname  : TEXT;
                nachname : TEXT;
              END;
TYPE Name2  = RECORD
                nachname : TEXT;
                vorname  : TEXT;
              END;
VAR n1 : Name1; n2 : Name2;
n1 := n2  nicht zuweisungskompatibel
```


Zusammengesetzte benutzerdefinierte Datentypen

Die WITH-Anweisung

- **WITH-Anweisung**
  - dient dazu, komplexe Selektoren, die mehrmals verwendet werden müssen, mit einem ***ALIAS-Namen*** zu versehen.
  - Code wird ***kompakter***, lesbarer
- **Syntax:**

- **Semantik:**
  - der im Binding eingeführte Bezeichner ist im Block bis zum END gültig
  - der eingeführte Bezeichner wird als "***Abkürzung***" verwendet

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel WITH-Anweisung

```
TYPE Name   = RECORD                    TYPE Person = RECORD
                vorname  : TEXT;                anrede : Anrede;
                nachname : TEXT;                name   : Name;
              END;                              persnr : PersNr;
                                              END;

  VAR bundeskanzler : Person;

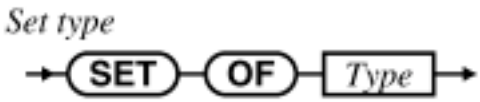
  bundeskanzler.anrede        := Anrede.Herr;
  bundeskanzler.name.vorname  := "Gerhard";
  bundeskanzler.name.nachname := "Schroeder";
  bundeskanzler.persnr        := 4711;
```

```
WITH bk = bundeskanzler DO
  bk.anrede := Anrede.Herr;
  WITH bkn = bk.name DO
    bkn.vorname := "Gerhard";
    bkn.nachname := "Schroeder";
  END;
  bk.persnr := 4700;
END
```


Zusammengesetzte benutzerdefinierte Datentypen

Mengen

- **Modula-3 bietet einen eigenen vordefinierten Mengentyp**
- **Syntax: (Typkonstruktor)**



- **Bemerkungen:**
  - Mengen sind ***ungeordnete*** Sammlungen von Elementen
  - der Elementtyp (Universum) muß ein ***Ordinaltyp*** sein!
  - Elemente einer Menge können **nicht** indiziert werden
  - Wertebereich eines Mengentyps ist die ***Potenzmenge***
    - Menge aller Teilmengen über dem Elementtyp
    - Bsp.: `ET = {rot, gruen}`

      Werte **`SET OF`** `ET: {}  {rot}  {gruen}  {rot, gruen}`
  - Aus Effizienzgründen sollen Mengen nur über Elementmengen mit ***kleiner Kardinalität*** gebildet werden.

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel : Mengendeklaration

```
TYPE Lottozahl = [1 .. 49];

TYPE Ziehung = SET OF Lottozahl;

CONST Leer := Ziehung {};

VAR   z1   := Ziehung {1 .. 7};
      z2   := Ziehung {4,7,34,20,44,23};
```

Mengenaggregat

- **Operationen:**
  - Zuweisung
    - zuweisungskompatibel: ***Elementtypen*** sind gleich
  - Vereinigung, Differenz, Durchschnitt, symmetrische Differenz
  - Gleichheit, Ungleichheit, Teilmenge, … , Enthalten

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Buchstabenzählen - 1

```
MODULE BuchstabenOrg EXPORTS Main;
IMPORT SIO, Text;

TYPE Buchstabe = ['A' .. 'Z'];
     BuchstabenMenge = SET OF Buchstabe;

CONST Alle = BuchstabenMenge {'A' .. 'Z'};
VAR einmal, mehrmals, nie := BuchstabenMenge {};
    eingabe : TEXT;
    z : CHAR;
BEGIN
  eingabe := SIO.GetLine();

  FOR i := 0 TO Text.Length(eingabe) - 1 DO
    z := Text.GetChar(eingabe, i);
    IF z IN Alle THEN
      IF z IN einmal THEN
        mehrmals := mehrmals + BuchstabenMenge{z};
      ELSE
        einmal := einmal + BuchstabenMenge{z};
      END;
    END;
  END;
  nie := Alle - einmal;
  einmal := einmal - mehrmals;
```

Set-Aggregat

Mengen-vereinigung

Mengen-differenz

Zusammengesetzte benutzerdefinierte Datentypen

Beispiel: Buchstabenzählen - 2

```
  SIO.PutLine("NIE:");
  FOR z := 'A' TO 'Z' DO
    IF z IN nie THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

  SIO.PutLine("EINMAL:");
  FOR z := 'A' TO 'Z' DO
    IF z IN einmal THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

  SIO.PutLine("MEHRMALS:");
  FOR z := 'A' TO 'Z' DO
    IF z IN mehrmals THEN
      SIO.PutChar(z)
    END;
  END;
  SIO.Nl();

END BuchstabenOrg.
```


Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 1 des Beispiels

```
PROCEDURE GebeMengeAus (m : BuchstabenMenge) =
BEGIN
  FOR z := FIRST(Buchstabe) TO LAST(Buchstabe) DO
    IF z IN m THEN
      SIO.PutChar(z)
    END;
  END;
END Ausgabe;

BEGIN (* Buchstaben1 *)
  eingabe := SIO.GetLine();
  FOR i := 0 TO Text.Length(eingabe) - 1 DO
    z := Text.GetChar(eingabe, i);
    IF z IN Alle THEN
      IF z IN einmal THEN
        mehrmals := mehrmals + BuchstabenMenge{z};
      ELSE
        einmal := einmal + BuchstabenMenge{z};
      END;
    END;
  END;
  nie := Alle - einmal;
  einmal := einmal - mehrmals;
  SIO.PutLine("NIE:");      GebeMengeAus(nie); SIO.Nl();
  SIO.PutLine("EINMAL:");   GebeMengeAus(einmal); SIO.Nl();
  SIO.PutLine("MEHRMALS:"); GebeMengeAus(mehrmals); SIO.Nl();
END Buchstaben1.
```

Verwenden einer Prozedur für die Ausgabe von Mengen

Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 2 des Beispiels - a

```
TYPE Vorkommen = RECORD
                   einmal, mehrmals, nie := BuchstabenMenge {};
                 END;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
  CONST Alle     := BuchstabenMenge {'A' .. 'Z'};
  VAR   resultat : Vorkommen; z : CHAR; vorgekommen : BuchstabenMenge;
BEGIN
  WITH r = resultat DO
    FOR i := 0 TO Text.Length(t) - 1 DO
      z := Text.GetChar (t, i);
      IF z IN Alle THEN
        IF z IN vorgekommen THEN
          r.mehrmals := r.mehrmals + BuchstabenMenge{z};
        ELSE
          vorgekommen := vorgekommen + BuchstabenMenge{z};
        END;
      END;
    END;
    r.nie := Alle - vorgekommen ;
    r.einmal := vorgekommen - r.mehrmals;
  END;
  RETURN resultat;
END Vorkommenzaehlen;
```

Wäre ein Array eine Alternative?

Bezeichner werden nur für einen Zweck eingesetzt!

Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 2 des Beispiels - b

```
MODULE Buchstaben1 EXPORTS Main;
…
VAR vork : Vorkommen;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
...


BEGIN

  vork := Vorkommenzaehlen(SIO.GetLine());

  SIO.PutLine("NIE:");       GebeMengeAus(vork.nie); SIO.Nl();
  SIO.PutLine("EINMAL:");    GebeMengeAus(vork.einmal); SIO.Nl();
  SIO.PutLine("MEHRMALS:"); GebeMengeAus(vork.mehrmals); SIO.Nl();

END Buchstaben1.
```

Verwenden die Ausgabeoperation für Mengen

Zusammengesetzte benutzerdefinierte Datentypen

Verbesserung 2 des Beispiels - c

```
PROCEDURE Ausgabe (v: Vorkommen) =
CONST NIE      = "     NIE : ";
      EINMAL   = "  EINMAL : ";
      MEHRMALS = "MEHRMALS : ";
BEGIN
  SIO.PutText(NIE);      GebeMengeAus(v.nie); SIO.Nl();
  SIO.PutText(EINMAL);   GebeMengeAus(v.einmal); SIO.Nl();
  SIO.PutText(MEHRMALS); GebeMengeAus(v.mehrmals); SIO.Nl();
END Ausgabe;

PROCEDURE Vorkommenzaehlen (t : TEXT) : Vorkommen =
...

BEGIN (*Buchstaben2 *)

  Ausgabe(Vorkommenzaehlen(SIO.GetLine()));

END Buchstaben2.
```


Zusammengesetzte benutzerdefinierte Datentypen

Arrays, Records, Mengen

- **Vergleich der Typkonstruktoren für**
  - statische, zusammengesetzte Typen

Aspekt	Array	Record	Menge
Größe	fest (!)	fest	fest
Element-typen	homogen	heterogen	homogen, Ordinaltyp
Zugriff	dynamisch indiziert	statisch	kein direkter Zugriff
Ordnung	statisch festgelegt Index ist geordnet	statisch festgelegt	ungeordnet

Was haben wir gelernt!

- Datentypen: Zweck, Typisierung, Deklaration, Einteilung
- benutzerdefinierte Datentypen mittels Datentypkonstruktoren
- Aufzählungs- und Unterbereichstypen als benutzerdefinierte skalare Typen, Aufzählungsliterale
- Feldtypen: Indextyp, Elementtyp, eindimensionale, mehrdimensionale Feldverarbeitung: mittels Zählschleifen, Feldattributen, Feldindizierung, Feldaggregate, Feldinitialisierung, Feldzuweisung und -vergleich
- Verbundtypen: Komponententypen, Komponentennamen, Selektor(pfad)
- Verbundverarbeitung: Komponentenzugriff, Verbundaggregate, Verbundzuweisung und -vergleich, with-Anweisung
- Mengentypen: Elementtyp (Trägermenge), Teilmengenbildung, charakteristische Speicherung
- Mengenverarbeitung: Mengenaggregate, Vereinigung, Durchschnitt, Teilmenge, Enthalten, etc.
- Vergleich Datentypkonstruktoren für zusammengesetzte Datentypen

Glossar

- Datentypenklassifikation: vor- oder selbstdefiniert, skalar oder zusammengesetzt, statisch oder dynamisch
- Typ: Charakterisierung
- Aufzählungstypen, Unterbereichstypen
- Feldtypen, Verbundtypen, Mengentypen
- Typdefinitionen, Typdeklarationen, Objektdeklarationen mittels Typ, ggfs. mit Initialisierung, bequem durch Aggregate
- Datentypkonstruktoren für Felder, Verbunde, Mengen
- Aggregate für Felder, Verbunde, Mengen

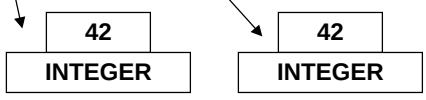
Datentypen II

- **Dynamische Datentypen**
  - Zeigertypen
  - dynamische Datenstrukturen
- **Anwendungsbeispiele**
  - lineare Liste
  - sortierte Liste
- **Prozedurtyp**

Wertsemantik

- **Die bisher betrachteten Variablen und die Zuweisung basieren auf der *Wertsemantik*:**
  - Eine Variable hat einen ***definierten*** oder ***undefinierten*** Wert.
  - Bei der ***Zuweisung*** wird der Wert des Ausdrucks der rechten Seite (rhv) an die Variable der linken Seite (lhv) zugewiesen.
  - Eine ***Identität*** von Objekten wird nicht hergestellt.
  -
```
VAR Antwort1, Antwort2: INTEGER;
…
Antwort1 := 42; (* 1 *)
Antwort2 := Antwort1; (* 2 *)
Antwort1 := 24; (* 3 *)

Antwort1 := Antwort2;
```

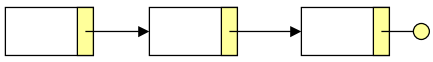


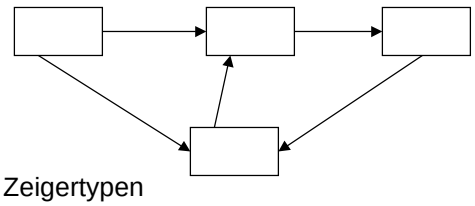
Statische Datentypen

- **Kennzeichnend für imperative Sprachen ist, daß *Wertsemantik* und *statische Datentypen* zusammenfallen:**
  - Die Zuordnung von Bezeichner und Wert geschieht über die Zuweisung.
  - Die Variablen eines statischen Typs ***behalten ihre Struktur*** während ihrer Lebensdauer bei.
  - Der ***Speicher*** einer statisch getypten Variablen wird bei der Übersetzung anhand der Deklaration ***festgelegt*** und bei der "ersten Verwendung implizit angelegt ".
- **Speicherbereiche für statisch getypte Variablen werden im sog. *Kellerspeicher* reserviert**
  - zusammenhängender Bereich pro Objekt
  - Bereich wird ***angelegt***, beim Eintritt in entsprechende Prozedur
  - Bereich wird ***freigegeben***, beim Verlassen der Prozedur

Statische Datentypen und Listen

- **Probleme mit statischen Datentypen**
  - Es werden Datenstrukturen benötigt, deren ***Größen Schwankungen*** unterworfen sind.
  - Es sollen ***komplizierte*** Datenstrukturen gebildet werden
    - Listen
    - Bäume
    - Graphen





- **Lösung**
  - Dynamische Datentypen
  - oder dynamische Variable und Zeigertypen

Zeigertyp - Referenztyp

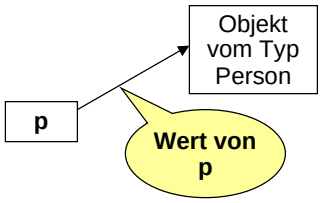
Zeigertyp

- um einen Zeigertyp zu deklarieren, bietet Modula-3 den ***Typkonstruktor***
- **<ZeigerTyp> = REF <ReferenzierterTyp>** an
- damit kann aus ***jedem Typ*** ein Zeigertyp abgeleitet werden

```
TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : PersNr;
              END;

TYPE PersonRef = REF Person;

VAR p : PersonRef;
```



- p kann als Werte ***Zeiger auf Objekte*** vom Typ `Person` annehmen
- zeigt p auf kein Objekt, dann wird dies durch den Wert ***NIL*** angezeigt
- `NIL` ist Objekt jedes Zeigertyps

Eigenschaften dynamischer Datentypen

Eigenschaften von Objekten dynamischer Datentypen

- ***Lebensdauer*** ist nicht an die Ausführung einer Prozedur oder Moduls gebunden
- sie werden zur Laufzeit ***explizit (vom Programmierer) erzeugt*** und eventuell auch wieder beseitigt
- sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
- können in prinzipiell ***beliebiger*** Menge geschaffen werden
- haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
- sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
- Zeiger können im Keller oder auf der Halde liegen
- die referenzierten Objekte liegen ***immer*** auf der Halde

Dynamische Datentypen

Erzeugen von Haldenobjekten

- Beispiel:

```
PROCEDURE PROC …
  TYPE PersonRef = REF Person;
  VAR p : PersonRef;
BEGIN
  p := NIL;
  p := NEW (PersonRef);
END
```

- Beim Betreten der Prozedur wird Speicher für p zur Verfügung gestellt und beim Verlassen wieder freigegeben.
- Durch den Aufruf von `NEW(PersonRef)`
  - wird auf der Halde ***Speicher*** für ein Objekt von Typ Person angelegt
  - die ***Adresse*** dieses Speicherplatzes wird zurückgeliefert und der Variablen p zugewiesen
  - Das Objekt selbst erhält keinen eigenen Bezeichner – man spricht von einer ***dynamischen*** oder auch von einer ***anonymen*** Variablen.
- Dieser Speicher wird nicht freigegeben, wenn die Proz. verlassen wird

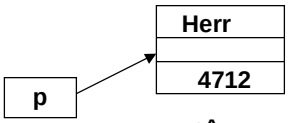
Dynamische Datentypen

Dereferenzierung - 1

- **Eine gesetzte Referenzvariable verweist auf ein Objekt.**
- **Um das Objekt selbst zu erhalten, müssen wir dem Verweis "nachgehen".**
  - Diese Operation, bei der auf das referenzierte Objekt einer Referenzvariablen zugegriffen wird, heißt ***dereferenzieren***.
- **Dereferenzieren ist, neben dem Erzeugen der referenzierten Objekte, die**
  - zweite charakteristische Operation von Referenztypen.
  - Nur eine ***gesetzte Referenzvariable*** kann dereferenziert werden.
  - Der Versuch einer Dereferenzierung der Referenz `NIL` führt in den meisten Programmiersprachen zum ***Programmabbruch***.

Dereferenzierung - 2

- **Dereferenzierung**
  - Zugriff auf ***Werte*** von Zeigerobjekten




```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p^.persnr := 4732;

p := NEW (PersonRef);
p^.persnr := 4712;
p^.anrede := AnredeTyp.Herr;

nr := p^.persnr;
```

Laufzeitfehler: Zeigervariable nicht gesetzt

Dereferenzierungsoperator

  - In Modula-3 kann der ^-Operator entfallen, wenn ein weiterer Operator folgt (z.B."[ ]", ".")
  - Das trägt ***nicht zur Klarheit*** bei !!!
  - Darum: Verwenden Sie ***immer*** den ^-Operator !!!

Operationen auf Referenztypen

- **Zulässige Werte von Referenztypen**
  - sind Referenzen oder der Wert "***keine Referenz***" (`NIL`).
- **Gesetzte Referenzen haben keine externe Repräsentation.**
  - Entsprechend können sie ***nicht*** ausgegeben oder z.B. in ***Rechenoperationen*** verwendet werden.
- **Die einzigen zulässigen Operationen auf Referenzen sind:**
  - Test auf ***Gleichheit*** oder ***Ungleichheit***,
  - ***Zuweisung*** auf Variablen von kompatiblem Typ.
- **Beispiel in Modula-3:**

```
VAR p1, p2 : PersonRef;
…
IF p1 = NIL THEN
  p1 := NEW (PersonRef)
END;
p2 := p1;
```

Dynamische Datentypen

Zuweisung

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
p := NEW(PersonRef);
p^.name.nachname := "Maier";
q := NEW(PersonRef);
q^.name.nachname := "Mueller";

p := q;      (1)

p^ := q^;    (2)
```

- Der Effekt der Anweisung (1) ist streng zu unterscheiden von der Wirkung der Anweisung (2)

  (1) ist eine ***Referenzzuweisung***

  (2) ist eine ***Wertzuweisung***

Dynamische Datentypen

Vergleich

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;

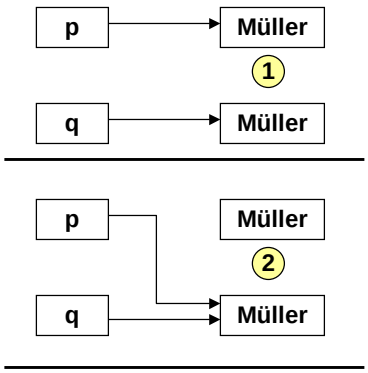
... p = q ...          (1)

... p^ = q^ ...



... p = q ...          (2)

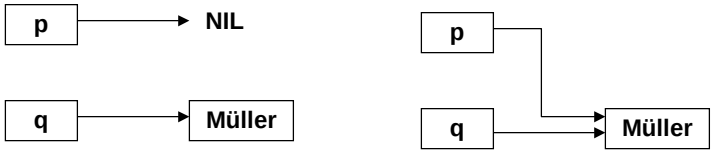
... p^ = q^ ...
```



- Zwei Zeiger sind gleich, wenn sie auf ***dasselbe*** referenzierte Objekt (oder: auf die gleiche Adresse) zeigen!

Das Alias-Problem

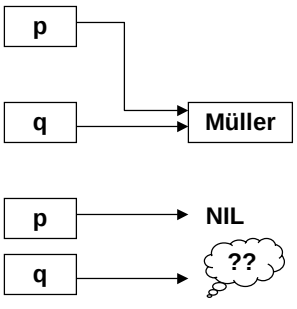
- **Die Zuweisung bei Referenztypen basiert auf der sog. *Referenzsemantik*:**
  - Der ***Verweis*** auf ein Objekt wird zugewiesen; nicht etwa der Wert des referenzierten Objekts.
  - Dies ist vielfach erwünscht, schafft aber folgendes Problem
- **Mehrere Referenzvariablen können auf dasselbe Objekt verweisen.**
  - Damit ist lokal oft nicht ***entscheidbar***, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben oder nicht.
  - Dies ist das ***Alias-Problem***, das bei allen Referenztypen auftritt.



Freigeben von Zeigerobjekten

- **In der Regel müssen Zeigerobjekte vom Programmierer kontrolliert werden:**
  - Er muß sie explizit ***anlegen*** und ***freigeben***
- **Beim Freigeben können folgende Fehlersituationen auftreten**
  - Nicht mehr benötigte Zeigerobjekte wurden ***nicht frei*** gegeben
    - Es wird Speicher verschwendet
  - Es werden Zeigerobjekte freigegeben, die noch ***benötigt*** werden
    - Problem der "***dangling references***"

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
...
p := q;
DISPOSE(p);
q^.anrede := ...
```

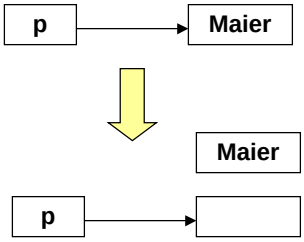


Probleme mit Referenzvariablen

- **"Lost Object":**
  - Es kann dynamisch angelegte Objekte geben, auf die keine ***Referenzvariable*** mehr verweist.
  - Dieses Objekt kann nicht mehr erreicht werden und wird als ***Garbage*** bezeichnet.
- **Häufige Fehlersituation:**

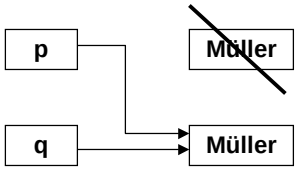
```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p := NEW (PersonRef);
p^.name := "Maier";
...
p := NEW (PersonRef);
```



Automatische Speicherbereinigung

- **Einige Laufzeitumgebungen von Sprachen können Zeigerobjekte kontrollieren**
  - sie geben Zeigerobjekte, die ***nicht mehr erreicht*** werden können, automatisch frei
  - "***Garbage Collector***" (Müllsammler)

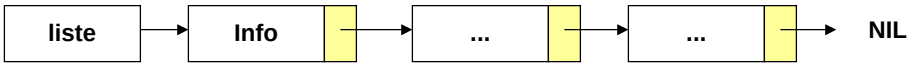


- **Diskussion Garbage Collector**
  - ***Vorteile*** (wenn keine explizite Freigabe vorhanden oder genutzt)
    - Fehlerklasse "dangling reference" ausgeschaltet
  - ***Nachteile***
    - Müllsammeln kostet Zeit

Anwendungs-beispiele

Dynamische Datenstrukturen

- **Ziel:**
  - Definition dynamischer Datenstrukturen, die ***einfach*** vom Programmierer erstellt, verwaltet und kontrolliert werden können
- **Referenzvariablen**
  - können auf zusammengesetzte Datenstrukturen verweisen, die selbst wieder ***Referenzobjekte*** enthalten. Wenn diese Strukturen rekursiv sind, sprechen wir von ***Verweisketten***, die den Aufbau dynamischer Datenstrukturen ermöglichen.
- **Beispiel: Einfach verkettete lineare Liste**
  - Elemente der Liste sind ***Zeigerobjekte***
  - Jedes Listenobjekt ist so konstruiert, das es ***selbst*** wieder auf ein Listenobjekt ***verweisen*** kann
  - zusätzlich enthält es die ***eigentlichen*** Informationen



Anwendungs-beispiele

Lineare Liste

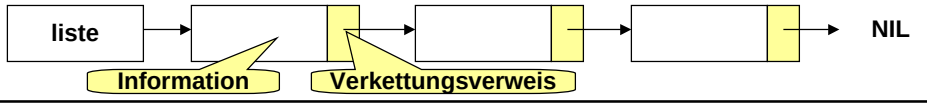
```
TYPE Jahr = [1890 .. 1998];

TYPE Person = RECORD
  name, vorname : TEXT;
  geburtsjahr : Jahr;
END;

TYPE ListenElement =
  RECORD
    person : Person;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : RListenElement;
```

Typ, für die in der Liste verwaltete Information

"Anker", um auf die Liste zugreifen zu können



Anwendungs-beispiele

Suchen in linearen Listen - 1

```
PROCEDURE Suche (liste : RListenElement;
                 was : Person): RListenElement =
VAR position : RListenElement;
    gefunden : BOOLEAN := FALSE;

BEGIN
  position := liste;
  WHILE (position # NIL AND NOT gefunden) DO
    IF was = position^.person THEN
      gefunden := TRUE;
    ELSE
      position := position^.nachfolger;
    END;
  END;
  RETURN position;
END Suche;
```

Sequentielles Suchen

Ergebnis : Zeiger auf das gefundene Element sonst NIL

Anwendungs-beispiele

Suchen in linearen Listen - 2

- **Listen sind rekursive Datenstrukturen**
  - ***Rekursive Datenstrukturen*** können *elegant* mit ***rekursiven Prozeduren*** bearbeitet werden!

```
PROCEDURE SucheRek (liste : RListenElement;
                    was : Person): RListenElement =
BEGIN
  IF liste # NIL THEN
    IF was = liste^.person THEN
       RETURN liste;
    ELSE
      RETURN SucheRek (liste^.nachfolger, was);
    END;
  ELSE
    RETURN NIL;
  END;
END SucheRek;
```

rekursiver Aufruf

Anwendungs-beispiele

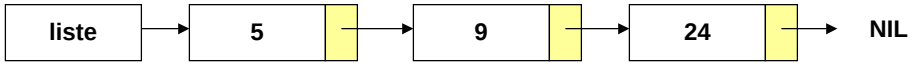
Sortierte lineare Liste

```
TYPE ListenElement =
  RECORD
    wert : INTEGER;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : ListenElement;
```

Für alle Elemente x der Liste gilt:

`x^.wert <= x^.nachfolger^.wert`

Liste soll immer im Zustand "aufsteigend" sortiert sein

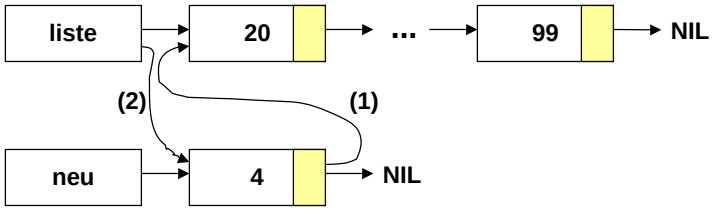


Anwendungs-beispiele

Einfügen in eine sortierte Liste - 1

- **Fall 1: Die Liste ist leer.**

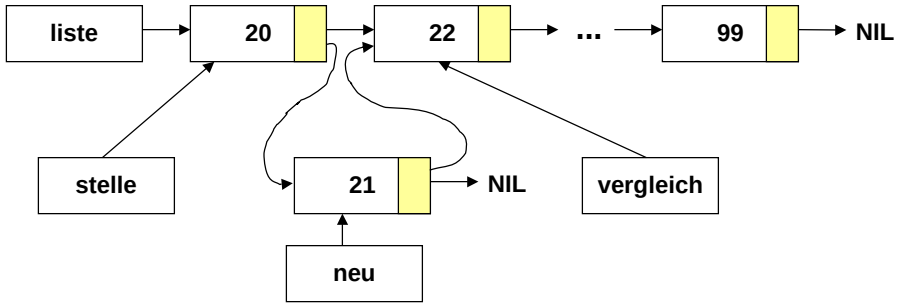
- **Fall 2: Element muß vorne eingefügt werden**



Anwendungs-beispiele

Einfügen in eine sortierte Liste - 2

- **Fall 3: Element muß sonst irgendwo eingefügt werden.**
  - Es muß nach der ***Einfügestelle*** gesucht werden
  - Damit man auf die Elemente vor und hinter der Einfügestelle zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das Element hinten angefügt werden muß !!

Anwendungs-beispiele

Einfügen: Lösung A -1

```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, einfuegestelle : RListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                  (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN         (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);        (* neu vorne einhaengen *)
  ELSE
    einfuegestelle := SucheEinfuegestelle(liste, w);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Anwendungs-beispiele

Einfügen: Lösung A -2

```
PROCEDURE ErzeugeNeuesListenelement(w : INTEGER): RListenElement =
VAR elem : RListenElement;
BEGIN
  elem := NEW(RListenElement);
  elem^.wert := w;
  elem^.nachfolger := NIL;
  RETURN elem
END ErzeugeNeuesListenelement;
```

```
PROCEDURE EinfuegenVorne(VAR liste : RListenElement;
                         VAR neu   : RListenElement)=
BEGIN
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegenVorne;
```


Anwendungs-beispiele

Einfügen: Lösung A -3

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                              w : INTEGER): RListenElement =
VAR stelle, vergleich: ListenElement;
BEGIN
  vergleich := liste;
  stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

```
PROCEDURE FuegeAnEinfuegestelleEin(VAR neu   : RListenElement;
                                   VAR stelle: RListenElement)=
BEGIN
  neu^.nachfolger := stelle^.nachfolger;
  stelle^.nachfolger := neu;
END FuegeAnEinfuegestelleEin;
```

Anwendungs-beispiele

Einfügen: Lösung B

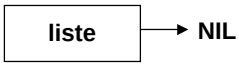
```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, position, vorgaenger : RListenElement;
BEGIN
  neu := NEW(RListenElement);
  neu^.wert := w;
  neu^.nachfolger := NIL;

  IF liste = NIL THEN             (* liste ist leer *)
    liste := neu;
  ELSIF (w <= liste^.wert) THEN   (* w ist kleinstes Element*)
    neu^.nachfolger := liste;     (* w vorne einhaengen *)
    liste := neu;
  ELSE                            (* Einfuegestelle suchen *)
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert <= w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* vorgaenger = Einfuegestelle *)
    neu^.nachfolger := position;  (* w einhaengen *)
    vorgaenger^.nachfolger := neu;
  END;
END Einfuegen;
```

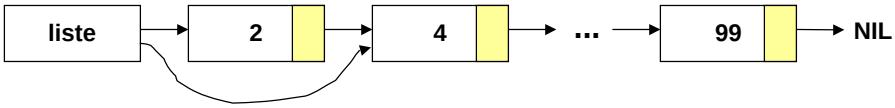

Anwendungs-beispiele

Löschen in einer sortierten Liste - 1

- **Fall 1: Die Liste ist leer.**



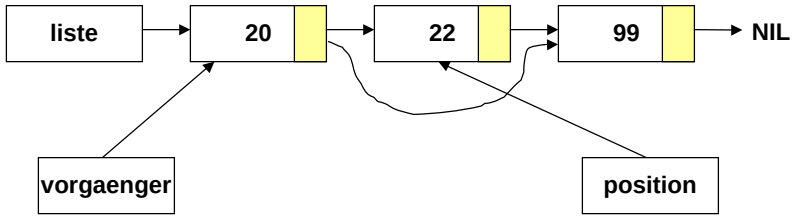
- **Fall 2: Das erste Element muß gelöscht werden**



Anwendungs-beispiele

Löschen in einer sortierten Liste - 2

- **Fall 3: Element muß sonst irgendwo gelöscht werden.**
  - Damit man auf die Elemente vor und hinter dem zu löschenden Element zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das letzte Element gelöscht werden muß !!

Anwendungs-beispiele

Löschen: Lösung A

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert # w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* position = NIL v position^.wert = w *)
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        liste := position^.nachfolger;
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Anwendungs-beispiele

Löschen: Lösung B -1

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position);
    gefunden := (position # NIL);  (* position= NIL v position^.wert = w *)
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```


Anwendungs-beispiele

Löschen: Lösung B -2

```
PROCEDURE Suche (liste : RListenElement;
                 w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```

Anwendungs-beispiele

Beobachtung !

- **Einfügen und Loeschen benutzen ähnliche Such-Prozeduren**

```
PROCEDURE Suche (liste : RListenElement; w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste; vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                                  w : INTEGER): RListenElement =
VAR stelle, vergleich: RListenElement;
BEGIN
  vergleich := liste; stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```


Prozedurtyp

Prozedurtyp und Prozeduren dieses Typs

- **Frage:**
  - Gibt es eine Möglichkeit, Prozeduren so zu ***parametrisieren***, daß als Parameter ein Algorithmus (Prozedur) übergeben werden kann?
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur.***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.
  - Prozedurvariablen können ***passende*** Prozeduren zugewiesen werden.
  - Prozedurvariablen können als ***Parameter*** übergeben werden.
  - Gesetzte Prozedurvariablen können in Anweisungen mit ***aktuellen Parametern*** aufgerufen werden.

Prozedurtyp

Beispiel: ProzedurTyp - 1

```
TYPE  Bereich      = [-10 .. 10]; Index   = [1 .. 10];
      Zahlenmenge  = SET OF Bereich;
      Zahlenfeld   = ARRAY Index OF Bereich;
      TestProzedur = PROCEDURE (a : INTEGER) : BOOLEAN;

PROCEDURE IstPositiv (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i > 0);
END IstPositiv;

PROCEDURE IstNegativ (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i < 0);
END IstNegativ;

PROCEDURE Filtern (        feld : FeldTyp;
                   VAR resultat : Zahlenmenge;  p : TestProzedur) =
BEGIN
  FOR i := FIRST(Index) TO LAST(Index) DO
    IF p(feld[i]) THEN
      resultat := resultat + Zahlenmenge{feld[i]};
    END;
  END;
END Filtern;
```

Gegeben ist ein Feld mit Zahlen. Bestimme die darin enthaltenen positiven und negativen Zahlen

Signatur-konforme Prozeduren

Prozedur-parameter

Prozedurtyp

Beispiel: ProzedurTyp - 2

```
VAR proc : TestProzedur;

    werte := Zahlenfeld{3, -5, 9, 8, -6, 9, -6, -1, -1, 5};
    positiv, negativ := Zahlenmenge{};

BEGIN
  proc := IstPositiv;
  Filtern (werte, positiv, proc);
  proc := IstNegativ;
  Filtern (werte, negativ, proc);

  SIO.PutLine ("Positive Zahlen:");
  FOR e := FIRST(Bereich) TO LAST(Bereich) DO
    IF e IN positiv THEN SIO.PutInt(e); SIO.PutText(", "); END;
  END;
  SIO.Nl();
```

Prozedurvariable

Zuweisung einer Prozedur an die Prozedurvariable

Zuweisung an Prozedurvariable

- **Zuweisung**
  - TYPE PType = PROCEDURE …
    VAR proc : Ptype

    proc := PT; (* Ausdruck der vom Typ PT ist *)
  - Diese Zuweisung ist korrekt, wenn
    - PT = ***NIL*** ist (NIL ist mit jedem Wert eines Prozedurtyps kompatibel)
    - ***Anzahl*** der Parameter und deren ***Typen*** sind gleich für PT und PType
    - Beide haben den gleichen ***Ergebnistyp*** oder keinen

```
TYPE TestProzedur = PROCEDURE (a : INTEGER)  : BOOLEAN;
     PT1          = PROCEDURE (in : INTEGER) : BOOLEAN;

VAR test : TestProzedur;
      p1 : PT1;
…
p1 := test;
test := p1;
```

Typen sind signatur-konform

Prozedurtyp

Verwendung Prozedurtyp - 1

```
TYPE Vergleichsoperation = PROCEDURE (a, b : INTEGER) : BOOLEAN;

PROCEDURE Suche (liste : ListenElement;
                     w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement;
                    op : Vergleichsoperation) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND  NOT op(stelle^.wert, w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

signaturkonform

```
PROCEDURE IstGleich
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a = b;
END IstGleich;
```

```
PROCEDURE Groesser
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a > b;
END Groesser;
```

Prozedurtyp

Verwendung Prozedurtyp - 2

```
PROCEDURE Loeschen (VAR liste : ListenElement; w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position, IstGleich);
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```


Prozedurtyp

Verwendung Prozedurtyp - 3

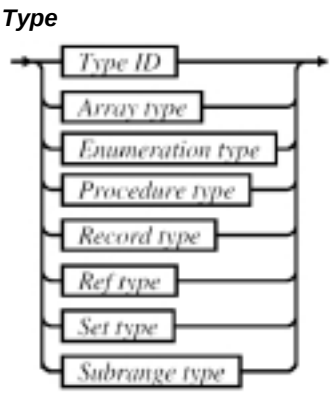
```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, einfuegestelle, groessererWert : ListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN       (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);      (* neu vorne einhaengen *)
  ELSE
    Suche(liste, w, einfuegestelle, groessererWert , Groesser);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Diskussion: Dynamische Datentypen

- **Je deutlicher die Realisierung dynamischer Datenstrukturen durch Zeiger in der Sprache sichtbar ist,**
  - desto leichter fällt die softwaretechnisch ***unsaubere*** Verwendung.
- **Die explizite Speicherverwaltung führt zu einigen Problemen.**
  - Moderne imperative Sprachen sollten daher auf explizites Löschen verzichten und hierfür einen ***Garbage Collector*** besitzen.
- **Erst in objektorientierten Sprachen,**
  - die vorrangig mit ***Referenzsemantik*** arbeiten,
  - können grundlegende Probleme der Referenzierung gelöst werden.
- **Prozedurtypen sind nicht dynamisch, aber**
  - Prozedurvariablen können dynamisch unterschiedlichen Prozedurobjekten zugewiesen werden

Zusammenfassung: Datentypen

- **Datentypen**
  - definieren ***Werte*** und zulässige ***Operationen***
  - helfen, ***Abstraktionen*** zu formulieren
  - repräsentieren das ***Vokabular*** eines Systems
    - Person, Liste etc.
  - dienen der ***Sicherheit*** der Programme
    - jedes Objekt hat einen Typ
    - Typ-Prüfung bei Parameterübergabe und Zuweisung
  - müssen wohlüberlegt ***entworfen*** werden
    - manifestieren sich im Programm
    - sind dann nur ***schwer veränderbar***



Was haben wir gelernt?

- Charakterisierung statischer Datentypen (Wiederholung) und Defizit für Listenverarbeitung
- Typen für Listenelemente, Zeigertypen, dynamische Objekte (Haldenobjekte)
- dynamische Datenobjekte: Explizite Erzeugung, implizite Bezeichnung, Lebensdauer, Halde (Heap), Freigabe
- Derefenzieren, Lesen und Setzen von Haldenobjekten, Setzen von Zeigerobjekten, Vergleich
- Aliasing, nicht mehr greifbare Haldenobjekte, hängende Zeiger, Garbage Collection
- Lineare Liste: Einfügen/Löschen (vorn und in der Mitte), Suchen, geordnete lineare Liste
- Arten von Zeigern (Zweck ihrer Verwendung)
- Prozedurobjekt, Prozedurtyp, Prozedurvariable, Einsatz für Flexibilität

Glossar

- Wertsemantik, Verweis (Zeiger) semantik
- statische Datentypen, dynamische Datentypen, rekursive Datentypen
- Zeigertyp, Zeigerobjekt (-konstante oder -variable)
- Haldenobjekttyp, dynamische (anonyme) Datenobjekte: Haldenobjekte, Erzeugung mit Einrichtung eines Zeigerwerts, Löschen mit Löschen des Zeigerwerts
- statische Datenobjekte, Datenobjekte auf dem Kellerspeicher, Datenobjekte auf der Halde
- Zeiger: Dereferenzieren, Setzen, Lesen, Vergleichen, Setzen/Lesen von Komponenten von Haldenobjekten
- Aliasing, inaccessible objects, dangling references
- Ankerzeiger, Verkettungszeiger, Durchlaufzeiger, „semantische“ Zeiger, Abkürzung von Zugriffswegen
- einfach verkettete lineare Liste, sortierte Liste, doppelt verkettete Liste
- Listenoperationen: Einfügen vorn, in der Mitte, Suchen, Löschen vorn, in der Mitte, Suche als allgemeine Prozedur für Listenoperationen
- Prozedurtyp und Parameterprofil, signaturkonforme Prozedurobjekte, Prozedurvariable und Prozedurobjektzuweisung, Lebensdauer von Zeigerobjekten, von über Zeiger zugegriffenen Haldenobjekten

Zusammenfassung
Programmiersprachen-Konstrukte

- **Ablaufkontrolle**
- **Datenstrukturierung**
- **Entsprechung Kontroll- und Datenstrukturen**
- **Typäquivalenz**

Ablauf-kontrolle **Kontrollstrukturen zur Bildung zusammengesetzer Anweisungen**

- **Formulierung der Berechnung, des Berechnungsablauf**
- **Bildung zusammengesetzter Anweisungen mit Konstruktionselementen (Kontrollstrukturen)**
  - Aneinanderreihung von Anweisungen
  - Fallunterscheidung: bedingte Anweisung **IF**, Auswahlanweisung **CASE**
  - Iteration: Zählschleife **FOR**, bedingte Schleifen **WHILE, UNTIL**
  - kontrollierter Sprung **EXIT**
  - unkontrollierter Sprung **GOTO**

- **Rekursiver Aufbau:**
  - einfache Anweisung —**Kontroll-strukturen**→ zusammengesetzte Anweisungen (**Kontroll-strukturen**)
  - dynamischer Ablauf ist Programmpfad des statischen Berechnungsablaufs

Daten-strukturierung

Datentypkonstruktoren

- **Formulierung des Datenaufbaus durch Datentypdeklarationen**
- **Zusammengesetzte / skalare Datentypen; vordefinierte / selbstdefinierte; statische / dynamische**
- **Datentypen**

zusammengesetzte:		*einfache*:
Felder	statisch	vordefinierte ganzzahlige, reelle
Verbunde		selbstdef. Aufzählungsdatentypen
Mengen (eingeschränkt)		Unterbereichstypen

- **Rekursiver Aufbau:**
  - skalare Datentypen —**Datentyp-konstruktor**→ zusammengesetzte Datentypen (**Datentyp-konstruktor**)
- **Aufbau eines Datentyps:**
  - fest bei statischen Typen, sinnvolle Werte hängen vom Berechnungsablauf ab
  - bei dynamischen Datentypen ergibt sich ein sinnvoller Aufbau durch Berechnung

H. Lichter / M. Nagl, 2000 Teil II. Zusammenfassung. - 3 -

Datentypen und Kontrollstrukturen

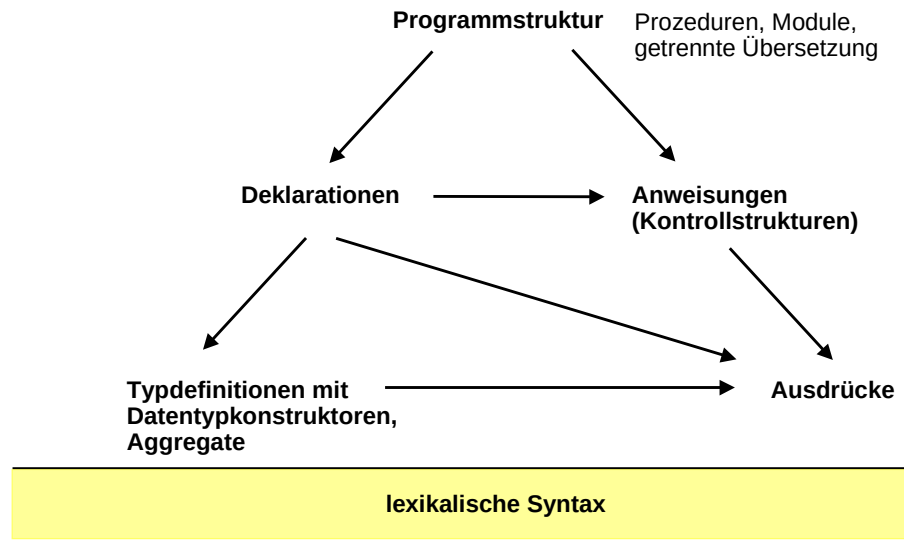
Entsprechung der Konstruktionselemente

Konstruktionsmuster	Anweisungsform	Datentyp
Atomares Element	Zuweisung	skalarer Datentyp
Zusammenfassung unterschiedlicher Elemente	Anweisungsfolge	Verbundtyp
Wiederholung/ Zusammenfassung gleicher Elemente, Anzahl bekannt	Zählschleife	Feldtyp
Auswahl	Bedingte Anweisung IF Auswahlanweisung CASE	(varianter Verbund, Vereinigungstyp)
Wiederholung, Anzahl unbekannt	Bedingte Schleifen WHILE, UNTIL	Sequenz (Datei)
Rekursion	rekursive Prozeduren	rekursive Datentypen
Allgemeiner Graph	Sprunganweisung	verkettete Strukturen auf Halde mit Zeigern

H. Lichter / M. Nagl, 2000 Teil II. Zusammenfassung. - 4 -

Gruppierung der Konstruktions-elemente

Gruppierung der Sprachkonstrukte (s. kontextfreie Syntax)



Typäquivalenz

Strukturäquivalenz

- **Zwei Typen sind gleich, wenn sie nach Expansion der Typdefinitionen gleich sind.**
- **Skalare Datentypen**
  - Datentypdefinition ist gleich
  - Name eines vordefinierten Typs oder gleiche Aufzählungs- bzw. Unterbereichsdefinition.
- **Zusammengesetzte Datentypen**
  - Felder: gleicher Komponententyp, gleiche Indextypen in gleicher Reihenfolge, gleiche Anzahl von Elementen für jeden Indextyp
  - Verbunde: gleiche Komponententypen, in gleicher Reihenfolge, gleiche Selektornamen
  - Mengen: gleicher Typ für Universum
- **Prozedurtypen**
  - gleiche Parameterzahl und Parametertypen, in gleicher Reihenfolge, gleicher Bindungsmodus, gleicher Ergebnistyp

Beispiel für systematische Programmentwicklung

- **Entwickeln**
- **Verbessern**
- **Effizienzbetrachtung**

Lösungsentwicklung

- **Bisher**
  - Problemformulierung (als Übungsaufgabe)
  - Lösung wird direkt Programm erstellt
    - **Hilfsmittel**: schrittweise Verfeinerung
      Wahl geeigneter Bezeichner
      Verwendung von Kommentaren
- **Realität**
  - Problemformulierung, Festlegung der Bedieneroberfläche etc. ist **wichtiger Teil** der Softwareerstellung
  - Softwareerstellung, -veränderung geschieht **in „Phasen“**
  - **viele andere Dokumente** ausser Quelltext sind nötig
  - Programm wird immer aus **Bausteinen** zusammengesetzt
  - bei Softwareerstellung wirken **viele Personen** mit
- **Softwaretechnik liefert geeignete Methoden und Techniken.**

Welche Lösung?

- **Problem → zugehörige Software (Programm)**
  es gibt (unendlich) viele Lösungen
- **Welche Lösung ist geeignet?**
- **Welche Eigenschaften soll die Lösung haben?**
  - Diese müssen explizit formuliert werden (Anforderungen)
  - Eigenschaften sind **Qualitätseigenschaften**
  - Suchen insbesondere ***effiziente(ste) Lösung***
- **Programme sind im allgemeinen falsch**
  - Große Programmsysteme sind immer falsch
  - Welche Maßnahmen sind notwendig, um Korrektheit zu „erzielen“?

Entwickeln

Präzisierung der Aufgabe

- **Aufgabenformulierung:**

  In einer Datei befindet sich eine unbekannte Anzahl von Zahlen. Diese soll gelesen und nach ihrer Größe ausgedruckt werden.
- **Unklarheiten**
  - Format: Wie ausdrucken?
  - Mehrfachvorkommnisse: erlaubt, mehrfach aufgeführt?
  - Sortierung: aufsteigend, absteigend?
  - Sortierung in einem Feld: max. Anzahl unbekannt?
    (internes Sortieren/externers Sortieren)
  - Welcher Typ der einzelnen Elemente: INTEGER, REAL?
  - Feste Randbedingungen: Zahlen < größte in der Basismaschine darstellbare Zahl
- **Neue Formulierung:**

  In einer Datei befinden sich max. 100 nicht notwendigerweise verschiedene INTEGER-Zahlen. Diese Zahlen sollen gelesen und der Größe nach aufsteigend sortiert werden und entsprechend ihrer Vorkommenshäufigkeit einzelnen ausgegeben werden und zwar je 10 pro Zeile.

Entwickeln

Erster Schritt

- **Schrittweise Verfeinerung**

```
MODULE Main;

CONST Max        = 100;
TYPE  Index      = [1..MAX];
      Zahlenfeld = ARRAY Index OF INTEGER;
VAR   anzahl     : CARDINAL;
      feld       : Zahlenfeld;

BEGIN
    Einlesen;
    Sortieren;
    Ausgeben;
END.
```


Entwickeln

Verfeinerung

```
PROCEDURE Einlesen () =
VAR x : INTEGER;
BEGIN
  anzahl := 0;
  x := GetInt();

  WHILE NOT EndOfFile() DO
    x := GetInt();
    anzahl := anzahl + 1;
    feld[anzahl] := x;
  END;

END Einlesen;
```

```
PROCEDURE Ausgeben () =
BEGIN

  FOR nr := 1 TO anzahl DO
    SIO.PutInt(feld[nr]);
    IF nr MOD 10 = 0 THEN
      SIO.Nl();
    END;
  END;

END Ausgeben;
```

Entwickeln

Sortierverfahren

- **Hauptteil Sortieren**
  - einfache Verfahren: n2
  - gute Verfahren: n log n
  - Sortierstrategien
    - Einfügen
    - Auswählen
    - Vertauschen ← Bubblesort
    - Verschmelzen

→ Vorlesung Datenstrukturen und Algorithmen

Entwickeln

Bubblesort

```
PROCEDURE Bubblesort () =
VAR hilf : INTEGER;
BEGIN

  FOR k := 1 TO anzahl DO
    FOR i := 1 TO anzahl-1 DO
      IF feld[i] > feld[i+1] THEN
        Vertausche(feld, i,i+1);
      END;
    END;
  END;

END Bubblesort;
```

Verbessern

Verbesserungen - 1. Schritt

i	a					
Undef.	16	33	94	82	6	58
1	16	33	94	82	6	58
2	16	33	94	82	6	58
3	16	33	82	94	6	58
4	16	33	82	6	94	58
5	16	33	82	6	58	94

1. *Durchlauf größte Zahl am richtigen Platz*
2. *Durchlauf zweitgrößte Zahl am richtigen Platz*

...

- **1. Brauchen nur maximal Anzahl-1 Durchläufe**
  - letzte Zahl ist dann automatisch am richtigen Platz
  - letzter Durchlauf ist überflüssig
- **2. Brauchen beim k-ten Durchlauf nur bis Anzahl-k zu laufen**
  - die restlichen Zahlen sind bereits sortiert
- **3. Abbruchkriterium einführen:**

  Nicht mehr durchlaufen, wenn im letzten Durchgang nicht vertauscht worden ist.

Verbessern

Verbesserung

```
PROCEDURE Bubblesort ()=
VAR getauscht : BOOLAEN;
BEGIN

  FOR k  := 1 TO anzahl-1 DO          (*1*)
    getauscht := FALSE;
    FOR i := 1 TO anzahl-k DO         (*2*)
      IF feld[ i ] > feld[ i+1 ] THEN
        Vertausche(feld, i, i+1);
        getauscht := TRUE;
      END;
    END;
    If NOT getauscht THEN EXIT;       (*3*)
  END;

END Bubblesort;
```

Verbessern

Verbesserung - 2. Schritt

k	i	a					
		16	33	94	82	6	58
1	5	16	33	82	6	58	94
2	1	16	33	82	6	58	94
2	2	16	33	82	6	58	94
2	3	16	33	6	82	58	94
2	4	16	33	6	58	82	94
3	1	16	33	6	58	82	94
	2	16	6	33	58	82	94
	3	16	6	33	58	82	94

Überflüssig bis Anzahl-k zu laufen, wenn im letzten Durchlauf nicht bis dorthin vertauscht wurde.

Verbessern

Verbesserung 2. Schritt

```
PROCEDURE Bubblesort() =
VAR hilf : INTEGER;
    rechtesEnde, letztesI: Index;
BEGIN
  letztesI := anzahl;

  REPEAT
    rechtesEnde := letztesI;
    letztesI := 0;
    For i := 1 TO rechtesEnde -1 DO
      IF feld[i] > feld[i+1] THEN
        Vertausche(feld, i, i+1);
        letztesI := i;
      END;
    END;
  UNTIL letztesI := 0;

END Bubblesort;
```

Rechtes Ende pro Durchlauf nicht nur um 1 verkleinern

Effizienzbetrachtungen

- **best-case-Analyse**
  - geordnet
  - 1 Schleifendurchlauf
  - n-1 Vergleiche, 0 Vertauschungen
  - hier Bubblesort gut !
- **worst-case-Analyse**
  - Umgekehrt geordnet
  - n-1 Durchläufe
  - k-ter Durchlauf: innere Schleife: (n-k)-mal vertauscht

$$\sum_{k=1}^{n-1} (n-k) = \frac{(n-1)(n-2)}{2} = \frac{n^2}{2} - \frac{3n}{2} - 1$$

Anzahl der Vergleiche/ Vertauschungen

- **Bubblesort**
  - $O(n^2)$ d.h. Konstante a, b, c mit
  - f^{wc}_{Zeit} (Bubblesort, FeldDerLaenge) $\leq an^2 + bn + c$

Testen von Programmen

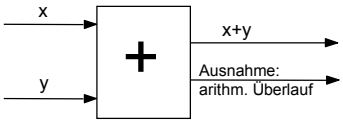
- Definitionen
- Black-box Testen
- White-box Testen
- Test-Prinzipien

Testen - Definition

- **Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden. (Myers 1979)**
  - Wurde ein Programm sorgfältig getestet (und sind alle gefundenen Fehler korrigiert), so ***steigt die Wahrscheinlichkeit***, daß das Programm sich auch in den nicht getesteten Fällen wunschgemäß verhält.
- **Ziel des Tests ist es,**
  - Fehler zu entdecken!
- **Ein Test ist erfolgreich,**
  - wenn er einen Fehler gefunden hat.

Testen - Ziele

- **Ein erfolgloser Test ist niemals ein Beweis für ein korrektes Programm – es wurden lediglich keine Fehler gefunden!**
  - Die Korrektheit eines Programms kann durch Testen (außer in trivialen Fällen) nicht bewiesen werden.
- **Grund: alle Kombinationen aller möglichen Werte der Eingabedaten müßten getestet werden**
  - Anzahl möglicher Eingaben: $2^{16} \cdot 2^{16} = 2^{32}$
  - Ein vollständiger Test erfordert mehr als 4'000'000'000 Testfälle
  - Ein Programm kann niemals “ausgetestet” werden!

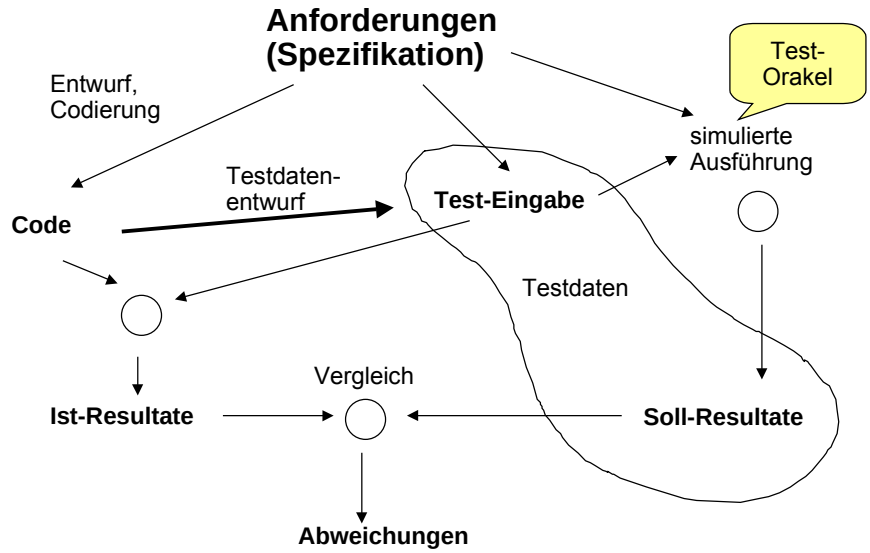


x, y ganze Zahlen, mit 16 Bit repräsentiert

Testfälle

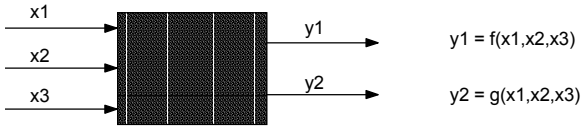
- **Auswahl der Testfälle ist die zentrale Aufgabe des Testens!**
  - Anforderungen an Testfälle
    - repräsentativ
    - fehlersensitiv
    - redundanzarm
    - ökonomisch
- **Ziel:**
  - Mit einer möglichst kleinen Auswahl der Testfälle möglichst vielen Fehlern auf die Spur kommen.
- **Auswahl entsprechend der (angestrebten Merkmale des Programms (Basis: Spezifikation)**
  - Black-Box-Test (Funktionstest)
- **Auswahl unter Einfluß der inneren Struktur des Prüflings**
  - Glass-Box-Test (Strukturtest)

Vorgehensweise beim Test

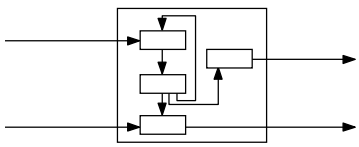


Testverfahren

- **funktionsorientiert (Black-Box-Test)**
  - Testfall-Auswahl aufgrund der Spezifikation
  - Programmstruktur kann unbekannt sein



- **strukturorientiert (White-Box-Test, Glass-Box-Test)**
  - Testfall-Auswahl aufgrund der Programmstruktur
  - Spezifikation muß ebenfalls bekannt sein (wegen der erwarteten Resultate)



Black-box Test (funktionaler Test)

- **Ausgangspunkt für Testfälle ist die Spezifikation**
  - Fehlt eine Spezifikation, dann hat das erhebliche Konsequenzen auch für das Testen
  - Wogegen soll getestet werden? Was sind die Sollergebnisse?
- **Motivation**
  - Es ist nicht zulässig, ein Programm lediglich gegen sich selbst zu testen.
- **Ziel**
  - Möglichst umfassende Prüfung der ***spezifizierten Funktionalität***.
- **Nachteile**
  - Die konkrete Implementierung wird ***nicht geeignet*** berücksichtigt
  - Häufig wird mit einem reinen Black-box Test nicht die Minimalanforderung von White-box-Tests erzielt
    - 100 % Zweigüberdeckung

Black-box - Testfallauswahlkriterien

- **Funktionsüberdeckung**
  - Jede spezifizierte Funktion wird mindestens einmal ausgeführt.
- **Eingabeüberdeckung**
  - Jedes Eingabedatum wird in mindestens einem Testfall verwendet.
  - Macht in der Regel Probleme!
- **Ausgabeüberdeckung**
  - Jede “Ausgabesituation” wird mindestens einmal erzeugt.
  - Beispiele: Bildschirmmasken, Fehlermeldungen, etc.
- **Aufwand dafür ist z.T. erheblich**
  - Bspl: Textverarbeitungsprogramm
  - Häufig müssen Funktionen in ihrem Zusammenspiel geprüft werden.
  - Zusätzlich müssen Leistungs- und Robustheitsprüfungen gemacht werden

Techniken der Testfall-Auswahl

- **Äquivalenzklassenbildung**
  - Um eine repräsentative Menge von Eingabedaten zu testen, werden die Eingaben in Äquivalenzklassen eingeteilt. Aus jeder Klasse wird ein Repräsentant getestet.
- **Grenzwertüberprüfung**
  - Da an den Grenzen zulässiger Datenbereiche erfahrungsgemäß häufig Fehler auftreten, werden außerdem auch Testfälle für solche Grenzfälle definiert.

Äquivalenzklassenbildung

- **Prinzip**
  - Zerlegung aller möglicher Eingabedaten in
    - gültige und
    - ungültige Äquivalenzklassen
  - Jede Äquivalenzklasse wird durch einen (oder mehrere) Repräsentanten getestet.
  - Werte aus einer Äquivalenzklasse verursachen ein identisches funktionales Verhalten.
  - Somit können alle Programmfunktionen getestet werden.
  - Die Anzahl der Testfälle wird reduziert.
  - Äquivalenz ist hypothetisch
    - Klassen werden z.T. nach Erfahrung und Intuition gebildet
- **Zerlegung mit Hilfe von**
  - spezifizierten Gültigkeitsbereichen
  - spezifizierten (oder vermuteten) Sonderbehandlungen

Vorgehensweise

- **Schritt 1:**
  - Aufstellung von Eingabebedingungen
- **Schritt 2:**
  - Bildung von Äquivalenzklassen für jede Eingabebedingung.
- **Schritt 3:**
  - Äquivalenzklassen nummerieren.
- **Schritt 4:**
  - Testfälle definieren (gültige Ä-Klassen)
- **Schritt 5:**
  - Testfälle definieren (ungültige Ä-Klassen)

Beispiel: Ä-Klassenbildung - 1

- **Spezifikation eines PRINT-Befehls**
  - Mit dem Ausgabebefehl PRINT wird die Datei auf den Bildschirm ausgegeben.
  - Der Befehl hat zwei Parameter: Dateiname und Zeilenanzahl.
  - Beide Parameter müssen angegeben werden.
  - PRINT hat folgende Syntax: PRINT <Dateiname> <Zeilenanzahl>
  - Der Dateiname besteht aus mindestens einem und bis zu sechs Zeichen, die Buchstaben oder Ziffern sein können.
  - Das erste Zeichen des Dateinamens muß ein Buchstabe sein.
  - Die Zeilenanzahl besteht aus mindestens einer und bis zu 3 Ziffern. Sie muß größer 0 und kleiner 1000 sein.

Beispiel: Ä-Klassenbildung - 2

- **Schritt 1: Eingabebedingungen definieren**
- **Schritte 2 und 3: Äquivalenzklassen finden**

Eingabebedingungen	gültige Ä-Klasse	ungültige Ä-Klasse
Anzahl der Parameter	zwei (1)	keine(1a), einen(1b) mehr als 2 (1c)
Dateiname (Länge)	1 bis 6 (2)	0 (2a), >6(2b)
Dateiname (Zeichen)	Buchst. oder Ziffern (3)	sonstige Zeichen(3a)
Dateiname (erstes Zeichen)	Buchstabe(4)	kein Buchstabe(4a)
Zeilenanzahl(Zeichen)	Ziffern (5)	ein Zeichen ist keine Ziffer(5a)
Zeilenanzahl (Ziffern)	1 bis 3 (6)	>3 (6a), 0 (6b = 1b)
Zeilenanzahl (Größe)	1 <= w <= 999 (7)	w <=0 (7a), w >=1000(7b)

Diskussion Ä-Klassenbildung

- **Feststellung**
  - Bei der Äquivalenzklassenmethode ist die ***Güte*** der Testfälle abhängig von der ***Aussagekraft*** der Spezifikation
  - Bspl:
    - Führende Nullen beim zweiten Parameter des PRINT Befehls
    - Die Spezifikation macht keine Aussagen dazu:
      - Es entstehen zwei ungültige Ä-Klassen (mehr als 3 Zeichen, Wert >= 1000)
- **Die Definition der Testfälle hängt nicht nur von den Eingabebedingungen ab**
  - Welche ***Werte*** aus einer Ä-Klasse sollen gewählt werden?
  - Welche Kombinationen von ***Eingabebedingungen*** sollen getestet werden?

Grenzwertanalyse -1

- **Wahl der Repräsentanten einer Ä-Klasse**
  - Besteht eine Ä-Klasse aus einer ***geordneten Menge von Werten***, dann kann zur Auswahl von Repräsentanten die Grenzwertanalyse durchgeführt werden.
- **Schritt 1 der Grenzwertanalyse**
  - Testdaten identifizieren, die direkt auf oder neben den Grenzen des Ä-Klasse liegen (plus einen mittleren Wert).
    - Richtlinien
      - Wertebereich :
        - gültige Testwerte: kleinster und größter Wert
        - ungültige Werte: Werte, die direkt daneben und außerhalb liegen
      - Beispiel
        - Natürliche Zahlen 1 .. 255
        - 4 Repräsentanten: 0, 1, 255, 256
        - Liste maximal 100 Elemente
        - 4 Repräsentanten der Längen: 0, 1, 100, 101

Grenzwertanalyse -2

- **Schritt 2 der Grenzwertanalyse**
  - Zusätzlich zu den Eingabe-Ä-Klassen werden ***Ausgabe-Ä-Klassen*** für die erwarteten Resultate gebildet.
  - Dies geschieht ***analog*** zur Vorgehensweise bei der Definition der Eingabe-Ä-Klassen.
  - Ausgabe-Ä-Klassen stellen ***Soll-Werte*** dar, für die die Eingabedaten bestimmt werden müssen.
  - Hinweis:
    - Es ist manchmal nicht möglich, diese Eingabedaten zu finden, da eine zu produzierende ungültige Ausgabe nicht vom Programm zugelassen wird.

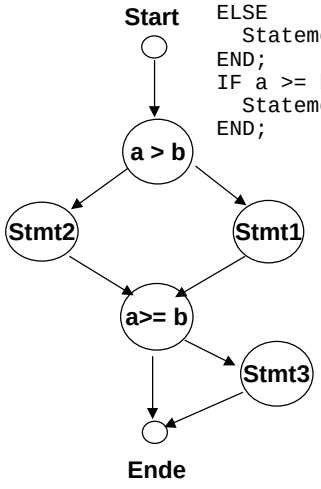
White-box Tests

- **Auswahl der Testfälle so, daß der Programmablauf oder der Datenfluß im Programm überprüft wird.**
  - Meistens wird der Programmablauf getestet: Testfälle werden so gewählt, daß das Programm systematisch durchlaufen wird.
- **Gebräuchlich sind drei sogenannte Testüberdeckungen:**
  - Anweisungsüberdeckung
  - Zweigüberdeckung
  - Pfadüberdeckung
- **Nachteil**
  - Fehlende, in der Spezifikation beschriebene Funktionalitäten, werden nicht erkannt.
- **Vorteil**
  - Es lassen sich formale Testauswahlkriterien für wb-Tests definieren.

Ablaufgraph

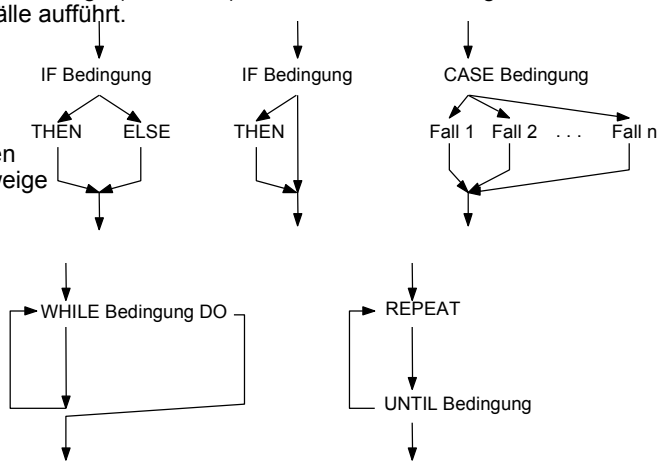
- **Programm wird als Flußdiagramm betrachtet:**
  - Anweisung: Knoten
  - Zweig: Kante nach Bedingung
  - Pfad: Weg vom Anfangs- bis zum Endknoten
- **Überdeckungen können nur rechnerunterstützt ermittelt werden:**
  - Ein Testinstrumentierer fügt Zähler in Knoten und Kanten ein
  - Der Überdeckungsgrad wird über mehrere Läufe kumuliert

```
IF a > b THEN
  Statement1;
ELSE
  Statement2;
END;
IF a >= b THEN
  Statement3;
END;
```



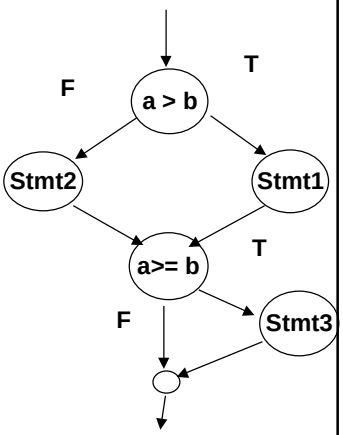
Zweige / Pfade im Programm

- **Bestimmung der Programmzweige:**
  - Betrachtung von Verzweigungen und Schleifen. Bei Programmiersprachen mit geschlossenen Ablaufkonstrukten (z.B. Pascal) haben jede IF-Anweisung und jede Schleife je zwei Zweige (siehe Bild). Eine CASE-Anweisung hat soviele Zweige, wie sie Fälle aufführt.
- **Bestimmung der Pfade (Wege):**
  - Alle Kombinationen aller Programmzweige bei maximalem Durchlauf aller Schleifen



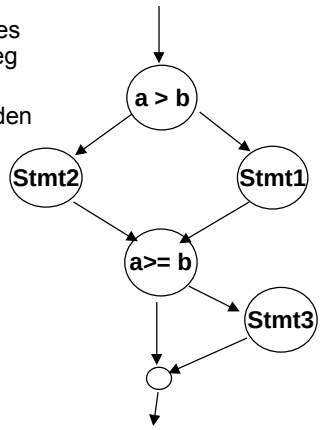
Anweisungsüberdeckung

- **Co-Test ist eine einfache kontrollflußorientierte Testmethode**
- **Definititon**
  - Eine Testfallmenge T erfüllt das Co-Kriterium, wenn es für jede Anweisung A des Programms P einen Testfall gibt, der die Anweisung A ausführt.
- **Überdeckungsgrad** $\frac{\text{ausgeführte Statements}}{\text{Anzahl aller Statements}}$
- **Black-Box-Testfälle produzieren ca. 60-70% Anweisungsüberdeckung.**
- **Angestrebt werden 95-100%**
- **Bewertung**
  - notwendiges aber schwaches Kriterium
  - hilft dead-code zu finden
  - gewisse Kontrollflußfehler werden nicht (immer) gefunden



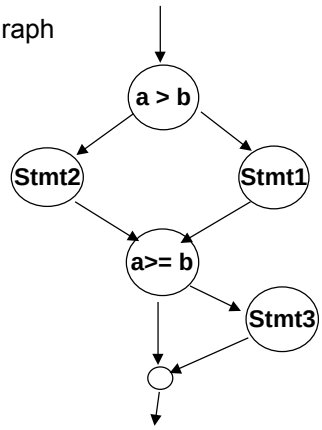
Zweigüberdeckung

- **C1-Test ist eine dynamische kontrollflußorientierte Testmethode**
- **Definition**
  - Eine Testfallmenge T erfüllt das C1-Kriterium, wenn es für jede Kante k im Kontrollflußgraph von P einen Weg in Wege(T,P) gibt, zu dem k gehört.
  - D.h. wenn alle Entscheidungskanten ausgeführt werden
- **Überdeckungsgrad** $\frac{\text{ausgeführte Kanten}}{\text{Anzahl aller Kanten}}$
- **Black-Box-Testfälle produzieren ca. 80% Zweigüberdeckung.**
  - Angestrebt werden 100%
- **Bewertung**
  - minimales Testkriterium
  - hilft dead-code zu finden
  - berücksichtigt nicht komplexe Bedingungen



Pfadüberdeckung

- **C2-Test ist die intensivste kontrollflußorientierte Testmethode**
- **Ziel**
  - Alle unterschiedlichen Pfade sollen einmal ausgeführt werden.
  - Pfad: Sequenz von Knoten im Ablaufgraph
- **Problem**
  - Pfadüberdeckung ist für reale Programme unrealistisch, da die Anzahl der Pfade durch Schleifen astronomisch hoch sein kann.



Eingeschränkte Pfadüberdeckung

- **Problem**
  - Schleifen führen zu sehr vielen/unbekannt vielen Pfaden
  - Dieser Überdeckungsgrad wird nur bei sicherheitskritischer Software angestrebt.
- **Idee**
  - Nicht alle Pfade durch Schleifen werden berücksichtigt.
  - Nur die Pfade, die neue Aspekte ansprechen.
- **Beschränkung auf 5 Aspekte pro Schleife:**
  - 0 Iterationen: die Schleife wird nicht betreten
  - 1 Iteration: zeigt häufig Initialisierungsfehler
  - 2 Iterationen: kann ebenfalls Initialisierungsfehler zeigen
  - typ. Anzahl Iterationen: Normalfall muß auch geprüft werden
  - max. Iterationen: zeigt Fehler beim Abbruchkriterium

Beispiel: eing. Pfadüberdeckung

```
:
Kandidat := 1;
WHILE (Kandidat < Anzahl) AND (Elemente [Kandidat] <> Suchwert) DO
   Kandidat := Kandidat + 1;
END;
```

- **0 Iterationen:**
- **1 Iteration:**
- **2 Iterationen:**
- **typ. Anzahl Iterationen:**
- **max. Iterationen:**

Testen - Sollergebnis

- **Testen setzt voraus, daß die erwarteten Ergebnisse bekannt sind**
  - Entweder muß gegen eine ***Spezifikation*** oder gegen vorhandene Testergebnisse
  - (z.B. bei der Wiederholung von Tests nach Programm-Modifikationen) getestet werden (sogenannter Regressionstest)
- **Unvorbereitete und undokumentierte Tests sind sinnlos!**
- **Mit Testen können nicht alle Eigenschaften eines Programms geprüft werden**
  - Wartbarkeit, etc.
- **Mit Testen werden nur Fehlersymptome,**
  - nicht aber die ***Fehlerursachen*** gefunden!

Prinzipien des Testens

- **Vollständiges Testen ist unmöglich!**
- **Zu jedem Testfall gehört ein Soll-Resultat**
  - Nur ein auffällig falsches Resultat springt ins Auge!
- **Teste niemals dein eigenes Programm**
  - Kein Programmierer will zeigen, daß sein Progamm Fehler hat
- **Testfälle müssen auch für ungültige und unerwartete Eingaben definiert werden!**
- **Vermeide Wegwerftestfälle, es sei denn das Programm ist wirklich ein Wegwerfprogramm.**
- **Ein erfolgreicher Testfall ist dadurch gekennzeichnet, daß er einen bisher unbekannten Fehler entdeckt.**
- **Ein Test ist nur so gut wie seine Testfälle!**

Vor- und Nachteile von Tests

- **Vorteile**
  - Testen ist ein ***natürliches*** Prüfverfahren
  - Tests sind ***reproduzierbar*** (=> objektiv)
  - investierter Aufwand ***mehrfach*** nutzbar
  - ***Zielumgebung*** wird mitgeprüft
  - Systemverhalten wird sichtbar gemacht
- **Nachteile**
  - Ergebnisse werden ***überschätzt*** (Korrektheitsaussage unmöglich)
    - Tests sind Stichprobenverfahren
  - ***nicht alle*** Programm-Eigenschaften sind testbar
  - nicht alle ***Anwendungssituationen*** sind nachbildbar
  - Test zeigt die Fehlerursache ***nicht***

Modularisierung und Module

- **Modulkonzept**
  - Export-Schnittstelle
  - Import-Schnittstelle
- **Modulrumpf**
- **Austausch der Implementierung**
- **Diskussion modularer Programme**

Modul-konzept

Vorteile modularer Programme

- **Module können von *unterschiedlichen* Personen entwickelt und gepflegt werden.**
  - Software-Entwicklung ist Team-Arbeit!
- **Module können einzeln *getestet* werden.**
  - Test großer Programme ist extrem aufwendig!
- **Module können geordnet zum Gesamtsystem *integriert* werden.**
- **Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung *ersetzt* werden.**
  - z.B. durch eine effizientere Implementierung
- **Module können in verschiedenen Programmen *wiederverwendet* werden (Modul-Bibliothek).**
  - Dies senkt die Kosten für die Entwicklung!

Modul-konzept

Module

- **Neuere imperative Sprachen sehen Module vor**
- **Entstanden aus der Notwendigkeit,**
  - große Programmtexte in für den ***Übersetzer faßliche Einheiten*** zu zerlegen,
  - Modulkonzept ist zum zentralen ***Organisationskonzept*** für Entwürfe und Programmtexte geworden.
- **Module werden hier als**
  - *Konstruktionshilfsmittel* der Sprache eingeführt.
- **Die Diskussion,**
  - wie das Modulkonzept genutzt werden sollte, folgt in einem eigenen Kapitel.

Modul-konzept

Definition: Modul

- **programmiersprachliche Definition:**
  - Ein Modul ist die ***Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren*** zu einer Einheit. Soll ein Modul von einem anderen benutzt werden, so muß man angeben, welche Teile der Schnittstelle dieses Moduls von ***außen sichtbar*** sein sollen und welche nicht. Grundsätzlich bleibt aber die Implementierung eines Moduls, also die konkrete Realisierung der Datentypen und Prozedurrümpfe, vor allen anderen Modulen ***verborgen***.
- **methodische Definition:**
  - Unter einem Modul verstehen wir eine ***Sammlung von Objekten und Algorithmen*** mit der Eigenschaft, daß ihre Kommunikation mit der Außenwelt nur über eine klar ***definierte Schnittstelle*** erfolgt. Das ***Zusammensetzen*** mehrerer Module zu einer Gesamtlösung darf keine Kenntnis ihres ***inneren Aufbaus*** voraussetzen, und die Korrektheit eines Moduls muß ohne Kenntnis seiner Einbettung in die Gesamtlösung nachprüfbar sein.

Modul-konzept

Erinnerung: Lebensdauer

- **Prozedur:**
  - Alle Objekte im Namensraum einer Prozedur existieren nur solange die Prozedur aktiv ist.
  - Bei jedem neuen Aufruf einer Prozedur werden u.a. die lokalen Variablen neu angelegt.
- **Modul:**
  - Module sind ***statisch***, d.h. ihr Namensraum existiert solange das Programm oder die Anwendung ***insgesamt*** aktiv ist.
  - Variablen, die im Deklarationsteil eines Moduls eingeführt werden, haben die gleiche Lebensdauer wie das Modul; sie heißen ***global***.
- **In Modula-3**
  - können Module ***nicht*** ineinander geschachtelt werden!
  - Bei Prozeduren ist dies möglich!

Modul-konzept

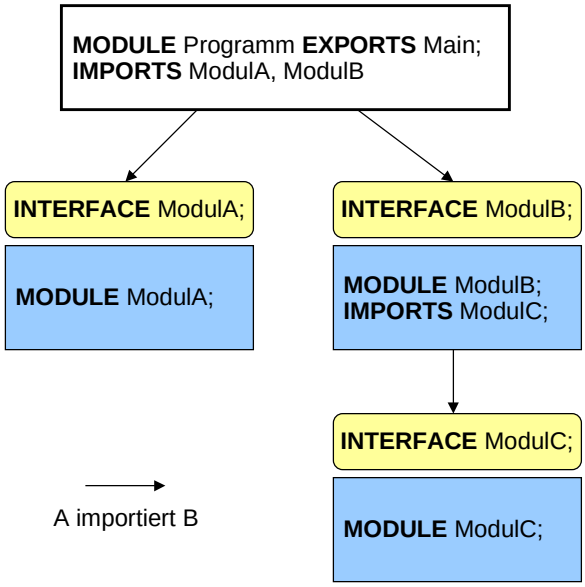
Aufbau von Modula-3 Programmen

- **Modula-3 Programm**
  - besteht wenigstens aus einem ***Modul***, dem ***Hauptmodul***
- **Modul-Aufbau**
  - ein Modul besteht (bis auf das Hauptmodul) aus
    - ***Schnittstelle*** (interface)
      - definiert, was ein Modul exportiert
      - Exportschnittstelle
    - ***Implementierung*** (body, Rumpf)
      - enthält die Implementierung der exportierten Elemente
      - versteckt die Implementierung
- **Bisher**
  - bestanden unsere Programme lediglich aus einem Modul, dem Hauptmodul und weiteren Prozeduren

Modul-konzept

Modul-Hierarchie

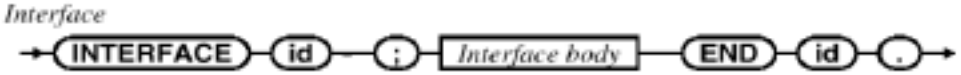
- ein Modul kann Elemente anderer Module ***benutzbar machen***
  - ein Modul ***importiert*** dazu andere Module
  - von einem importierten Modul ist nur die ***Schnittstelle*** sichtbar

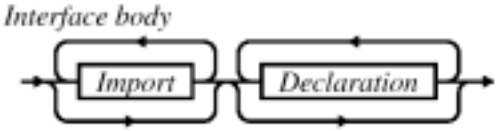


Modul-konzept

Die Export-Schnittstelle

- **Wie sieht eine Export-Schnittstelle eines Moduls aus?**





- **Alle in der Schnittstelle deklarierten Objekte werden vom Modul *exportiert***
  - können von importierenden Modulen ***verwendet*** werden.
- **Hinweis**
  - alles was in der Schnittstelle "***versprochen***" wird, muß auch von der Implementierung realisiert werden

Modul-konzept

Beispiel: Export-Schnittstelle

```
INTERFACE SIO;

IMPORT Fmt, Rd, Wr, Word;

...

PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};
(* Read next character from stream rd and return it. *)

PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL);
(* Write ch to outputstream wr. *)

PROCEDURE GetText(rd: Reader := NIL; len: CARDINAL): TEXT;
(* Read a sequence of len characters from rd and return them. If there are not
enougth characters return what is there. *)

PROCEDURE PutText(t: TEXT; wr: Writer := NIL);
(* Write character sequence t to outputstream wr. *)

PROCEDURE GetLine(rd: Reader := NIL): TEXT RAISES {Error};
(* Read a full line of text terminated by the next RETURN from
   inputstream rd and return it (without RETURN!). *)
. . .
END SIO.
```


Modul-konzept

IMPORT-Schnittstelle - 1

- **Um Programmobjekte über Modulgrenzen zu verwenden,**
  - müssen sie ***gezielt angefordert*** werden.
  - Die IMPORT-Klausel dient dazu,
    - die Dienste in einem anderen Modul ***sichtbar*** zu machen.
- **Mögliche IMPORT-Varianten:**
  - importieren ***aller*** Dienste eines Moduls
  - importieren ***aller*** Dienste eines Moduls unter einem ***Alias-Namen***
  - importieren nur der Dienste eines Moduls, die ***tatsächlich*** vom importierenden Modul verwendet werden

Modul-konzept

IMPORT-Klausel - 2

- **Beispiele für Importe:**
  - wird nicht selektiv importiert, muß der Modulname als ***Qualifikator*** verwendet werden

```
IMPORT Text;                          (* import aller Deklarationen *)
IMPORT Rd AS Reader;                  (* import mit Alias-Namen *)
FROM SIO IMPORT                       (* selektives Importieren *)
  (*PROCS*) PutLine, PutText, GetChar;
…
VAR eingabestrom : Reader.T;
…
n := Text.Length(t1);
…
PutText("abcdefg");
```

Modul-konzept

Diskussion: Import-Varianten

- **Globales Importieren**

  An ***jeder Stelle*** im Programmtext ist ersichtlich, wo das jeweilige Objekt deklariert ist.

  ```
  Text.Length(t1);   Length(m3);   List.Length(l1);
  ```

  Identisch deklarierte Bezeichner verschiedener Module können verwendet werden.

  Zum Teil erheblich ***längere Schreibweise.***

  Erst mit einem Werkzeug kann einfach ermittelt werden, was ***tatsächlich*** alles von einem Modul verwendet wird.

- **Selektives Importieren**
  - ***kürzere*** Schreibweise
  - Es ist alles das, was verwendet wird, auch ***explizit angegeben***
  - Dies erhöht die Änderbarkeit

    Es ist nicht direkt ersichtlich, woher ein Dienst kommt.

Modul-konzept

Regeln

- **1. Schnittstelle und Implementierung eines Moduls sind in *unterschiedlichen* Dateien (Programmtextdatei) enthalten.**
  - Jede Programmtextdatei bildet eine ***Übersetzungseinheit*** und kann vom Übersetzer getrennt behandelt werden.
- **2. Import**
  - alle Dienste: Qualifikation zeigt Herkunft des Dienstes
  - selektiver Import: Erhöht die Änderungsfreundlichkeit
- **3. Zyklische Importe sind verboten!**
  - A importiert B, B importiert A
  - zyklischer Import ist ein Hinweis auf eine ***schlechte Modularisierung***

Modul-konzept

Import für Schnittstelle oder für Rumpf

- **Import zur Schnittstellendefinition**
  - z.B. Typ für Deklaration von Formalparameter oder Ergebnis
  - Konstante für Vorberechnung von Parametern
- **Import für Rumpfrealisierung**
  - z.B. Typ für die Realisierung einer modulrumpflokalen Variablen
  - Prozedur/Funktion als Hilfe für die Implementierung einer Schnittstellenoperation oder des Anweisungsteils

Modul-rumpf

Implementierung eines Moduls

- **Regel:**
  - Die Implementierung einer Schnittstelle realisiert ***alle*** in der Schnittstelle ***deklarierten*** Prozeduren (Funktionen).
- **EXPORTS-Klausel**
  - gibt an, welche Schnittstelle ein Modul realisiert

```
MODULE Geometrie EXPORTS Geometrie;
```

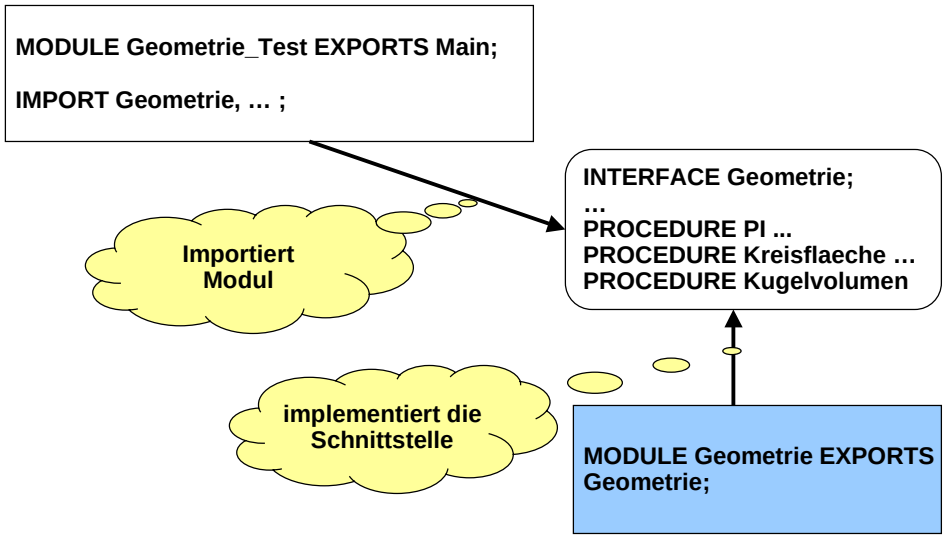
  - Fehlt die EXPORTS-Klausel, dann realisiert das Modul eine Schnittstelle ***gleichen Namens.***

```
MODULE Geometrie;
```

- **Modul-Initialisierung**
  - Im ***Block*** eines Moduls wird das Modul initialisiert.
  - Regel: Ein importiertes Modul wird vor dem importierenden Modul initialisiert.

Modul-rumpf

Beispiel



Modul-rumpf

Beispiel: Schnittstelle

```
MODULE Geometrie_Test EXPORTS Main;
IMPORT Geometrie, SIO;

VAR radius : REAL;
BEGIN
  SIO.PutText ("Geben Sie bitte einen Radius ein: ");
  radius := SIO.GetReal();
  SIO.PutText ("Kreisflaeche: ");
  SIO.PutReal (Geometrie.Kreisflaeche(radius));
  SIO.Nl();
  SIO.PutText ("Kugelvolumen: ");
  SIO.PutReal (Geometrie.Kugelvolumen(radius));
END Geometrie_Test.
```

```
INTERFACE Geometrie;

PROCEDURE PI () : REAL;
PROCEDURE Kreisflaeche(r :REAL): REAL;
PROCEDURE Kugelvolumen(r : REAL): REAL;

END Geometrie.
```


Modul-rumpf

Beispiel: Implementierung

```
MODULE Geometrie EXPORTS Geometrie;

VAR pi : REAL;

PROCEDURE PI () : REAL =
BEGIN
  RETURN pi;
END PI;

PROCEDURE Kreisflaeche(radius :REAL): REAL =
BEGIN
  RETURN (pi * radius * radius);
END Kreisflaeche;

PROCEDURE Kugelvolumen(radius : REAL): REAL =
BEGIN
  RETURN ( (4.0/3.0) * pi * radius * radius * radius );
END Kugelvolumen;

BEGIN
  pi := 3.14147;
END Geometrie.
```

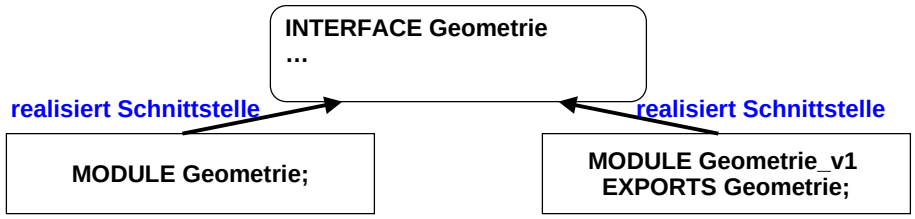
Interne Variable

Realisierung der Prozeduren / Funktionen der Schnittstelle

Initialisierung der Moduls

Austausch der Implementierung

Zwei Rümpfe für die gleiche Schnittstelle




```
% Makefile ...
…

% module(Geometrie)

interface(Geometrie)
implementation (Geometrie_v1)

implementation (Geometrie_Test)
program (Geometrie_Test)
```

```
MODULE Geometrie_v1 EXPORTS Geometrie;

PROCEDURE PI () : REAL =
BEGIN
  RETURN 3.14147;
END PI;

PROCEDURE Kreisflaeche(r :REAL): REAL = ...

PROCEDURE Kugelvolumen(r : REAL): REAL = ...
BEGIN
END Geometrie_v1.
```


Diskussion modularer Programme

Vorteile modularer Programme

- **Vorteile modularer Programme**
  - Module können von ***unterschiedlichen*** Personen entwickelt und gepflegt werden.
    - Software-Entwicklung ist Team-Arbeit!
  - Module können einzeln ***getestet*** werden.
    - Test großer Programme ist extrem aufwendig!
  - Module können geordnet zum Gesamtsystem ***integriert*** werden.
  - Eine Implementierung eines Moduls kann leicht durch eine neue Implementierung ***ersetzt*** werden.
    - z.B. durch eine effizientere Implementierung
  - Module können in verschiedenen Programmen ***wiederverwendet*** werden (Modul-Bibliothek).
    - Dies senkt die Kosten für die Entwicklung!

Diskussion modularer Programme

Modulkonzept

- **Module sind Sammlungen von Programmobjekten und Algorithmen:**
  - Sie sind keine ***direkt aufrufbaren*** Programmeinheiten (wie Prozeduren).
  - Sie sind eine Einheit für die ***Übersetzung***.
- **Als Sammlung sollen sie *keine* beliebige Anordnung sein**
  - Kriterien für die Zusammenstellung eines Moduls müssen geklärt werden.
  - Module verbergen ***Implementierungen*** d.h. ihren inneren Aufbau und zeigen nur ihre Schnittstelle:
  - Es gibt unterschiedlich ***starke Möglichkeiten*** des Verbergens.
  - Der Aufbau einer Schnittstelle unterliegt bestimmten Kriterien.
- **Module benutzen andere Module:**
  - Das Zusammenspiel verschiedener Module bezeichnet man auch als ***Architektur***.
  - Die ***Import-Beziehungen*** koppeln Module miteinander.

Was haben wir gelernt?

- **Module als Sprachkonstrukt moderner imperativer Programmiersprachen**
- **Modul Schnittstelle - Rumpf, verschiedenen Rümpfe zu einer Schnittstelle**
- **Vorteile der Modularisierung bzgl. Qualität und Effizienz der Softwareerstellung bzw. bzgl. Qualität des resultierenden Programmsystems**
- **Modulhierarchie über Importbeziehungen**
- **Implementierung eines Moduls: Realisierung der Dienste und Initialisierung**

Glossar

- **Exportschnittstelle, Importschnittstelle eines Moduls, Rumpf eines Moduls**
- **getrennte Übersetzung von Schnittstelle und Rumpf**
- **programmiersprachliche und methodische Definition eines Moduls**
- **statische Lebensdauer von Modulvariablen (immer im Rumpf)**
- **Import für Schnittstelle oder für Rumpf**
- **Schnittstellenimplementierung, Implementierung der Initialisierung**
- **schlechter Sprachgebrauch Interface (Module), (Implementation) Module**

Datenabstraktion

- **Prozeß- und Datenabstraktion**
- **Objektmodule (Datenkapselung)**
- **Abstrakte Datentypen**

Programmieren im Großen

- **Programmieren im Kleinen**
  - befaßt sich mit der Konstruktion eines ***Programms***. Wir haben bisher die dazu notwendigen Konzepte kennengelernt:
    - Daten- und Kontrollstrukturen,
    - Typen,
    - Prozeduren und Funktionen.
- **Programmieren im Großen**
  - für die Entwicklung großer Softwaresysteme müssen weitere Konzepte hinzukommen. Das vorrangige Problem ist, die ***Komplexität*** großer Softwaresysteme zu beherrschen.
  - Die gewählte Lösung heißt ***Abstraktion***.
- **Für den modularen Softwareentwurf sind zwei Abstraktionskonzepte entscheidend:**
  - ***Prozeßabstraktion***,
  - ***Datenabstraktion***.

Prozeß- und Datenabstraktion

Abstraktion: Prozeß, Daten

- **Prozeßabstraktion (auch algorith. Abstraktion):**
  - Funktionen und Prozeduren werden zu ***abstrakten*** Konzepten.
  - Module können gleichartige zusammenfassen: funktionaler Modul.
  - Bekannt sind nur die ***Eingabe***- und ***Ausgabegrößen***, aber nicht die Implementation.
  - z.B.: Für eine mathematische Funktion ist nur wesentlich, daß sie ein korrektes Ergebnis liefert, aber nicht wie dies geschieht
- **Datenabstraktion:**
  - Die ***Details*** der verwendeten Daten werden verborgen.
  - Schließt konzeptionell die Prozeßabstraktion für die Zugriffsoperationen ein.
  - Beispiel:
    - Für eine Liste ist wichtig, daß sie ein Behälter für Elemente ist und daß bestimmte Zugriffsoperationen erlaubt sind,
    - aber nicht, wie die Elemente der Liste gespeichert und bearbeitet werden.
  - Eine weitere von Datenabstraktion abstrahiert auch von der ***Art der Elemente***, die in der Liste gespeichert werden.

Prozeß- und Datenabstraktion

Information Hiding

- **Prinzip:**
  - Es werden nur die Informationen zur Verfügung gestellt, die ***absolut notwendig*** sind!
  - Alle anderen, insbesondere die ***wichtigen Informationen*** werden ***versteckt***!
  - Der Zugriff auf diese Informationen geschieht über "***Vermittler***".
- **Beispiel: "Offene Bank"**
  - Funktioniert nicht, weil
    - die Buchhaltung nicht klappt (Änderungen werden nicht jedem bekannt sein)
    - doch gestohlen wird (was mißbraucht werden kann, wird mißbraucht)
- **Deshalb:**
  - Das wertvolle wird versteckt (Geld -Tresor)
  - Es werden Vermittler eingesetzt (Mitarbeiter, Automaten)

D. Parnas, 1972

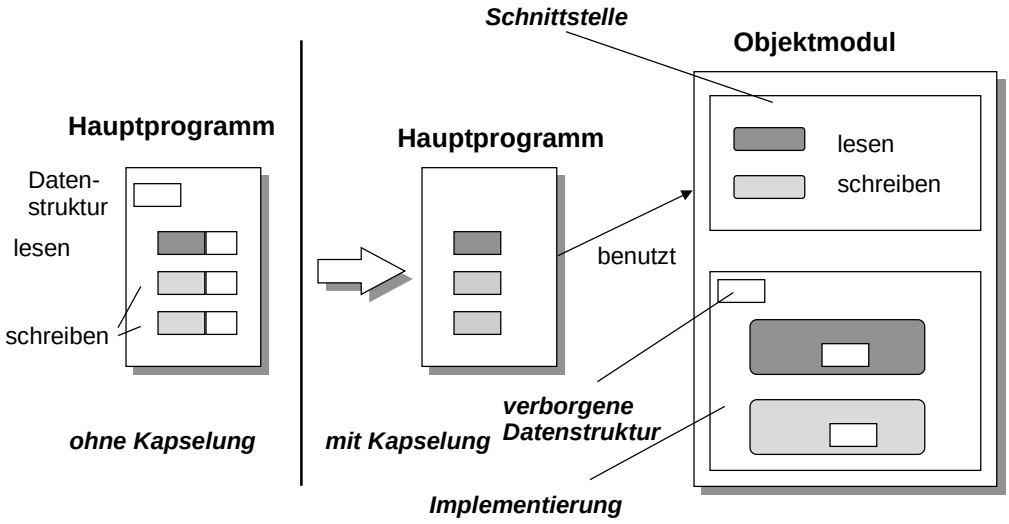
Objekt-modul

Datenkapsel

- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Datenstruktur von ihren sichtbaren Eigenschaften.
- **Merkmale:**
  - Eine Datenstruktur wird in einem Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen sichtbar***, die den allgemeinen Umgang mit der Datenstruktur beschreiben.
  - Die Datenstruktur selbst ist ***verborgen***.
- **Jedes Objektmodul**
  - beschreibt und realisiert nur eine ***einzige sog. abstrakte Datenstruktur***.
- **Kapselung von Daten**
  - ist ein zentraler Denkansatz und ein wesentliches ***Entwurfsprinzip***

Objekt-modul

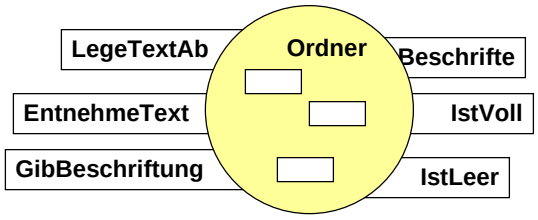
Schema: Objektmodul



Objekt-modul

Beispiel: Das Objektmodul Ordner

- **"Ordner " als Konzept**
  - ***enthält*** Texte
  - Texte können ***abgelegt*** und ***entnommen*** werden
  - ein Ordner kann ***beschriftet*** werden
  - ein Ordner kann ***leer*** oder ***voll*** sein
- **Ein Objektmodul**
  - realisiert ein ***fachliches Konzept*** (z.B. das Konzept "Ordner")
  - als ***genau eine abstrakte Datenstruktur***
  - hat ***Zustand*** (Gedächtnis)



Objekt-modul

Realisierung Objektmodul

```
INTERFACE Ordner;

PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();

END Ordner.
```

Objektmodul Ordner ist ein Behälter für Daten des Typs TEXT

Abstrakte Beschreibung des fachlichen Konzepts Ordner

Das eine Objekt Ordner soll mehrfach pro Programmablauf verwendbar sein

Objekt-modul

Verwendung eines Objektmoduls 1

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Ordner,  SIO;

BEGIN
  Ordner. Initialisiere();
  Ordner.Beschrifte ("Kleine Gedichte");
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
  Ordner.LegeTextAb ("Herr von Ribeck auf Ribeck ...");
  Ordner.LegeTextAb ("Von drauss vom Walde komm ich her ...");
  SIO.PutLine (Ordner.GibBeschriftung());
  SIO.PutLine ("--------------------------");
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  SIO.PutLine (Ordner.EntnehmeText());
  Ordner.Initialisiere();
  ...
END Ordner_Test.
```

Diskussion: Wie darf ein Objekt-modul verwendet werden?

Objekt-modul

Verwendung eines Objektmoduls 2

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - Ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in bestimmten Situationen (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, werden an der Schnittstelle entsprechende ***Testfunktionen*** wie `IstLeer` oder `IstVoll` zur Verfügung.

Objekt-modul

Einsatz der Testfunktionen

Prüfen der Vorbedingung

Das Objektmodul **Ordner** wird vor der ersten Verwendung initialisiert, d.h. der Inhalt wird gelöscht

```
Ordner.Initialisiere();

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Vor Entnahme wird der Zustand des Objektmoduls Ordner geprüft und entsprechend gehandelt.

Objekt-modul

Entwurfskonzept

- **Um ein fachliches Konzept als Objektmodul zu realisieren, stellen wir *drei Arten von Operationen* zur Verfügung.**
- **Prozeduren:**
  - ***verändern*** den ***Zustand*** des Objekts. Meist sind sie von Vorbedingungen abhängig, d.h. sie können nicht in jedem Zustand ausgeführt werden.
- **Funktionen:**
  - liefern Informationen, ***ohne*** den Objektzustand nach außen sichtbar ***zu verändern***.
  - Fachliche Funktionen:
    - liefern ***fachliche Informationen*** und sind vom Objektzustand abhängig.
  - Testfunktionen:
    - ***prüfen*** den Objektzustand und werden zum Prüfen der Vorbedingungen verwendet.

Objekt-modul

Entwurfskonzept-Beispiel: Ordner

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Prozeduren:**
  - `Initialisiere`, `LegeTextAb`, `EntnehmeText`, `Beschrifte` sind verändernde Prozeduren.
- **Fachliche Funktionen:**
  - `GibBeschriftung` ist eine fachliche Funktion; sie verändert im Gegensatz zu `EntnehmeText` nicht den Ordnerzustand.
- **Textfunktionen:**
  - `IstLeer` und `IstVoll`

Objekt-modul

Ein Blick ins Innere - 1

```
MODULE Ordner EXPORTS Ordner;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
TYPE Texte = ARRAY Fassungsvermoegen OF TEXT;
VAR ordnerInhalt : Texte;
    anzahlTexte : CARDINAL;
    beschriftung : TEXT := "";

PROCEDURE LegeTextAb (t : TEXT)=
BEGIN
  anzahlTexte := anzahlTexte + 1;
  ordnerInhalt[anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText () : TEXT =
VAR t : TEXT;
BEGIN
  t := ordnerInhalt[anzahlTexte];
  ordnerInhalt[anzahlTexte] := "";
  anzahlTexte := anzahlTexte - 1;
  RETURN t;
END EntnehmeText;
```

Es wird ein Array verwendet

Es wird der zuletzt abgelegte Text entnommen

Objekt-modul

Ein Blick ins Innere - 2

```
PROCEDURE IstVoll () : BOOLEAN =
BEGIN
  RETURN (anzahlTexte = MaxTexte);
END IstVoll
...

PROCEDURE Beschrifte (t : TEXT)=
BEGIN
  beschriftung := t;
END Beschrifte;
...

PROCEDURE Initialisiere () =
BEGIN
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    ordnerInhalt[i] := "";
  END;
  anzahlTexte := 0;
END Initialisiere;

BEGIN
  Initialisiere ();
END Ordner.
```

`IstVoll` verwendet die Variable `anzahlTexte`

Objekt-modul

Zusammenfassung Objektmodul

- **Eigenschaften eines Objektmoduls:**
  - Das Modul verwaltet seine Daten ***selbst***.
  - Die interne Repräsentation der Daten ist vollständig ***verborgen***.
  - Die interne Repräsentation ist ***austauschbar***.
  - Zur Laufzeit existiert immer ***nur ein Exemplar*** eines Objektmoduls, d.h. es können z.B. nicht mehrere Ordner erzeugt werden.
  - Das Objektmodul kann von ***mehreren*** anderen "Kunden" verwendet werden.
  - Dadurch kann z.B. ein ***gemeinsamer*** Ordner verwaltet werden.

Abstrakte Datentypen

Module als Sprachelement

- **Ein Modul kann zwar**
  - ***definiert*** und zur Laufzeit von anderen Modulen ***importiert*** werden,
  - aber es ist kein ***primäres*** Sprachelement (first class), d.h.
  - ein Modul kann ***nicht*** wie ein Typ ***zur Deklaration*** von Bezeichnern verwendet werden.

```
o1, o2 : Ordner (* geht nicht, da Ordner Modul ist *)
```

- **Folge:**
  - es gibt ***nur ein Exemplar*** eines Objektmoduls.
- **Problem:**
  - Es werden oft mehrere Exemplare eines durch ein Objektmodul realisierten Objekts gebraucht.
- **Lösungsansatz:**
  - Wir formulieren ***einen Typ*** für die im Modul beschriebenen Objekte.

Abstrakte Datentypen

Konzept Abstrakter Datentyp

- **Kann als *Verallgemeinerung* des Objektmoduls (Datenkapsel) betrachtet werden.**
- **Anstatt eines Objektes wird ein Typ (für diese Objekte) definiert.**
- **Betrachten wir den ADT als (formale) Spezifikation eines Typs,**
  - dann entwerfen wir ihn durch Angabe von
  - ***Typnamen***
  - ***Signaturen*** (Operationen)
    - zum Erzeugen von Objekten, zum Verändern etc.
  - ***Axiome***
    - formulieren den semantischen Zusammenhang der Operationen.
  - ***Vorbedingungen***
    - geben an, in welchem Zustand welche Operationen gültig sind.

Abstrakte Datentypen

Beispiel 1: ADT Bool

```
TYPE BOOL

FUNCTIONS

  true:  -> BOOL
  false: -> BOOL
  not:   BOOL -> BOOL
  and:   BOOL x BOOL -> BOOL
  or:    BOOL x BOOL -> BOOL

AXIOMS

  not(true) = false
  not(false) = true

  For any x: BOOL

    not(not(x)) = x

    or(true, x) = true     and(false, x) = false
    or(x, true) = true     and(x, false) = false
    or(false, x) = x       and(true, x)  = x
    or(x, false) = x       and(x, true)  = x
```


Abstrakte Datentypen

Beispiel 2: ADT NAT

```
TYPE   NAT

FUNCTIONS

  zero: -> NAT
  succ: NAT -> NAT
  iszero: NAT -> BOOL
  pred: NAT -> NAT

  add, mult, sub : NAT x NAT -> NAT

AXIOMS                                  PRECONDITIONS

  For any x, y: NAT                     pred(x: NAT)
                                        requires iszero(x) = false

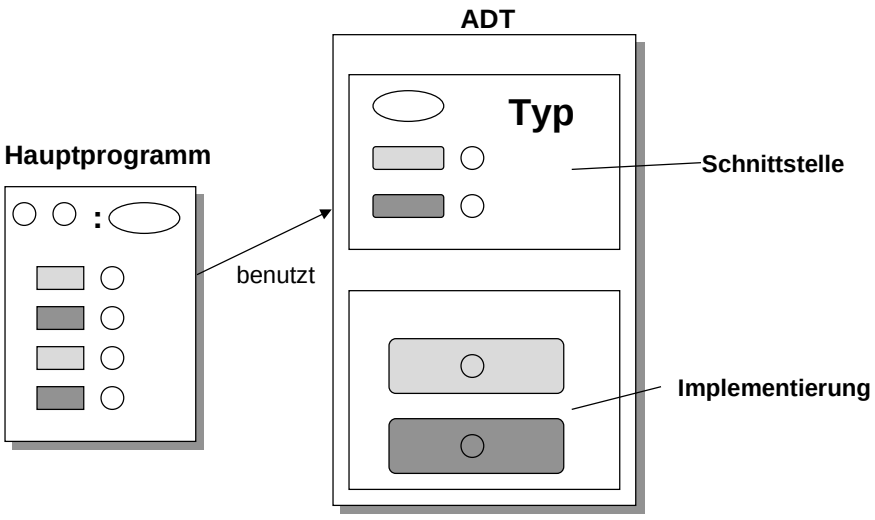
    pred(succ(x))    = x
    iszero(zero)     = true
    iszero(succ(x))  = false

    add(zero, x)     = x
    add(succ(x), y)  = succ(add(x, y))
    sub(x, zero)     = x
    sub(x, succ(y))  = pred(sub(x, y))
    mult(x, zero)    = zero
    mult(x, succ(y)) = add(mult(x, y), x)
```

Realisierung von ADTs durch Module

- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Menge gleichartiger Datenstrukturen von ihren allgemeinen Eigenschaften.
- **Merkmale:**
  - Die Beschreibung der gleichartigen Exemplare einer Datenstruktur wird in einem sog. ADT-Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen*** sichtbar, die den allgemeinen Umgang mit ***jedem Exemplar*** der Datenstruktur beschreiben.
  - Ein ***Bezeichner für den Typ*** der Datenstruktur wird exportiert.
  - Die Implementierung der Datenstruktur selbst ist ***verborgen***.
- **Jedes ADT-Modul beschreibt und realisiert eine Menge von Exemplaren der abstrakten Datenstruktur.**
  - Die Exemplare werden von ***ihren Kunden*** verwaltet. Ihre Bearbeitung geschieht nur mit Hilfe der exportierten Operationen des ADT-Moduls.

Realisierungsschema ADT



Abstrakte Datentypen

Realisierung eines ADT in Modula-3

- **Forderung:**
  - In der Schnittstelle darf die Typstruktur eines ADT ***nicht sichtbar*** sein.
  - Lediglich der ***Name*** soll bekannt gemacht werden.
- **Realisierung im Modula-3**
  - In der Schnittstelle wird ein ***opakter Typ*** (verdeckter Typ) deklariert
  - Dies geschieht, indem der Typ als ***Untertyp*** zum vordefinierten Typ REFANY deklariert wird.
  - ***Obertyp*** eines opakten Typs muß ein ***Referenztyp*** sein.
  - In der Implementierung des ADTs wird der opake Typ ***offengelegt*** ("enthüllt")
  - Bsp.:

```
TYPE Ordner <: REFANY;
```

Mithilfe des Subtyp-Konzepts und mit opaken Typen können die Forderungen umgesetzt werden

Abstrakte Datentypen

Schnittstelle des ADT-Moduls Ordner

Nur der Name des ADT ist sichtbar!

Alle Operationen erhalten als ersten Parameter das jeweilige Ordnerobjekt

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

Erzeugt ein neues Ordnerobjekt und gibt es zurück

Abstrakte Datentypen

Beispiel: ADT Ordner

Angabe des Typnamens

Angabe der Operationen

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
PROCEDURE Beschrifte (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung ( o: Ordner; ) : TEXT;
PROCEDURE Anlegen ( ) : Ordner;

END OrdnerADT.
```

```
IMPORT OrdnerADT; (* Client *)
VAR
  ord1, ord2 : OrdnerADT.Ordner

BEGIN
  ord1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ord1, "Nicht immer sind bequeme ...");

  ord2 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ord2, "Musikstuecke");
  OrdnerADT.LegeTextAb (ord2, "Tief im Westen ...");
```


Abstrakte Datentypen

Implementierung des ADT Ordner

```
MODULE OrdnerADT EXPORTS OrdnerADT;

CONST MaxTexte = 20;
TYPE Fassungsvermoegen = [1 .. MaxTexte];
     Texte = ARRAY Fassungsvermoegen OF TEXT;

REVEAL Ordner = BRANDED REF RECORD
                  ordnerInhalt : Texte;
                  anzahlTexte  : Fassungsvermoegen;
                  beschriftung : TEXT := "";
                END;
```

- **REVEAL-Deklaration**
  - damit wird die ***interne Struktur*** des opakenTyps bekannt gegeben
  - Steht immer in der ***Implementierung*** eines ADTs (information hiding).
  - Der äußere Typ-Konstruktor ***muß*** ein mit einem ***Brandzeichen*** versehener Referenztyp sein
  - Dieses unterscheidet den Typ von anderen ***strukturell gleichen*** Typen.

Abstrakte Datentypen

Ein Blick ins Innere des ADTs

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)=
BEGIN
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT =
VAR t : TEXT;
BEGIN
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  RETURN t;
END EntnehmeText;


PROCEDURE Anlegen () : Ordner =
VAR o : Ordner;
BEGIN
  o := NEW(Ordner);
  FOR i := FIRST(Fassungsvermoegen) TO LAST(Fassungsvermoegen) DO
    o^.ordnerInhalt[i] := "";
  END;
  o^.anzahlTexte := 0;
  RETURN o;
END Anlegen;
```


Abstrakte Datentypen

Diskussion : ADT-Realisierung in Modula-3

- **Wir können feststellen**
  - Als Typ für einen ADT müssen wir einen opaken Referenztyp wählen.
  - Grund: So kann der Übersetzer für Objekte eines ADTs ausreichend Speicherplatz reservieren, ohne den inneren Aufbau des Typs zu kennen.
- **Konsequenz:**
  - Es können neben den Operationen der Schnittstelle des ADTs auch die Operationen
    - Zuweisung und
    - Vergleich auf Objekten des ADTs durchgeführt werden (Pointer-Zuweisung und Pointer-Vergleich).
  - Diese sollten jedoch nicht genutzt werden!
  - Das Brandzeichen verhindert, daß einem Objekt eines ADT zufälligerweise ein Wert eines strukturell gleichen Typs zugewiesen werden kann.

Abstrakte Datentypen

Zuweisung und Vergleich von ADT-Objekten

```
ordner1 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner1, "Nicht immer sind bequeme Stuehle ...");

ordner2 := OrdnerADT.Anlegen();
OrdnerADT.Beschrifte (ordner2, "Kleine Gedichte");
OrdnerADT.LegeTextAb (ordner2, "Nicht immer sind bequeme Stuehle ...");

IF ordner1 = ordner2 THEN
  SIO.PutLine ("1 ordner sind gleich");
ELSE
  ordner2 := ordner1;
  OrdnerADT.Beschrifte (ordner1, "Romane");
END;
IF ordner1 = ordner2 THEN
  SIO.PutLine ("nach Zuweisung sind die Ordner gleich");
END;

SIO.PutLine (OrdnerADT.GibBeschriftung(ordner1));
SIO.PutLine ("--------------------------");
SIO.PutLine (OrdnerADT.GibBeschriftung(ordner2));
```

Vergleich der Referenzen (Adressen

ordner2 zeigt auf die selbe Adresse wie ordner1

Ändert ordner1 und ordner2

Abstrakte Datentypen

Modul als Entwicklungseinheit

- **Ein Modul ist charakterisiert durch eine *Entwurfs-entscheidung*.**
  - (z.B. wir wollen Ordner verarbeiten können)
- **Jedes Modul basiert auf genau einer Entwurfsentscheidung.**
- **Module *verbergen* Implementierungsentscheidungen.**
  - Jedes Modul hat sein Geheimnis.
- **Es werden immer die Entscheidungen**
  - in einem Modul gekapselt, die vielleicht ***revidiert*** werden müssen als andere.
  - z.B. Datenaufbau
- **Der Änderungsaufwand muß möglichst immer auf ein Modul begrenzt werden.**

Abstrakte Datentypen

Zusammenfassung ADT

- **In imperativen Sprachen**
  - mit einem ***Modulkonzept***
  - realisieren wir abstrakte Datentypen mit einem ADT-Modul.
- **Ein ADT-Modul zeigt folgende Eigenschaften:**
  - Das Modul liefert ***Exemplare einer Datenstruktur***, die beim Kunden aufbewahrt werden.
  - Dadurch können ***mehrere Exemplare*** eines ADT-Moduls bearbeitet werden.
  - Die Datenstruktur ist nur als ***Verweis*** auf die interne Repräsentation bekannt.
  - Die Repräsentation selbst ist ***verborgen*** und ***austauschbar***.
  - Die Datenstrukturen des ADT-Moduls können (fast) nur über die ***exportierten Operationen*** des Moduls bearbeitet werden.

Was haben wir gelernt?

- **Modularisierung**
  - Mittel, um ***handhabbare*** Programmeinheiten zu konstruieren
  - Module bestehen aus ***Schnittstelle*** und ***Implementierung***
- **Information Hiding**
  - Ziel:
    - Implementierung wesentlicher Details ist ***nicht*** nach ***außen sichtbar,*** insbesondere anwendbar auf Datenstrukturierung
  - Objektmodul:
    - Es wird in einem Modul ***ein Objekt*** gemäß Information Hiding realisiert, Datenabstraktion für ein Objekt
  - Abstrakter Datentyp:
    - Es wird ein ***geschützter Typ*** realisiert. Objekte des ADTs können nur mit den Operationen des ADTs manipuliert werden, Datenabstraktion für eine „Klasse“ von Objekten

Glossar

- **Information Hiding: Prozeß- oder funktionale Abstraktion, Datenabstraktion, funktionaler Module, Datenobjektmodul (Kapsel), abstrakter Datentyp**
- **opake Datenstruktur, opake (geschützte, abstrakte, geheime) Datentypen**
- **Schnittstellenoperationen eines Datenabstraktionsbausteins: Initialisierung (Löschung), Lesen, Verändern, Sicherheitsoperationen**
- **Realisierung opaker Datentypen durch Verweistypen**
- **Typname, Signatur, Axiome, Vorbedingungen eines ADT**
- **Subtypen in Modula-3 haben Referenzsemantik (keine Zuweisung und keine Gleichheitsabfrage nutzen!)**

Vertragsmodell

- **Konzept des Vertragsmodells**
- **Zusicherungen**
- **Realisierung von Zusicherungen mit Pragmas**
- **Exkurs Ausnahmebehandlung**
- **Zusicherungen mittels Ausnahmebehandlung**

Erinnerung

```
INTERFACE Ordner;
PROCEDURE LegeTextAb (t : TEXT);
PROCEDURE EntnehmeText () : TEXT ;
PRODEDURE IstVoll () : BOOLEAN ;
PROCEDURE IstLeer () : BOOLEAN ;
PROCEDURE Beschrifte (t : TEXT);
PROCEDURE GibBeschriftung () : TEXT;
PROCEDURE Initialisiere ();
```

- **Feststellung:**
  - `LegeTextAb` und `Entnehme` sind ***nicht in jedem Zustand*** des Ordners sinnvoll:
    - ein voller Ordner kann keine weiteren Texte aufnehmen; ein leerer keine herausgeben.
  - Solche Operationen sind nur in ***bestimmten Situationen*** (abhängig von bestimmten Bedingungen) sinnvoll.
  - Um den sicheren Umgang mit einem solchen Objekt zu gewährleisten, stellen wir an der Schnittstelle entsprechende ***Testfunktionen*** (Sicherheitsabfragen) wie `IstLeer` oder `IstVoll` zur Verfügung.

Konzept des Vertragsmodells

Einsatz der Testfunktionen

```
Ordner.Initialisiere;

IF Ordner.IstVoll() THEN
  SIO.PutLine ("Ordner ist bereits voll");
ELSE
  Ordner.LegeTextAb ("Nicht immer sind bequeme Stuehle ...");
END;

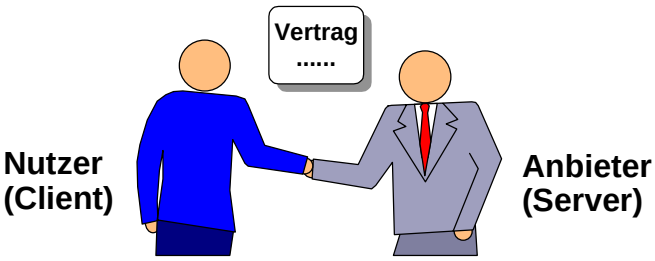
IF Ordner.IstLeer() THEN
  SIO.PutLine ("Ordner ist leer");
ELSE
  t := Ordner.EntnehmeText();
END;
```

Prüfen, ob die Operation angewendet werden darf.

- **Frage:**
  - Wer soll ***sicherstellen***, dass eine Operation ausgeführt werden darf?
  - ***Nutzer*** oder ***Anbieter***? oder
  - zur Sicherheit auf ***beiden Seiten***?

Konzept des Vertragsmodells

Idee des Vertragsmodells



- **Vertrag**
  - zwischen Nutzer und Anbieter einer Operation regelt, wer der beiden Partner welche ***Verpflichtungen*** einhalten muss (und welchen ***Nutzen*** er dadurch hat)
- **Operation**
  - arbeitet korrekt, wenn sie vertragsgemäß ***benutzt*** bzw. ***realisiert*** wird.
- **Frage**
  - Wie kann ein solcher Vertrag ***programmtechnisch*** realisiert werden?

Zusicherungen

Erstes Beispiel

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  require NOT IstVoll(o)
  ensure  NOT IstLeer(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  require NOT IstLeer(o)
  ensure  NOT IstVoll(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .

END OrdnerADT.
```

Vorbedingung

Nachbedingung

Zusicherungen

Arten und Nutzung

- **Zusicherungen**
  - sind eine Technik, um eine bestimmte Art von Verträgen zwischen Anbieter und Nutzer zu formulieren.
- **Zusicherungen werden formuliert als**
  - ***Vorbedingungen*** für Operationen
  - ***Nachbedingungen*** von Operationen
  - ***Invarianten*** von abstrakten Datentypen
- **Zusicherungen**
  - erhöhen die ***Benutzbarkeit***, indem sie diese formal definieren
  - verbessern die ***Testbarkeit***
  - verbessern die ***Fehlersuche*** (debugging)
  - verlangen vom Entwickler ***abstraktes*** Denken

Zusicherungen

Vor- und Nachbedingungen

- **Vorbedingung**
  - beschreibt eine Bedingung, die der ***Nutzer*** (Aufrufer) einer Operation ***einhalten*** muß, damit die Operation korrekt arbeitet
- **Nachbedingung**
  - beschreibt einen Zustand, der nach dem erfolgreichen Ausführen der Operation vorhanden ist
  - diese garantiert der ***Anbieter***

Die Technik der Zusicherungen sollte insb. bei der Entwicklung von Programmkomponenten (Modulen) eingesetzt werden, die wiederverwendet werden sollen!

Zusicherungen

Verpflichtung <-> Nutzen

- **Vorbedingungen sind**
  - Verpflichtungen für den Benutzer
  - Nutzen für den Anbieter
- **Nachbedingungen sind**
  - Nutzen für den Benutzer
  - Verpflichtungen für den Anbieter

Operation `EntnehmeText`

	Verpflichtungen	**Nutzen**
Nutzer	Der Ordner darf nicht leer sein.	Der zuletzt eingegebene Text wird entnommen und zurückgegeben.
Anbieter	Verändere den Ordner so, daß der zuletzt eingegebene Text entnommen wird.	Ist sicher, daß es noch einen Text im Ordner gibt

Zusicherungen

Invariante

- **Invariante beschreibt Bedingungen,**
  - die erfüllt sind, wenn Objekte eines ADTs ***erzeugt*** werden
  - die während der ***gesamten*** Lebenszeit der Objekte gelten
  - d.h. von allen Operationen ***nicht verletzt*** werden

- **Beispiel: ADT Ordner**
  - zu jedem Zeitpunkt im "Leben" eines Ordnerobjekts gilt:
  - **`anzahlTexte >= 0 AND anzahlTexte <= MaxTexte`**
  - ADT Ordner wird um eine interne Funktion erweitert, die die Invariante prüft.

```
PROCEDURE Invariante (o : Ordner): BOOLEAN =
BEGIN
  RETURN (o^.anzahlTexte >= 0 AND o^.anzahlTexte <= MaxTexte);
END Invariante;
```


Zusicherungen

ADT Ordner mit Invariante

```
INTERFACE OrdnerADT;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT);
  require NOT IstVoll(o)
  ensure  NOT IstLeer(o)
  ensure  Invariante(o)

PROCEDURE EntnehmeText (VAR o: Ordner; ) : TEXT ;
  require NOT IstLeer(o)
  ensure  NOT IstVoll(o)
  ensure  Invariante(o)

PROCEDURE IstVoll ( o: Ordner; ) : BOOLEAN ;
PROCEDURE IstLeer ( o: Ordner; ) : BOOLEAN ;
. . .
END OrdnerADT.
```

Invariante

Realisierung mit Pragmas

Pragmas in M3

- **Pragmas in Modula-3**
  - Pragmas sind Anweisungen an den ***Übersetzer***
  - Sie ändern die ***Semantik*** des Programmes nicht
  - Die Implementierung eines Pragmas ist ***übersetzerspezifisch***
  - Pragmas können ***irgendwo*** im Programmtext auftreten
  - Pragmas müssen einer vordefinierte Syntax entsprechen
    - *<* Pragmaname Parameter *>*
- **Das Pragma ASSERT**
  - `<* ASSERT AUSDRUCK *>`
  - Der AUSDRUCK muß einen Wert vom Typ BOOLEAN liefern
  - Der Ausdruck wird zur ***Laufzeit*** ausgewertet
  - Ist das Ergebnis des Ausdrucks FALSE
    - wird ein ***Laufzeitfehler*** ausgelöst
    - und das Programm bricht ab!

Realisierung mit Pragmas

Zusicherungen mit Pragma ASSERT

```
PROCEDURE EntnehmeText (VAR o: Ordner): TEXT =
VAR t : TEXT;
BEGIN
  <* ASSERT NOT IstLeer(o) *>

  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;

 <* ASSERT NOT IstVoll(o)  *>
 <* ASSERT Invariante(o)   *>

  RETURN t;
END EntnehmeText;
```

```
VAR ordner1, ordner2 : OrdnerADT.Ordner; t : TEXT;

BEGIN
  ordner1 := OrdnerADT.Anlegen();
  OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
  OrdnerADT.LegeTextAb (ordner1, "Nicht immer...");
  t := OrdnerADT.EntnehmeText(ordner1);
  t := OrdnerADT.EntnehmeText(ordner1);
```

Laufzeitfehler

Realisierung mit Pragmas

Diskussion: Einsatz von ASSERT

- **ASSERT**
  - bietet eine einfache Möglichkeit, Zusicherungen zu implementieren.
- **Nachteile**
  - "brutale" Implementierung: Programmabbruch
  - es wird keine Information über die Verletzung des Vertrages geliefert
  - es gibt keine Möglichkeit, auf die Verletzung des Vertrages zu reagieren
- **Bessere Lösung**
  - Entwerfen eines Moduls zur Prüfung von Zusicherungen

```
PROCEDURE Require (expr : BOOLEAN);
PROCEDURE Ensure  (expr : BOOLEAN);
```

- **Frage**
  - Was soll geschehen, wenn eine Zusicherung verletzt wird?
  - Verletzung: Ausnahmesituation!

Exkurs Ausnahme-behandlung

Ausnahmen: Bsp. und Def.

- **Beispiel für Ausnahmesituationen**
  - Während des Schreibens einer Datei auf Diskette wird die Diskette entfernt.
  - Kein Plattenplatz mehr verfügbar.
  - Berechnung eines Addition ist größer als LAST(INTEGER).
  - Falsche Daten werden von einer Datei eingelesen.
- **Definition: Ausnahme**
  - IEEE Glossary: "An event that causes suspension of normal program execution. Types include addressing exception, data exception, operation exception, overflow exception, protection exception, underflow exception."
  - Ausnahmen sind Programmzustände, die ***nicht im normalen*** Programmablauf vorgesehen sind.

Exkurs Ausnahme-behandlung

Merkmale von Ausnahmen

- **Merkmale**
  - Ausnahmen entstehen zur ***Laufzeit.***
  - Einige Programmiersprachen (Java, Ada, Modula-3) erlauben, benutzerdefinierte ***Ausnahmen*** zu deklarieren und ***Ausnahmebehandlung*** durchzuführen.
  - Beispiel: vorgegebene Ausnahme
    - *SIO.Error*
      Wird von den IO-Modulen erweckt, wenn Datei-Operationen nicht wie intendiert durchgeführt werden können.
- **Deklaration benutzerdefinierter Ausnahmen**
  - EXCEPTION <name>;
  - Wird eine Ausnahme von einer Schnittstelle exportiert, können auch die Klienten diese Ausnahme generieren.
- **Erwecken einer Ausnahme**
  - RAISE-Anweisung

Exkurs Ausnahme-behandlung

Ausnahme und Ausnahmebehandlung

```
MODULE Ausnahme EXPORTS Main;
IMPORT SIO;
VAR eingabe : INTEGER;
BEGIN
  LOOP
    TRY
      SIO.PutLine ("Geben Sie bitte eine ganze Zahl ein:");
      eingabe := SIO.GetInt();
      EXIT;
    EXCEPT
      SIO.Error => SIO.PutLine ("*** Eingabeformat falsch");
                   line := SIO.GetLine();
    END;
  END;
  . . .
END Ausnahme.
```

Schützt Anweisungen, bzgl. dem Auftreten von Ausnahmen

Ausnahmebehandlung "exception handler"

Exkurs Ausnahme-behandlung

Weiterleiten von Ausnahmen

- **Idee:**
  - Wenn in einer Prozedur eine Ausnahme nicht behandelt werden soll, dann kann diese Prozedur die Ausnahme an die ***sie rufende Prozedur*** weiterleiten.
  - So können Ausnahmen über ***mehrere Stufen*** weitergeleitet und an der entsprechenden Stelle behandelt werden.
- **Weiterleitung von Ausnahmen**
  - muss bei der Prozedur-Deklaration angegeben werden
  - PROCEDURE <name> <signature> RAISES {exc1, .. excN}
- **Beispiele:**
  - viele Operationen des Moduls `SIO` leiten die Ausnahme `Error` weiter
  - `PROCEDURE GetChar(rd: Reader := NIL): CHAR RAISES {Error};`

Exkurs Ausnahme-behandlung

Kontrollfluß bei Ausnahmen

- Eine Ausnahme tritt in TRY-EXCEPT-Anweisung auf und die Ausnahme wird dort behandelt, dann
  - werden die in dem Ausnahmebehandler für die Ausnahme stehenden Anweisungen durchgeführt,
  - Das Programm wird anschließend nach dem END der TRY-EXCEPT-Anweisung ***fortgeführt***.
- Eine Ausnahme tritt in einem ***ungeschützten*** Bereich einer Prozedur auf und die Ausnahme ist Element der RAISES-Liste der Prozedur, dann,
  - wird die Prozedur abgebrochen und die Ausnahme an die die Prozedur rufende Prozedur ***weitergeleitet***
- Kann eine Ausnahme weder behandelt noch weitergeleitet werden, dann,
  - wird das Programm mit einem Laufzeitfehler ***abgebrochen***

Zusicherungen mittels Ausnahmebeh.

Schnittstelle des Moduls `Assertion`

```
INTERFACE Assertion;

EXCEPTION Violated;
          Terminate;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated};

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated};

PROCEDURE EnableAssertions();
PROCEDURE DisableAssertions();

END Assertion.
```

Ausnahme `Violated` wird generiert, wenn eine Zusicherung verletzt wird

Prüfung der Zusicherungen kann unterdrückt werden

Zusicherungen mittels Ausnahmebeh.

Implementierung `Assertion` - 1

```
MODULE Assertion;
IMPORT SIO;

VAR enabled : BOOLEAN;

PROCEDURE Require (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Precondition violated: " & procName);
    RAISE Violated;
  END;
END Require;

PROCEDURE Ensure  (expr : BOOLEAN; procName: TEXT)
                   RAISES {Violated} =
BEGIN
  IF enabled AND NOT expr THEN
    SIO.PutLine("*** Postcondition violated: " & procName);
    RAISE Violated;
  END;
END Ensure;
```

Geschützte Variable

Zusicherungen mittels Ausnahmebeh.

Implementierung Assertion - 2

```
PROCEDURE EnableAssertions()=
BEGIN
  enabled := TRUE;
END EnableAssertions;

PROCEDURE DisableAssertions()=
BEGIN
  enabled := FALSE;
END DisableAssertions;


BEGIN
  EnableAssertions();
END Assertion.
```


Zusicherungen mittels Ausnahmebeh.

Ordner-Operationen mit Assertion - 1

```
INTERFACE OrdnerADT;

IMPORT Assertion AS As;

TYPE Ordner <: REFANY;

PROCEDURE LegeTextAb     (VAR o: Ordner; t : TEXT)
                          RAISES {As.Violated};
PROCEDURE EntnehmeText   (VAR o: Ordner): TEXT
                          RAISES {As.Violated};
PROCEDURE IstVoll        (o: Ordner): BOOLEAN;
PROCEDURE IstLeer        (o: Ordner): BOOLEAN;
PROCEDURE Beschrifte     (VAR o: Ordner; t : TEXT);
PROCEDURE GibBeschriftung(o: Ordner): TEXT;
PROCEDURE Anlegen        ( ) : Ordner RAISES {As. Violated};

END OrdnerADT.
```

Zusicherungen mittels Ausnahmebeh.

Ordner-Operationen mit Assertion - 2

```
PROCEDURE LegeTextAb (VAR o: Ordner; t : TEXT)
                     RAISES {As. Violated} =
BEGIN
  As.Require (NOT IstVoll(o), "OrdnerADT.LegeTextAb");
  o^.anzahlTexte := o^.anzahlTexte + 1;
  o^.ordnerInhalt[o^.anzahlTexte] := t;
  As.Ensure (NOT IstLeer(o), "OrdnerADT.LegeTextAb");
  As.Ensure (Invariante(o), "OrdnerADT.LegeTextAb");
END LegeTextAb;

PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT
                       RAISES {As. Violated} =
VAR t : TEXT;
BEGIN
  As.Require (NOT IstLeer(o), "OrdnerADT.EntnehmeText");
  t := o^.ordnerInhalt[o^.anzahlTexte];
  o^.ordnerInhalt[o^.anzahlTexte] := "";
  o^.anzahlTexte := o^.anzahlTexte - 1;
  As.Ensure (NOT IstVoll(o), "OrdnerADT.EntnehmeText");
  As.Ensure (Invariante(o), "OrdnerADT.EntnehmeText");
  RETURN t;
END EntnehmeText;
```


Zusicherungen mittels Ausnahmebeh.

Umgang mit den Ausnahmen - 1

- **Empfehlung**
  - Der aufrufende Block klammert alle Anweisungen in einer TRY-EXCEPT-Anweisung.
  - Im EXCEPT-Teil werden noch mögliche Abschluß-Operationen durchgeführt (z.B. Schließen von Dateien) und eine entsprechende Meldung ausgegeben und die Ausnahme `Terminate` generiert.

```
PROCEDURE ArbeiteMitOrdner() RAISES {As.Terminate} =
VAR ordner1 : OrdnerADT.Ordner;
BEGIN
  TRY
    ordner1 := OrdnerADT.Anlegen();
    OrdnerADT.Beschrifte (ordner1, "Kleine Gedichte");
    SIO.PutLine (OrdnerADT.EntnehmeText(ordner1));
    ...
  EXCEPT
    As.Violated =>
      SIO.PutLine ("*** Called from: ArbeiteMitOrdner");
      RAISE As.Terminate;
  END;
END ArbeiteMitOrdner.
```

Zusicherungen mittels Ausnahmebeh.

Umgang mit den Ausnahmen - 2

```
MODULE Ordner_Test EXPORTS Main;

IMPORT Assertion AS As;
IMPORT OrdnerADT,  SIO;

PROCEDURE ArbeiteMitOrdner()RAISES {As.Terminate}=
BEGIN
  ...
END ArbeiteMitOrdner;

BEGIN
  As.EnableAssertions();
  TRY
    ArbeiteMitOrdner();
  EXCEPT
    As.Terminate =>
      SIO.PutLine ("*** Program terminated ");
  END
END Ordner_Test.
```

```
*** Precondition violated: OrdnerADT.EntnehmeText
*** Called from: ArbeiteMitOrdner
*** Program terminated
```


Zusicherungen mittels Ausnahmebeh.

Arbeiten mit Zusicherungen - 1

- **Im Rumpf einer Operation darf die Vorbedingung nicht geprüft werden**
  - Widerspricht dem herkömmlichen ***defensiven*** Programmieren
- **Vorteil**
  - Schon mittelgroße Systeme enthalten ca. 10 -20% Code, um solche Eingangsprüfungen durchzuführen.
  - Dabei entstehen komplexe Prüfungen.
  - Prüfroutinen sind häufig Ursachen für Fehler.

```
PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
END Wurzel ;

PROCEDURE Wurzel (x: REAL): REAL is
  require X >= 0
BEGIN
  if x < 0 then
    “handle error”
  else
    RETURN (x * x);
  end
END Wurzel ;
```

Zusicherungen mittels Ausnahmebeh.

Arbeiten mit Zusicherungen - 2

- **Zusicherungen sollen nicht verwendet werden, um Spezialfälle zu behandeln**
  - Zusicherungen sind Aussagen über die ***korrekte*** Nutzung.
  - Sollen Spezialfälle behandelt werden, dann werden herkömmliche ***Kontrollanweisungen*** verwendet (IF oder CASE).
- **Beispiel**
  - Wenn `Wurzel` den Fall x < 0 als Spezialfall behandeln soll, dann ist das zu programmieren.
  - Wenn x >= 0 die Vorbedingung ist, dann ist ein Aufruf

    ```
    Wurzel (-1);
    ```

    nicht erlaubt, also eine ***nicht vertragskonforme Benutzung***!
- **Wird eine Zusicherung verletzt (zur Laufzeit)**
  - Vorbedingung: Nutzer hat den Vertrag verletzt
  - Nachbedingung: Anbieter hat den Vertrag verletzt

Was haben wir gelernt?

- **Vertragsmodell: Vertrag zwischen Nutzer (Client) und Anbieter (Server),**
- **Konsistenz durch Vor- und Nachbedingungen sowie Invarianten**
- **Pragmas, Pragma `Assert`**
- **Ausnahmen: Motivation, Ausnahmedeklaration, Erwecken einer Ausnahme, Behandlung durch Ausnahmebehandler, Weiterreichen einer Ausnahme**
- **Zusicherungen mit Ausnahmen, allgemeingültiges Ausnahmebehandlungsmodul `Assertion`, Verwendung bei Clienten und bei Server**

Glossar

- **Vertragsmodell, Vorbedingungen und Benutzerverpflichtung, Nachbedingungen und Anbieterverpflichtung**
- **Formalisierung von Zusicherungen, Vorbedingungen, Nachbedingungen, Invarianten**
- **ADT-Schnittstelle mit Zusicherung**
- **Pragmas in Modula-3, vordefiniertes Pragma `Assert`, Nutzung für Zusicherungsrealisierung**
- **Ausnahmesituationen, Definition Ausnahme, (vordefinierte und) benutzerdefinierte Ausnahme, Ausnahmedeklaration, Erwecken einer Ausnahme, Ausnahmebehandlung und TRY-EXCEPT-Anweisung**
- **Ausnahmebehandlung im Ausnahmebehandler, ungeschützte Ausnahmen und weiterreichen, falls gerufene Prozedur diese Ausnahme im Kopf angibt, Programmabbruch, falls weder behandelt noch weitergeleitet**
- **Ausnahmen für die Realisierung von Zusicherungen**

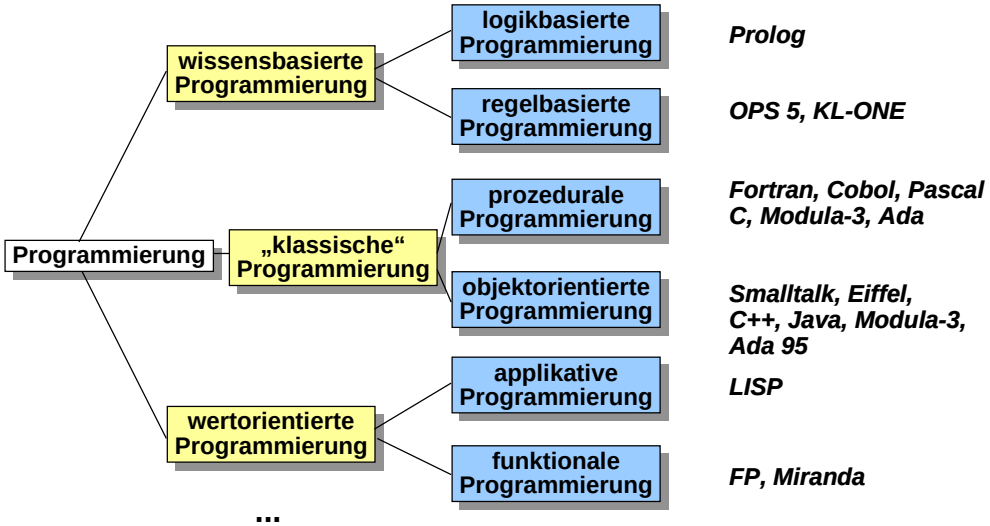
Objektorientierte Programmierung I

- Verständnis der objektorientierten Programmierung
- Objekte
- Klassen
- Vererbung
- Polymorphismus und dynamisches Binden
- Diskussion

Verständnis der OO Programmierung

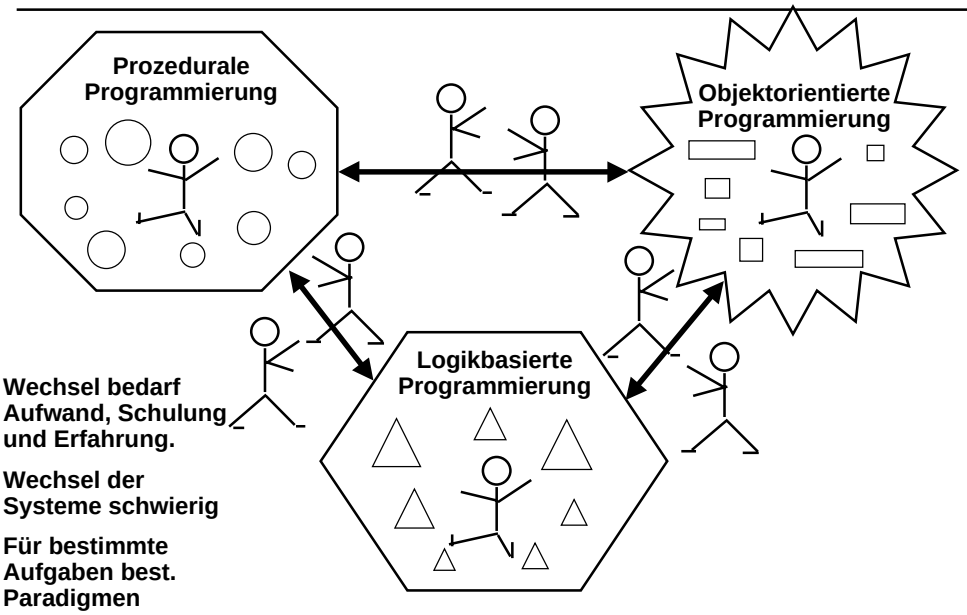
Programmier-Paradigmen

- Es gibt unterschiedliche Arten der Programmierung:



Verständnis der OO Programmierung

Wechsel des Programmier-Paradigmas



Wechsel bedarf Aufwand, Schulung und Erfahrung.

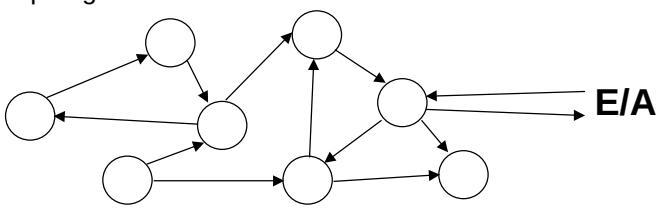
Wechsel der Systeme schwierig

Für bestimmte Aufgaben best. Paradigmen

Verständnis der OO Programmierung

Was ist ein rein objektorientiertes System

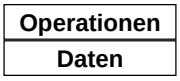
- **Ein Software-System besteht aus Objekten**
- **Jede Systemaktivität ist die Aktivität eines Objektes**
- **Objekte**
  - haben eine Lebensdauer (werden erzeugt und vernichtet)
  - sind identifizierbar
  - sind aktiv
  - verändern sich während ihrer Lebensdauer
  - senden und empfangen Nachrichten



Objekte

Objekte - die Dynamik des Systems

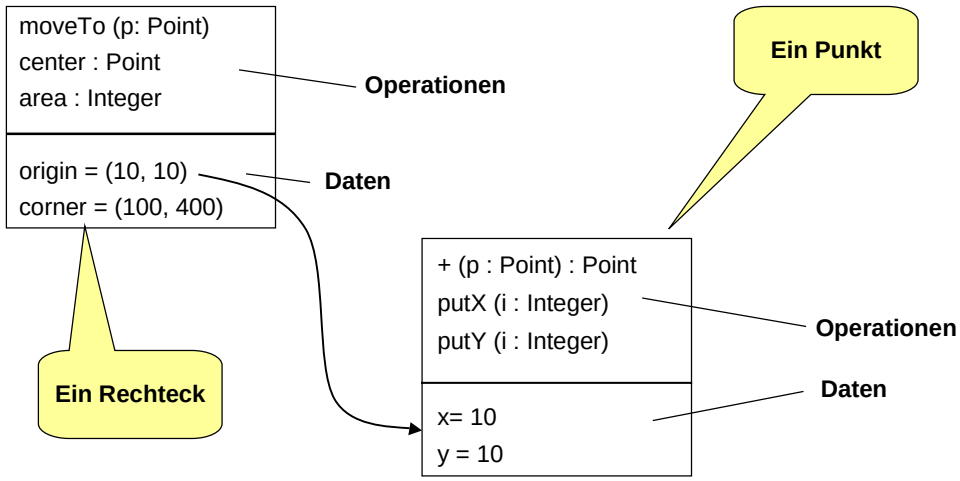
- **Ein Objekt ist eine Datenkapsel, die aus zwei Teilen besteht**
  - Der Wert der Daten repräsentiert den **Zustand** des Objekts
  - Daten können nur mithilfe von **Operationen** verändert werden (Kapselung)



- **Die Aktivität der Objekte ist die Ausführung ihrer Operationen**
  - Eine Operation wird ausgeführt, wenn ein Objekt eine entsprechende **Nachricht** erhält.
  - Der Objektzustand kann sich **verändern**.
  - Mögliche Operationen sind durch den **Objekttyp** bestimmt.
- **Ein Objekt ist ein Exemplar genau einer Klasse.**
- **Objekte existieren nur zur Laufzeit**
  - können aber auch persistent gespeichert werden

Objekte

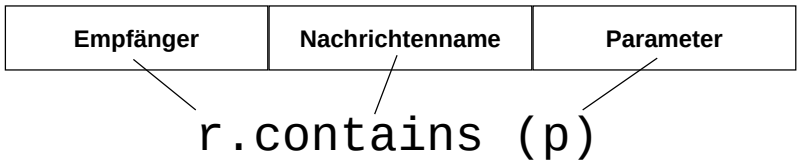
Beispiele für Objekte



Objekte

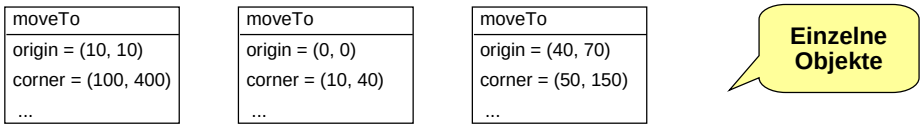
Nachrichten (Botschaften)

- **Objekte kommunizieren miteinander,**
  - dadurch daß sie **Nachrichten** versenden und Nachrichten empfangen.
- **Objekte reagieren auf Nachrichten,**
  - indem sie eine **Operation** (**Methode, Zugriffsunterprogramm**) ausführen.
- **Der Empfänger einer Nachricht (immer ein Objekt) ist verantwortlich für**
  - **Entschlüsselung** der Nachricht (verstehe ich die Nachricht?)
  - **Wirkung** (was tue ich?)



Klassen

Klassen - die Statik des Systems



- **Gleichartige Objekte werden zusammengefaßt**
  - und an einer Stelle beschrieben (Objekttyp oder Klasse)
- **Eine Klasse definiert für ihre Objekte**
  - die **Speicherstruktur**
    - Daten, Attribute, Exemplarvariablen
  - die **Operationen** (Methoden, Routinen),
    - von denen ein Teil exportiert wird (exported, public <-> private)
    - andere werden nur intern benötigt
- **Von einer Klasse können beliebig viele Objekte erzeugt werden**

Klassen

Beispiel : Klasse Point

```
class Point

feature
  x, y : Integer;

feature
  +: (p : Point) is
      result : Point;
    do
      result.x(x + p.x);
      result.y(x + p.y);
    end;
  x: (i : Integer) is
    do
      x := i;
    end;
  y: (i : Integer) is
    do
      y := i;
    end;
  ...
end -- class Rectangle
```

Exemplarvariablen (int. Verbundkomponente)

exportierte Operationen

Klassen

Beispiel : Klasse Rechteck

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do ...
    end;

feature {NONE}
  extent : Point
    do
      -- liefert einen Punkt, der die Höhe und
      -- Weite des Rechtecks repräsentiert
    end;
end -- class Rectangle
```

Exemplarvariablen

exportierte Operationen

private Operationen

Klassen

Klasse - Objekt

- **Eine Klasse entspricht einem ADT**
- **Ein Objekt "entspricht " einem abstrakten Datenobjekt**
  - d.h. die Datenimplementierung ist verborgen.
- **Von einer Klasse sind außerhalb sichtbar:**
  - der Klassenname
  - die exportierten Operationen
- **Jedes Objekt ist ein Exemplar**
  - einer Klasse des Programms
- **Objekte einer Klasse kennen dieselben Operationen**
- **Objekte einer Klasse unterscheiden sich in den Werten ihrer Daten**

Klassen

Objekte und Klassen

- Nach außen sichtbar ist

```
class Rectangle
interface
  Create;
  origin : Point;
  corner : Point;
  moveTo (p : Point);
  contains (p : Point)
   Boolean:
end -- class Rectangle
```

Zusammenhang Objekt <-> Klasse

```
r : Rectangle;
p : Point

p.Create;
r.Create;

r.contains (p);
```

```
class Rectangle

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean is
    do
    end;
feature {NONE}

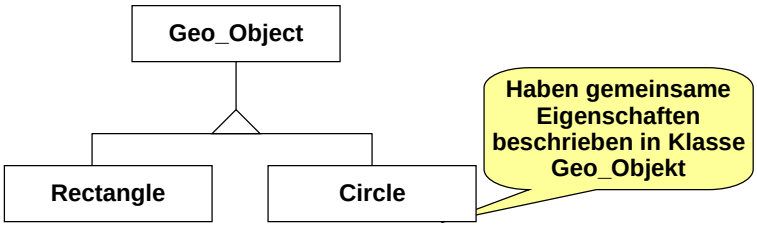
  extent : Point
    do
      -- liefert ...
    end;
end -- class Rectangle
```

Vererbung

Gemeinsame Eigenschaften, Spezialisierung

- **Gemeinsame Eigenschaften verschiedener Klassen K_1, K_2**
  - werden in einer **eigenen Klasse K** zusammengefaßt und definiert,
  - und anschließend an K_1, K_2 vererbt.

- **Beispiel**



- **Regel:**
  - Eine Klasse K_1 erbt von einer Klasse **K** genau dann, wenn K_1 eine **Spezialisierung** (Unterbegriff) von **K** ist. Umgekehrt wird **K** die **Generalisierung** genannt.

Vererbung

Beispiel 1: Einfachvererbung

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

```
class Circle

inherit
  Geo_Object
    define moveTo, contains

feature
  center : Point;
  radius : Integer:
  ...
end -- class Circle
```

Abstrakte Klasse Spezifikationsklasse

Einfachvererbung

Vererbung

Beispiel 1: Mehrfachvererbung

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;
contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;
  ...
end -- class Rectangle
```

```
class DisplayableObject

feature
  psDescription : String is
    deferred
    end;
end -- class DisplayableObject
```

```
class DisplayableRectangle

inherit
  Rectangle

inherit
  DispayableObject
    define psDescription

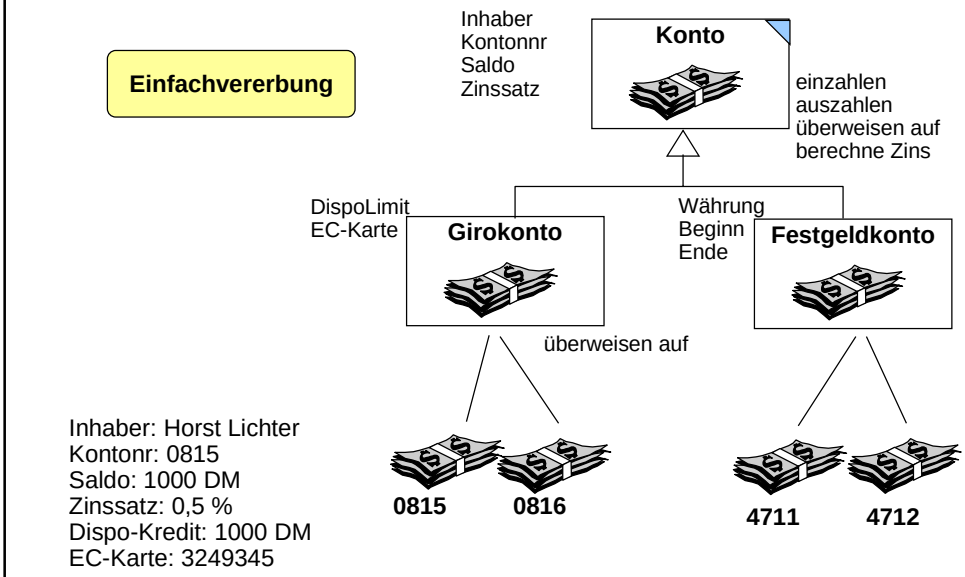
feature
  ...
end -- class DisplayableRectangle
```

Mehrfachvererbung

Vererbung

Beispiel 2: Einfachvererbung

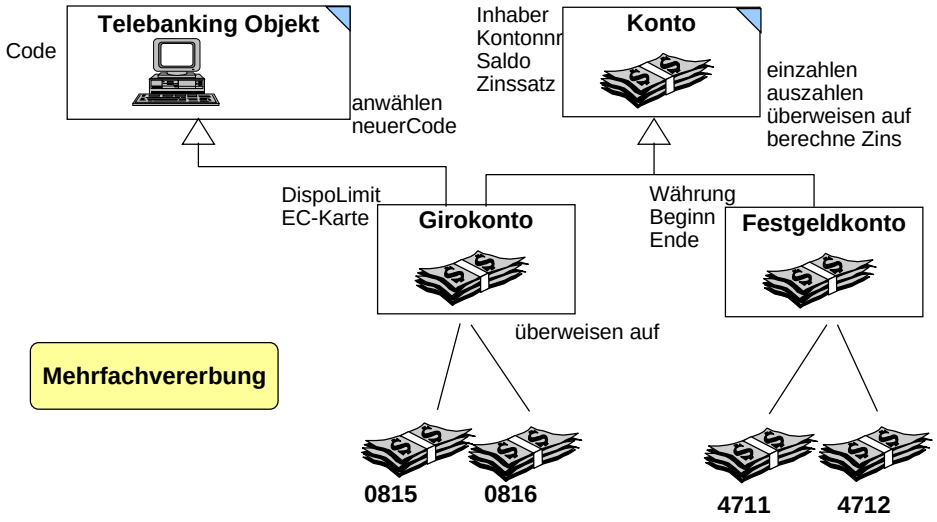
Einfachvererbung



Inhaber: Horst Lichter
Kontonr: 0815
Saldo: 1000 DM
Zinssatz: 0,5 %
Dispo-Kredit: 1000 DM
EC-Karte: 3249345

Vererbung

Beispiel 2: Mehrfachvererbung



Vererbung

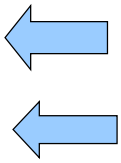
Abstrakte Klassen

- **Eine Klasse ist ein**
  - möglicherweise **nicht vollständig implementierter** ADT.
- **In konkreten Klassen (implementiert, effektiv)**
  - sind alle Operationen **ausführbar.**
- **In abstrakten Klassen (deferred, virtual)**
  - sind einige Operationen ***noch nicht implementiert*** , sondern nur spezifiziert (deklariert). Diese müssen in den konkreten Klassen angegeben werden.
- **Von konkreten Klassen**
  - können **beliebig viele** Objekte (Exemplare) erzeugt werden.
  - new- bzw. create- Nachrichten (Instanziierung)
- **Von abstrakten Klassen**
  - können **keine** Objekte erzeugt werden!

Vererbung

Beschreibungsschema von Klassen

- **Eine Klasse ist definiert durch**
- **ihren Namen**
- **ihre direkten Oberklassen**
  - die erbt_von-Beziehung muß zyklenfrei sein
- **eine Speicherstrukturbeschreibung**
  - erweitert die geerbten Beschreibungen
- **eine Menge von Operationsbeschreibungen**
  - erweitert die geerbten Beschreibungen



Vererbung

Oberklasse <-> Unterklasse

Geerbte Eigenschaften können auf drei Arten in einer Unterklasse modifiziert werden

Erweitern	etwas Neues hinzufügen
Redefinieren	sich ähnlich verhalten, Diff. formulieren
Definieren	etwas Versprochenes realisieren

Vererbung

Erweitern

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo,
    contains

feature
  origin, corner :
   Point;

feature

  height : Integer is
    do ...
    end;
  width : Integer is
    do ...
    end;
  ...
end -- class Rectangle
```

Spezifische Eigenschaften werden hinzugefügt

Vererbung

Definieren

```
class Geo_Object

feature
  moveTo: (p : Point) is
    deferred
    end;

  contains (p : Point) : Boolean
   is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

inherit
  Geo_Object
    define moveTo, contains

feature
  origin, corner : Point;

feature
  moveTo: (p : Point) is
    do
      origin := origin + p;
      corner := corner + p;
    end;
  contains (p : Point) : Boolean
   is
    do
      ...
    end;
  ...
end -- class Rectangle
```

Definieren versprochener, aber noch nicht implementierter Eigenschaften

Vererbung

Redefinieren

```
class DisplayableObject
  feature
  psDescription: String is
    deferred
    end;
end -- class DisplayableObject
```

```
class Rectangle
...
feature
  initialize is do
      origin.Create; corner.Create;
      origin.initialize;
      corner.initialize;
    end
  ...
end -- class Rectangle
```

```
class DisplayableRectangle
inherit
  Rectangle
    rename initialize as initRect
    redefine initialize
...
feature
  color : Color;

feature
  initialize is do
    current.initRect;
    color.create; color.black;
  end;
...
end -- class DisplayableRectangle
```

Die ererbte Implementierung paßt im Kontext der Unterklasse nicht mehr.

Sie wird redefiniert!

Polymorphismus und dyn. Binden

Polymorphismus

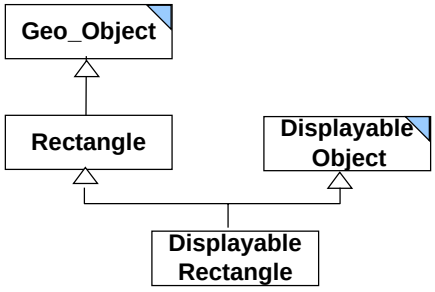
- **Allgemein:**
  - "Polymorphismus ist die Fähigkeit von Etwas, von verschiedener Gestalt zu sein"
- **In Programmiersprachen:**
  - Etwas: Variablen, Funktionen, Prozeduren
  - Gestalt: Typ
- **Eine polymorphe "Entität" kann in verschiedenen Kontexten verwendet werden,**
  - die unterschiedliche Typen verlangen.
- **Beispiele**

```
write (100, 3)
write ('a')
```

Polymorphismus und dyn. Binden

Beispiel: Polymorphismus - 1

- **Die Vererbung ist *ein* Mechanismus, um Polymorphismus in Programmiersprachen zu realisieren.**



ein DisplayableRectangle **verhält sich wie** ein Rechteck und wie ein DisplayableObject

```
dr : DisplayableRectangle;
dr.Create;

dr.contains (p);
x := dr.center;
dr.moveTo (p);

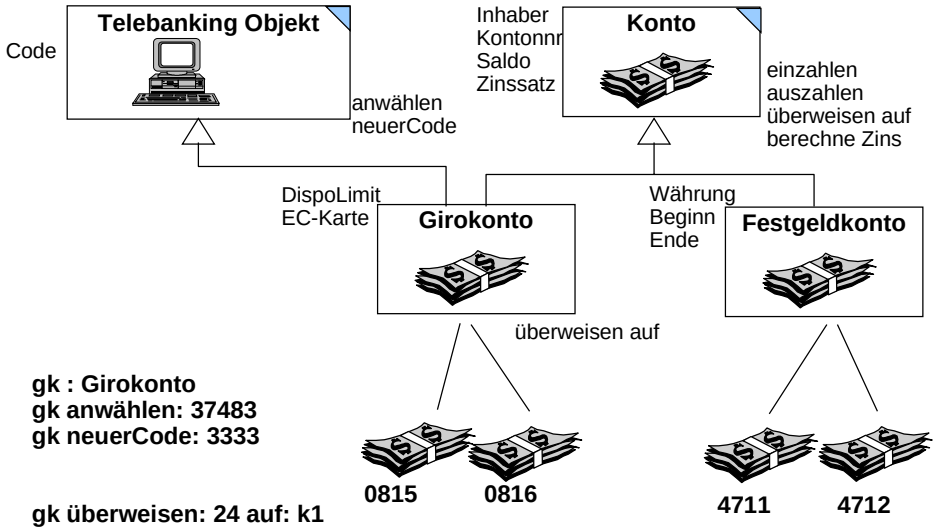
s := dr.psDescription
```

Rectangle

DisplayableObject

Polymorphismus und dyn. Binden

Beispiel: Polymorphismus - 2



gk : Girokonto
gk anwählen: 37483
gk neuerCode: 3333

gk überweisen: 24 auf: k1

Polymorphismus und dyn. Binden

Dynamisches Binden: Beispiel 1

```
g : Geo_Object;
r : Rectangle;
c : Circle;

r.Create; c.Create

g := r;
g.contains (p);

g := c;
g.contains (p);
```

```
class Geo_Object

feature
  contains (p : Point) : Boolean is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

feature
   contains (p : Point) :Boolean is
     ...
end -- class Rectangle
```

```
class Circle

feature
  contains (p : Point) :Boolean is  ...
end -- class Circle
```

Dynamisches Binden heißt:

die **richtige** Implementierung zur **Laufzeit** finden

Polymorphismus und dyn. Binden

Dynamisches Binden: Beispiel 2

Kontoarten
- Festgeld
- Giro
- Spar
- Kredit

k : Konto;
k := system.waehleKonto;
...
k überweisen: 2000 auf: k1;

```
überweisen: betrag auf: empfKonto
   saldo := saldo - betrag.
   empfängerKonto einzahlen: betrag.
```
Konto

```
überweisen: betrag auf: empfKonto
   if (saldo - betrag) >= DispoLimit then
     ...
     super überweisen: betrag auf:empfKonto.
   else
     ...
```
Girokonto

```
überweisen: betrag auf: empfKonto
   if (Datum heute <= Ende) then
     ...
     super überweisen: betrag auf:empfKonto.
   else
     ...
```
Festgeldkonto

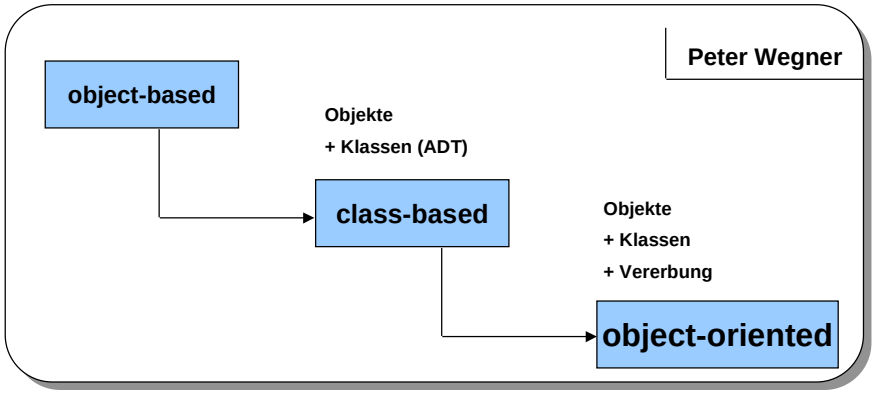
Diskussion

Varianten der Vererbung

Modifikations-möglichkeiten \ Anzahl der Oberklassen	eine oder keine	beliebig viele
nur Erweitern und Definieren	***strikte Einfachvererbung***	***strikte Mehrfachvererbung***
Erweitern Redefinieren Definieren	***nicht-strikte Einfachvererbung***	***nicht-strikte Mehrfachvererbung***

Verständnis der OO Programmierung

Merkmale objektorientierter Systeme/Sprachen



Verständnis der OO Programmierung

Klassifikation

- **Objektbasiert**
  - Objektbasierte Programmiersprachen bieten die Möglichkeit, **Objekte** im Sinne einer **Datenkapsel** resp. eines **Objektmoduls** zu realsieren.
  - Jedes Objekt wir **einzeln** beschrieben und benutzt
- **Klassenbasiert**
  - Klassenbasierte Programmiersprachen bieten die Möglichkeit, Objekte in Forn von **Objekttypen (Klassen)** zu beschreiben.
  - Von Objekttypen können beliebig viele **Exemplare** (Objekte des Typs) erzeugt werden.
- **Objektorientiert**
  - Objektorientierte Programmiersprachen erlauben, Objekttypen (Klassen) mithilfe der **Vererbungs-Beziehung** zu strukturieren.
  - Dadurch lassen sich **Objekttyp-Hierarchien** im Sinne der Spezialisierung resp. Generalisierung modellieren.

Was haben wir gelernt!

- **Klassifikation**
  - objektbasiert, klassenbasiert
  - objektorientiert
- **Klassen sind (partiell) implementierte ADTs**
  - abstrakte Klassen
  - konkrete Klassen
- **Vererbung**
  - dient dazu, Spezialisierungsbeziehung zwischen Begriffen programmiertechnisch zu realisieren.
  - Unterschied zwischen Einfach- und Mehrfachvererbung
- **Polymorphismus**
  - kann mit Hilfe der Vererbung realisiert werden
  - führt zum dynamischen Binden von Implementierungen

Glossar

- **wissenbasierte, wertorientierte, prozedurale, objektorientierte Programmierung**
- **Objektmodul, ADT, Klasse**
- **Struktur eines objektorientierten Programms**
- **Nachrichten, Methoden, Laufzeitzustand eines objektorientierten Systems**
- **abstrakte Klasse, konkrete Klasse**
- **Definieren, Redefinieren, Erweitern von Methoden bei einem Spezialisierungsschritt**
- **Vererbung (Spezialisierung), Verallgemeinerung (Generalisierung)**
- **Einfach-, Mehrfachvererbung; strikte Vererbung, nichtstrikte Vererbung**
- **Polymorphismus, Polymorphismus durch Vererbung**
- **dynamisches Binden (Dispatching)**

Objektorientierte Programmierung II: OO in Modula-3

- **Objekttypen für Klassen**
- **Untertypen für opake Klassen**
- **Vererbung zwischen opaken Klassen**
- **Diskussion**

Wiederholung ADT

- **ADTs in Modula-3**
  - ein ADT ist immer ein **Referenztyp**
  - in der Schnittstelle wird ein **opaker Typ als Untertyp** des vordefinierten Referenztyps REFANY deklariert
- **Ein ADT entspricht dem Konzept einer Klasse.**
- **Eine Exemplar eines ADTs entspricht einem abstrakten Objekt**
- **Mit Hilfe der ADTs**
  - kann eine **klassenbasierte** Programmierung in Modula-3 umgesetzt werden.
- **Zur Objektorientierung fehlen noch**
  - Sprachkonstrukte, um die **Vererbung** zu modellieren.
- **Hierfür**
  - stellt Modula-3 das Konzept der **Objekttypen** bereit.

Objekttypen für Klassen

Objekttypen in Modula-3

- **Mit Hilfe der Objekttypen können in Modula-3 Klassen beschrieben werden.**
- **Ein Objekttyp gibt an**
  - Bezeichner der Klasse (des Objekttyps)
  - Bezeichner der Oberklasse
    - Es kann maximal eine Oberklasse geben

    Einfachvererbung
  - Bezeichner der Exemplarvariablen
  - Signaturen der Methoden
  - Bezeichner der Methoden, die in der Klasse redefiniert werden
    - in Modula-3 spricht man von überschreiben (overrides)
- **Ein Objekttyp (Klasse) wird**
  - in einer Modulschnittstelle deklariert
  - die Implementierung enthält den strukturellen Aufbau des Objekttyps (Klasse) und die Realisierung der Methoden.

Objekttypen für Klassen

Beispiel Objekttyp : Schnittstelle

INTERFACE Point

```
TYPE Point = ROOT OBJECT
                x : INTEGER; y : INTEGER;
             METHODS
                getX(): INTEGER          := PointGetX;
                setX (value : INTEGER)   := PointSetX;
                getY (): INTEGER         := PointGetY;
                setY (value : INTEGER)   := PointSetY;
                add (p: Point) : Point   := PointAdd;
                . . .
...          END;
```

- **Die Klasse**
  - hat die vordefinierte Klasse **ROOT** als Oberklasse
  - deklariert zwei Exemplarvariablen x und y
- **In der METHODS-Klausel**
  - kann die **Bindung** zwischen Methode und der Prozedur, die sie realisiert, hergestellt werden!
- **Objekttypen sind spezielle Referenztypen**
  - Objekte werden mit der **NEW**-Operation erzeugt.

Objekttypen für Klassen

Beispiel Objekttyp: Rumpf des Moduls, Verwendung der Klasse

Die Prozeduren, die Methoden realisieren, erwarten als ersten Parameter ein Objekt der Klasse.

```
MODULE Points;

PROCEDURE PointGetX(self : Point): INTEGER =
BEGIN
  RETURN self.x;
END PointGetX;

PROCEDURE PointSetX(self: Point; value : INTEGER) =
BEGIN
  self.x := value;
END PointSetX;
...

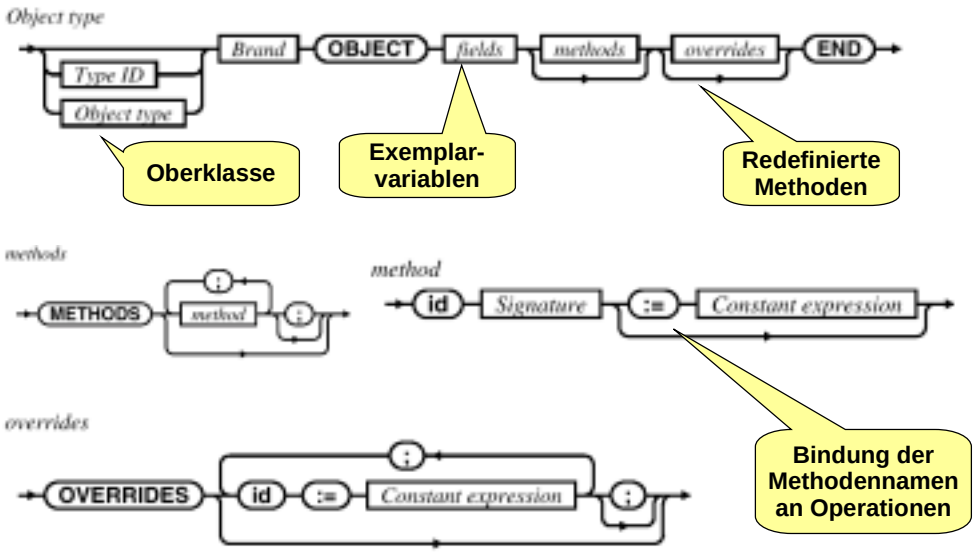
PROCEDURE PointAdd (self: Point; p: Point) : Point =
VAR newP : Point;
BEGIN
  newP := NEW(Point);
  newP.setX(self.x + p.getX());
  newP.setY(self.y + p.getY());
  RETURN newP;
END PointAdd;
...
```

Verwendung der Klasse `Point`

```
VAR p1, p2, p3 : Point;
BEGIN
  p1 := NEW(Point);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
END Point_Test.
```


Objekttypen für Klassen

Syntax der Objekttyp-Deklaration



Objekttypen für Klassen

Operationenbindung und Redefinition

```
TYPE Geo_Object =  ROOT OBJECT
       METHODS
        moveTo (p : Point):= NIL ;
        contains (p : Point) : BOOLEAN:= NIL;
       END;
```

abstrakte Klasse

```
TYPE Rectangle = Geo_Object OBJECT
         origin : Point;
         corner : Point;
       METHODS
         area() : INTEGER      := RectArea;
         height () : INTEGER   := RectHeight;
         setOrigin (p : Point) := RectOrigin;
         getOrigin(): Point    := RectGetOrigin;
         …
         asText(): TEXT        := RectAsText;
       OVERRIDES
         moveTo   := RectMoveTo;
         contains := RectContains;
       END;
```

konkrete Klasse

konkrete Klasse

```
TYPE DispRectangle = Rectangle OBJECT
         color : TEXT;
       METHODS
         setColor(c : TEXT) := DispRectSetColor;
         getColor() : TEXT  := DispRectGetColor;
         draw(): TEXT       := DispRectDraw;
       OVERRIDES
         asText := DispRectAsText
       END;
```


Untertypen für opake Klassen

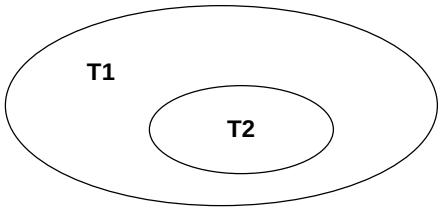
Opake Objekttypen

- **Um opake (geschützte) Objekttypen zu deklarieren**
  - verwenden wir die M3-Sprachkonstrukte, die wir zur Realisierung ADTen benutzt haben!
    - **Trennung** von Schnittstelle und Implementierung
      - Pro Objekttp (Klasse) ein Modul
    - **Opake** Typen
    - Konzept der **Untertypen**
    - Konvention der **T-Notation** für Objekttypen
- **Der Name der Klasse wird als opaker Typ deklariert**
  - Deklaration in der Schnittstelle.
  - Ein opaker Typ muß immer ein Untertyp eines Referenz- oder Objekttyps sein.
  - Der Obertyp (ein Objektyp) deklariert die **öffentliche** Schnittstelle
    - D.h. nur die Signaturen der Methoden werden angegeben

Untertypen für opake Klassen

Untertypen in Modula-3

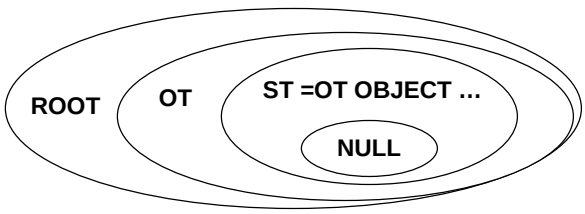
- **Subtypen in Modula-3**
  - Modula-3 kennt das Konzept der Konstruktion von ***Untertypen***
  - Subtypbeziehung wird durch "***<:***" angezeigt
- **Definition**
  - Seien T1 und T2 Typen und die Relation T2 <: T1 besteht, dann sind ***alle Werte*** von T2 auch ***Werte*** von T1
  - T1 nennt man ***Obertyp***; T2 nennt man ***Untertyp***
  - Es gilt: ein Typ kann beliebig viele Untertypen haben, ein Typ kann ***maximal*** einen Obertyp haben.



Untertypen für opake Klassen

Untertypen von Referenz- und Objekttypen

- **Modula-3 definiert zwei vorgegebene Referenztypen**
  - ***REFANY*** und ***NULL*** mit folgender Subtypbeziehung
  - `NULL <: REF T <: REFANY`
- **Modula-3 definiert einen vorgegebenen Objekttyp**
  - ROOT mit folgender Subtypbeziehung
  - `NULL <: ST = OT OBJECT … END <: OT <: ROOT <: REFANY`
- **Interpretation:**
  - jeder Objekttyp ist ***Untertyp*** von ROOT (und damit ein Referenztyp)



Untertypen für opake Klassen

Beispiel: opaker Objekttyp, Schnittstelle

```
INTERFACE Point;

TYPE Point <: PublicPoint;

     PublicPoint = ROOT OBJECT
             METHODS
               getX(): INTEGER;
               setX (value : INTEGER);
               getY (): INTEGER;
               setY (value : INTEGER);
               add (p: Point) : Point;
               minus (p : Point) : Point;
               asText(): TEXT;
             END;
END Point.
```

Point ist Untertyp des "öffentlichen Teils von Point"

Dieser Typ wird exportiert und kann benutzt werden!!!!!!

In der Implementierung müssen die Exemplar-variablen und die Operationen, die die Methoden realisieren, angegeben werden!

Point <: PublicPoint <: ROOT

Untertypen für opake Klassen

Opaker Objekttyp: Rumpf

```
MODULE Point;

REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX;
        getX := PointGetX;
        setY := PointSetY;
        getY := PointGetY;
        add := PointAdd;
        minus := PointMinus;
        asText := PointAsText;
      END;

PROCEDURE PointGetX(self : Point): INTEGER
=
BEGIN
  RETURN self.x;
END PointGetX;
...
END Point.
```

Rumpf der Klasse

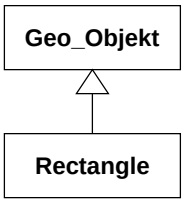
- **Point wird als Objekttyp deklariert**
  - PublicPoint ist der **Obertyp**
  - Point <: PublicPoint <: ROOT
- **Point erweitert den Obertyp**
  - um die fehlenden **Instanzvariablen.**
- **Point bindet**
  - an die Methoden entsprechende **Implementierungen.**

Untertypen für opake Klassen

Diskussion dieser Realisierung

- **Untertyp-Konzept wird verwendet, um**
  - **Spezialisierungsbeziehung** zwischen (hier offenen) Klassen auszudrücken

```
TYPE Rectangle = Geo_Object OBJECT
        origin : Point;
        corner : Point;
      METHODS
        area() : INTEGER      := RectArea;
        …
        asText(): TEXT        := RectAsText;
      OVERRIDES
        moveTo   := RectMoveTo;
        contains := RectContains;
      END;
```



  - geschützte Objekttypen (Klassen) zu **implementieren**

```
REVEAL
  Point = PublicPoint BRANDED OBJECT
        x : INTEGER;
        y : INTEGER;
      OVERRIDES
        setX := PointSetX; ...
      END;
```


Vererbung zwischen opaken Klassen

T-Konvention

T bezeichnet immer den im Modul realisierten Objekttyp (Klasse)

Verwender

```
MODULE Point_Test  EXPORTS Main;

IMPORT Point,  SIO;

VAR p1, p2, p3 : Point.T;
BEGIN
  p1 := NEW(Point.T);
  p1.setX(10);
  p1.setY(10);
  p2 := NEW(Point.T);
  p2.setX(20);
  p2.setY(20);
  p3 := p1.add(p2);
  SIO.PutLine(p3.asText());
END Point_Test.
```

Schnittstelle

```
INTERFACE Point;

TYPE T <: Public;

          Public = ROOT OBJECT
              METHODS
                setX(): INTEGER;
                ...
                add (p: T) : T;
                minus (p : T) : T;
                asText(): TEXT;
              END;
END Point.
```

Rumpf

```
MODULE Point;

  REVEAL T = Public BRANDED OBJECT
        x:INTEGER;
        y:INTEGER;
     OVERRIDES
        setX:= PointSetX;
        ...
     END;
  ...
```

Vererbung zwischen opaken Klassen

Implementierung der Klasse Point

```
PROCEDURE AsText (self : T) : TEXT =
BEGIN
  RETURN (Fmt.Int(self.x)  &  "&"  &  Fmt.Int(self.y));
END AsText;

PROCEDURE GetX(self : T): INTEGER =
BEGIN
  RETURN self.x;
END GetX;

PROCEDURE GetY(self : T): INTEGER =
BEGIN
  RETURN self.y;
END GetY;

...

PROCEDURE Add (self: T; p: T) : T =
VAR newP : T;
BEGIN
  newP := NEW(T);
  newP.setX(self.getX() + p.getX());
  newP.setY(self.getY() + p.getY());
  RETURN newP;
END Add;
```


Vererbung zwischen opaken Klassen

Beispiel: Klassenhierarchie Geo_Objects

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```

Vererbung zwischen opaken Klassen

Implementierung von Rectangle

```
MODULE Rectangle;
IMPORT Point;

REVEAL
  T = Public BRANDED OBJECT
        origin : Point.T;
        corner : Point.T;
      OVERRIDES
        setOrigin := SetOrigin;
        getOrigin := GetOrigin;
        setCorner := SetCorner;
        getCorner := GetCorner;
        area := Area;
        height := Height;

        moveTo   := MoveTo;
        contains := Contains;
        asText   := AsText;
      END;

PROCEDURE AsText(self : T) : TEXT =
BEGIN
  RETURN ("Rect origin: " & self.origin.asText() &
             " corner: " & self.corner.asText());
END AsText;
```

Notwendig, um geschützte Implementierung zu erhalten

Echte Redefinitionen

```
PROCEDURE MoveTo (self : T; p : Point.T)=
VAR newPoint := NEW(Point.T);
BEGIN
  newPoint := self.getCorner();
  self.setCorner(newPoint.add(p));

  newPoint := self.getOrigin();
  self.setOrigin(newPoint.add(p));
END MoveTo;


PROCEDURE Contains (self : T; p : Point.T):
                    BOOLEAN =
BEGIN
  ...
END Contains;
```


Vererbung zwischen opaken Klassen

Implementierung von `DispRectangle`

```
MODULE DisplayableRectangle;

IMPORT Rectangle;
TYPE SuperClass = Rectangle.T;

REVEAL
  T = Public BRANDED OBJECT
        color : TEXT;
      OVERRIDES
        setColor := SetColor;
        getColor := GetColor;
        draw     := Draw;

        asText := AsText;
END;
```

Echte Redefinition

- **Direkter Aufruf einer Methode der Oberklasse**
  - Sollte im Normalfall **nicht** gemacht werden.
  - Sinnvoll jedoch, wenn **rekursiv redefiniert** wird, d.h. daß die Leistung der redefinierten Methode bei der neuen Implementierung verwendet werden soll.

```
PROCEDURE AsText(self : T) : TEXT =
VAR t : TEXT;
BEGIN
  t := SuperClass.asText(self);
  RETURN (t & " color: " & self.color );
END AsText;
```

Verwenden der gerade redefinierten der Oberklasse

Vererbung zwischen opaken Klassen

Umgang mit Objekten

```
IMPORT Point,  SIO, Rectangle, DisplayableRectangle;
VAR
   p1, p2, p3 := NEW(Point.T);
   r1         := NEW(DisplayableRectangle.T);

BEGIN
  p1.setX(10);
  p1.setY(10);
  SIO.PutLine(p1.asText());

  p2.setX(100);
  p2.setY(100);
  SIO.PutLine(p2.asText());

  r1.setOrigin(p1);
  r1.setCorner(p2);
  r1.setColor("black");
  SIO.PutLine(r1.asText());

  p3.setX(50);
  p3.setY(50);
  SIO.PutLine(p3.asText());

  r1.moveTo(p3);
  SIO.PutLine(r1.asText());
```

Deklarieren der Variablen und erzeugen der Objekte

Senden von Nachrichten an die erzeugen Objekte

```
10&10
100&100
Rect origin: 10&10 corner: 100&100 color: black
50&50
Rect origin: 60&60 corner: 150&150 color: black
```


Diskussion

Objektorientierung in M3

- **Feststellung:**
  - Es ist **möglich**!
- **Anwendbarkeit und Verbindung mit anderen Paradigmen**
  - Modula-3 ist eine **hybride** Sprache
  - Sie erlaubt **imperative** und **objektorientierte** Programmierung
  - Beides kann bunt durcheinander **gemischt** werden
    - Vorteil: Jedes Konzept an seinem Platz
    - Nachteil: Mischung, unüberlegt angewendet, kann undurchsichtig werden
- **Die Implementierung der objektorientierten Konzepte**
  - basiert auf dem Untertyp-Konzept der Sprache
  - die OO-Konzepte können damit umgesetzt werden
- **Konsequenz**
  - Soll durchgängig objektorientiert entwickelt werden, dann sollte man eine **rein objektorientierte** Sprache verwenden, z.B. Java, Eiffel, Smalltalk!
  - die Vorteile eines hybriden Ansatzes sind nicht mehr gegeben

Was haben wir gelernt?

- Realisierung von Klassen in Modula-3
- Objekttypen zur Formulierung von Klassen in der Schnittstelle von Modulen
- offene Klassen (nicht zu empfehlen, wider Datenabstraktion), opake Klassen
- Unterkonzept für opake Klassen (Information Hiding) und für Vererbung
- Je Klasse ein Modul und eine Schnittstelle
- In Modula-3 kann objektorientiert programmiert werden (etwas umständlich)
- T-Konvention
- Vorteil hybrider Sprachen: OO und objektbasierte, adt-basierte sowie prozedurale Programmierung
- Nachteil: Programme können sehr unübersichtlich sein; reine OO-Programme auch
- Objektorientierte Programmierung schwierig: Struktur ist nur die Klassenhierachie, saubere Klassenhierachien zu modellieren, ist schwer

Glossar

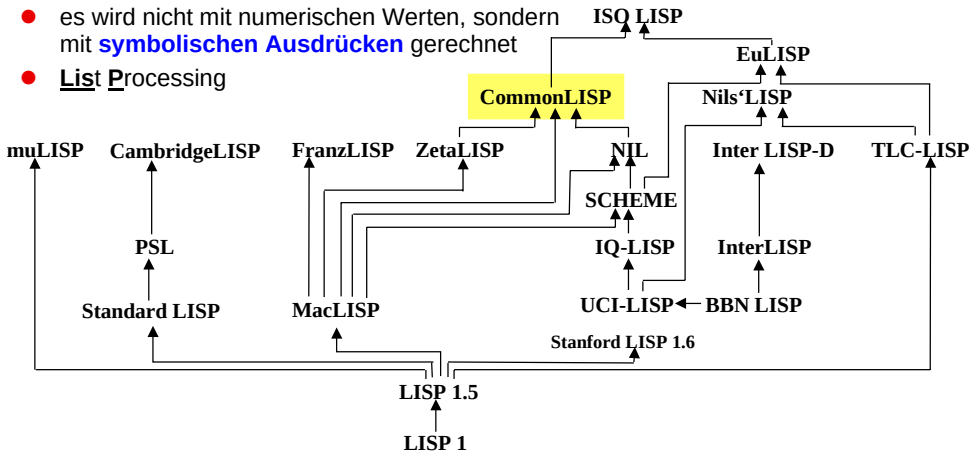
- ADT bzw. Klasse, Realisierung durch ADT-Modul bzw. opake Klasse in Interface mit zugehörigem Rumpf
- Objekttypen von Modula-3, Deklaration in einer Schnittstelle
- Examplarvariable, Signatur von Methoden
- Redefinition und Binden von Prozeduren an die Methoden der Signatur bzw. der redefinierten Methoden
- offene Klassen, opake Klassen
- Schnittstelle für opake Klasse, Details im Rumpf über die REVEAL-Klausel
- Aufruf einer Methode der gleiche Klasse über `self`
- Aufruf einer Methode der Oberklasse über `super`

Applikative Programmierung in LISP I

- Allgemeines
- Listen für Daten und Programme
- Wertzuweisungen, -ermittlung, -interpretation
- Listenverarbeitung
- Prädikate

Historie

- **LISP**
  - die bekannteste **applikative** Sprache
  - 1958 von John McCarthy entwickelt
  - für Probleme der **KI** entwickelt
  - es wird nicht mit numerischen Werten, sondern mit **symbolischen Ausdrücken** gerechnet
  - **List Processing**



Allgemeines

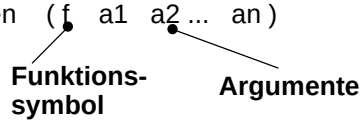
Charakterisierung

- **Vielfältiger Einsatz in der KI**
  - **Wissensverarbeitung**
  - **Bildverarbeitung / Bildverstehen**
  - **Übersetzung natürlicher Sprachen**
  - **Automatisches Beweisen**
  - **Symbolische Algebra etc.**
- **Interaktive Arbeitsweise**
  - Interpreter statt Compiler
- **LISP Programme und Daten haben dieselbe Form**
  - LISP-Programm kann andere Programme als Daten benutzen
- **Literatur:**
  - **Anderson/Corbett/Reisner: Essential LISP**
  - **Winston/Horn: LISP**
  - **Stoyan/Görz: LISP, eine Einführung in die Programmierung**
  - **Mayer: Programmieren in COMMON Lisp**

Listen für Daten und Programme

LISP-Programm

- **LISP – Programm:** **Folge von Ausdrücken, die nacheinander ausgewertet werden**
- **Ausdrücke**
  - Atome mit Wert
  - oder Listen (f a1 a2 ... an)



Wert des Ausdrucks ist der Funktionswert für diese Argumente

  - Präfixnotation, klammerlos

Listen für Daten und Programme

Beispiele für Ausdrücke

Prompt Ausdruck Kommentare

Ergebnis

```
----> (+  11  6)            ;  Addition
17
----> (*  7  8)
56
----> ( /  10.0  4)
2.5
----> (-  4711 )            ;  unäres Minus
-4711
----> (GCD  91  49)
7
----> (MAX 13  8  25)
25
----> (+ (+ 17.0  4) (EXPT  2  3))
29.0
```


Listen für Daten und Programme

Atom, Liste, Form

- **Atome**
  - Zahlenliterale
  - Symbole (Literale, Konstante, Variablen)
    `T, NIL, ANTON, ARGUMENT-1, X_17`
  - Zeichenketten(literale) „a b c d“ „~ A“
- **Liste**

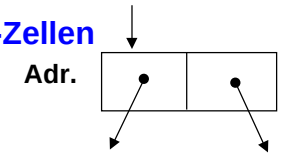
  (listelem listelem ... listelem)

  ( ) } leere Liste
  NIL }
- **Ausdrücke**
  - Atome oder Liste
- **Form**
  - auswertbarer Ausdruck

Listen für Daten und Programme

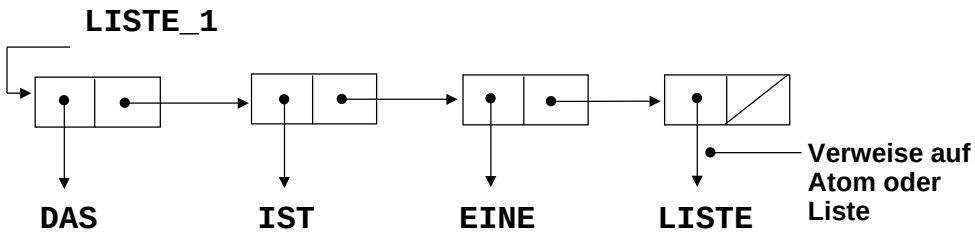
Interne Listendarstellung

- **Listen können durch CONS-Zellen repräsentiert werden.**



2 Inhalte jeweils Adressen (Zeiger)

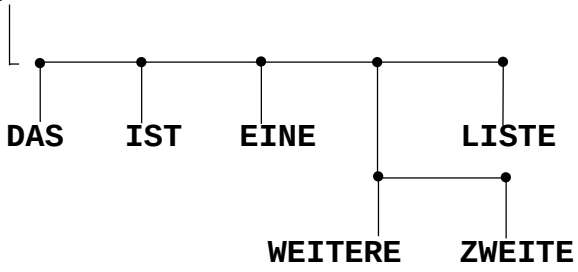
- **Liste**
  - Folge von CONS-Zellen, die über den rechten Zeiger verknüpft sind.
- **Beispiel**
  - `(DAS IST EINE LISTE)` sei Wert des Atoms `LISTE_1`



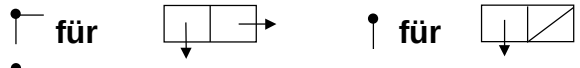
Listen für Daten und Programme

Liste für Daten

- **`(DAS IST EINE (WEITERE ZWEITE) LISTE)` abgekürzt notiert als**



- **Abkürzungen / Bedeutung**



Referenz auf den Wert: Atom, Liste

Übergang zu nächsten Listenelement

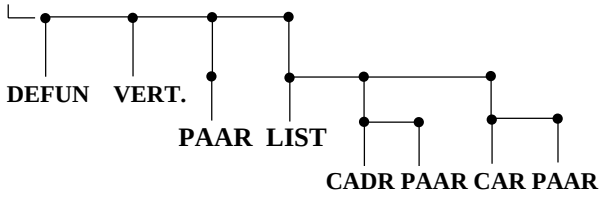
Listen für Daten und Programme

Liste für Funktion

- `(DEFUN VERTAUSCHE (PAAR)`

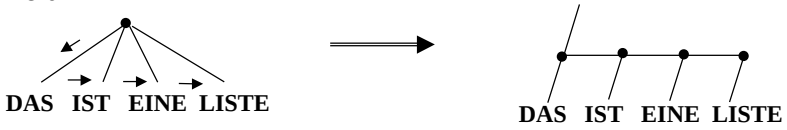
Liste für Funktion

`(LIST(CADR PAAR)(CAR PAAR))))`



- **Lisp - Liste und allgemeine Liste**

Lisp – Liste ist erster Sohn – nächster Bruder– Darstellung (Binärbaum-Darstellung) eines n-ären Baums (→ Datenstrukturen): Wert ins linke Zeigerfeld



Funktionen

Funktionen: Definition und Aufruf

- **Definition (Deklaration)**
  - `(DEFUN FName [Parlist] [Rumpf] )`
  - Auswertung:
    - Wert Funktionsname
      Seiteneffekt Zuordnung FName → Rumpf
- **Aufruf**
  - `(FName` a_1 a_2 … a_n `)` a_i Ausdrücke
  - Auswertung:
    - a_1, a_2 ... werden ausgewertet
    - Bindung an formale Parameter (der Reihenfolge nach)
    - Auswertung des Rumpfs
    - Wert des letzten Ausdrucks ist der Wert der Funktion
    - falls 0 ist Wert NIL
    - Bindungen werden aufgelöst

Wertzuweisung, -ermittlung, Interpretation

Identitätsfunktion

- **QUOTE ist die Identitätsfunktion**
- **Beispiel**
  - --> `(QUOTE (DA DA DA))`
  - `(DA DA DA)`
- **Achtung**
  - QUOTE wertet seine Argumente nicht aus!
  - Sonst wäre das Resultat von (QUOTE (DA DA DA)) nicht (DA DA DA). Sondern das Resultat
    - der Anwendung der Funktion DA
    - auf die Argumente DA DA
- **Abkürzung**
  - `(QUOTE A)` kann kürzer dargestellt werden als `'A`

Wertzuweisung, -ermittlung, Interpretation

Wertzuweisung

- **Die Wertzuweisung an ein symbolisches Atom geschieht durch die Systemfunktion** `SETQ`

```
(SETQ L '( A  B ))
```

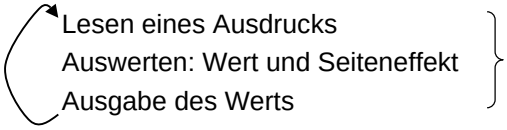
Quote-Zeichen für `(QUOTE(A B))`, Funktion mit Seiteneffekt

- **Beispiele**
  - `(SETQ x 5)`
  - `(SETQ y x)`
  - `(QUOTE x)`
  - `(SETQ y 'x)`
  - `y`
  - `(+ 'x 7)`
  - `'(+ 5 7)`
  - `('+ 5 7)`
  - `(+ '7 '4)`

Wertzuweisung, -ermittlung, Interpretation

Ausführung eines Programms

- **Nacheinanderauswertung der Ausdrücke durch Interpreter**



- **Seiteneffekte**
  - Zustand z: Gesamtheit von Bindungen von Symbolen an Werte
  - nach Ausdrucksermittlung, neuer Zustand z'
- **Änderungen von Bindungen nur über Seiteneffekte**

Wertzuweisung, -ermittlung, Interpretation

EVAL als Systemfunktion

```
----> (SETQ A 'B)   ; Zuweisung des Symbols und nicht des
B                   ; Werts von B an A
----> (SETQ B   '(+ 3 4))
(+ 3 4)
----> A
B
----> B
(+ 3 4)
----> (EVAL  A)
7
```

Auswertung von A

damit EVAL B

Listen verarbeitung

Motivation und Operationen

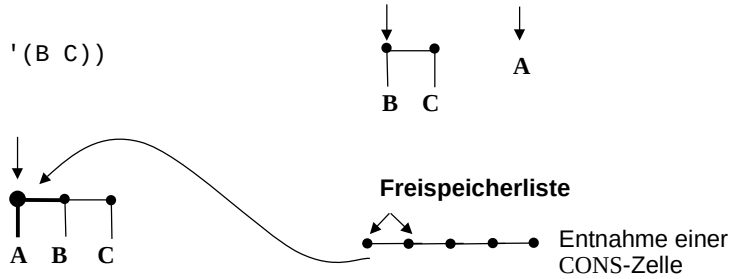
- **LISP Symbolverarbeitung**
  - abgebildet auf Listenverarbeitung
  - Grundmaschinerie jedes LISP-Systems
  - Notwendigkeit entsprechender Listenoperationen
- **Listenoperationen**
  - Aufbau
  - Zerlegung
  - Erweiterung
  - Spiegelung
  - etc.

Listen verarbeitung

Listenaufbau mit CONS

- **Aufbau**
  - `(CONS Ausdruck1 Ausdruck2)`
    Wert: Listenel1 Liste1
- **Ergebnis**
  - Die Liste, die durch Anhängen von Listenel1 am vorderen Ende von Liste1 entsteht
  - kein Seiteneffekt !!
- **Beispiel**

```
----> (CONS 'A '(B C))
```



Listen verarbeitung

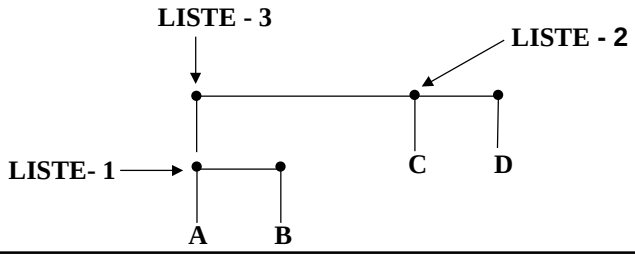
Weitere Beispiele mit CONS

```
-----> (CONS '(A B)'(C D))
((A B) C D)

-----> (SETQ LISTE_1 '(A B))
(A B)

-----> (SETQ LISTE_2 '(C D))
(C D)

-----> (SETQ LISTE_3 (CONS LISTE_1 LISTE_2))
((A B) C D)
```



Listen verarbeitung

Zerlegen von Listen mit CAR, CDR

- `(CAR L)`
  - liefert erstes Element, falls L nicht leer sonst ()
- `(CDR L)`
  - liefert Rest als Liste, leere Liste falls L leer war oder nur aus einem Element bestand
- **Beispiele:**

Keine Seiten-effekte

```
----> (CAR '( (A B) C))

----> (CDR '((A B) C))

----> (SETQ SPRUCH '(WISSEN IST MACHT))

----> (CAR SPRUCH)

----> (CDR SPRUCH)

----> (CAR 'SPRUCH)
```

Listen verarbeitung

Weitere Beispiele mit CONS, CAR, CDR

```
----> (CDR  '(CAR  '((A B)  C)))      ; Quotierung schützt '(CAR...)
((QUOTE  ((A B)  C)))                  ; vor Auswertung

----> (CAR  (CONS  'A  '(B C)))       ; Listen man. auf Programme
A                                      ; als Listen

----> (CDR  (CONS  'A  '(B C)))
(B C)

----> (CAR   (+  17  4))              ; Wert von (+ 17  4) ist Atom
ERROR

----> (CAR  '(+  17  4))
+

----> (CDR  '(+  17  4))
(17  4)
```


Listen verarbeitung

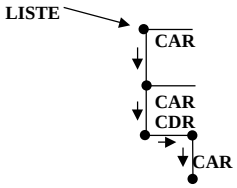
Zusammengesetzte Zerlegung: Zugriffspfade

- **Erweiterungen durch zusammengesetzte Zugriffsoperationen**

  `(CAR (CDR (CAR (CAR LISTE))))`

  abgek. zu `(CADAAR LISTE)`

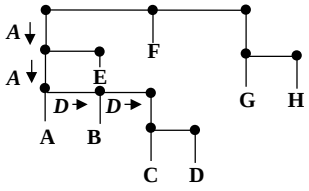
  bis zur Tiefe 4, Lesen von rechts nach links



- **Beispiel:**

  `(CDDAAR '(((A B (C D )) E) F (G H)))`

  `((C D))`



Listen verarbeitung

Zusammenhang

- (CONS 'Π '($_1$ $_2$ ... $_n$)) Wert (Π $_1$ $_2$ ... $_n$)
  (CAR '(Π $_1$ $_2$ ... $_n$)) Wert Π
  (CDR '(Π $_1$ $_2$ ... $_n$)) Wert ($_1$ ... $_n$)
- **LISP bietet bequemere Listenoperationen**
- **Diese sind alle mit Hilfe von `CONS, CAR, CDR` implementierbar**

Listen verarbeitung

Zusammenfügen durch APPEND

- `(APPEND L1 L2 .... Ln)` **Li hat als Wert eine Liste**
- **Ergebnis:**
  - Die Liste, die durch Aneinanderfügen der Elemente von Li gewonnen wird.
- **Beispiele:**

```
-->(APPEND '(A B) '(C D))
(ABCD)
-->(SETQ L '(A B))
(A B)
-->(CONS L L)
((A B) A B)
-->(APPEND L L)
(A B A B)
```

Listen verarbeitung

Zusammenfügen durch LIST

- `(LIST Ausdr-1 Ausdr-2 ... Ausdr-n)`
- **Ergebnis**
  - Liste der Werte der Argumente in der gegebenen Reihenfolge
- **Beispiele:**

```
--->(LIST '(A B) '(C D))
((A B) (C D))

--->(LIST L L)
((A B) (A B))
```


Listen verarbeitung

Weitere Beispiele

```
-----> ( CONS  'L  L)
( L  A  B)

-----> (APPEND 'L  L)            ; 'L keine Liste
ERROR

-----> (LIST  'L  L)
(L  (A B))

-----> (APPEND '(C)'() L '(D E))
(C A B D E)

-----> (LIST  L L  '(C D ))
(( A B  (AB)  (C D))

-----> (APPEND '() '() )
NIL

-----> (LIST   '() '() )
( NIL  NIL)
```

Listen verarbeitung

Weitere Funktionen zur Listenmanipulation

- **(REVERSE Liste)**

  Wert: Liste, die auf oberstem Niveau die Reihenfolge der Elemente invertiert, die Elemente bleiben unverändert

  ---▸**(REVERSE '(A B))**
  (B A)

  ---▸**(REVERSE '((A B) (C D)))**
  ((C D) (A B))

  ---▸**(REVERSE (APPEND '(A B) '(C D)))**
  (D C B A)

- **(LAST Liste1)**

  Wert: Liste, die das letzte Element von Liste1 als einziges Element enthält

  ---▸**(LAST '(((A B) (C D)) (A B)))**
  ((A B))

Prädikate und Strukturermittlung

Prädikate

- **Bedingungen**
  - werden durch Prädikate, die durch Junktoren verknüpft sind realisiert
- **Prädikat ist eine Funktion, die einen Wahrheitswert liefert**
- **Wahrheitswerte**
  - falsch NIL
  - wahr T
- **Strukturprädikate für Atome, CONS-Paare, Listen**
  - ist Atom
  - ist leer
  - ist Lisp-Ausdruck
  - Übereinstimmung von Listen

Prädikate und Strukturermittlung

Strukturprädikate für Atome

- `(ATOM Ausdruck)`

  Wert: T, falls der Wert von Ausdruck nicht mit CONS aufgebaut werden kann, NIL sonst

- `(CONSP Ausdruck)`

  Wert: T, falls der Wert von Ausdruck mit CONS aufgebaut werden kann, NIL sonst

  damit Präzisierung des Begriffs „Atom“

- **Beispiel:**

```
--->(ATOM 'NIL)
T
--->(ATOM T)
T
--->(ATOM 'ZWILLINGE)
T
--->(SETQ ZWILLINGE '(MAX MORITZ))
(MAX MORITZ)
--->(ATOM ZWILLINGE)
NIL
```

```
--->(CONSP ZWILLINGE)
T
--->(CONSP '(MAX MORITZ)
T
--->(ATOM '(+ 17 4))
NIL
--->(CONSP '(A B C))
T
--->(CONSP '17)
NIL
```


Prädikate und Strukturermittlung

Strukturprädikate für Ausdrücke

- **(NULL Ausdruck)**

Wert: T, falls Ausdruck als Wert die leere Liste hat, NIL sonst

```
-----> (NULL '() )
T
-----> (NULL NIL)
T
-----> (NULL 0)
NIL
-----> (NULL (CDDR '(A B)))
T
```

- **(LISTP Ausdruck)**

Wert: T, falls der Wert von Ausdruck das Prädikat CONSP erfüllt oder NIL ist, NIL sonst

```
-----> (LISTP 'T)
NIL
-----> (LISTP 'NIL)
T
-----> (LISTP '( ))
T
```

Prädikate und Strukturermittlung

Strukturprädikate für Listen

- **Übereinstimmung von LISP-Strukturen**

  Vergleich bzgl. interner Speicherstrukturen : EQ
  Vergleich bzgl. externer Darstellung : EQUAL

- `(EQ Ausdr1 Ausdr2)`

  Wert T, falls dem Wert der beiden Argumente dieselbe Speicherstruktur zugrundeliegt, NIL sonst

- `(EQUAL Ausdr1 Ausdr2)`

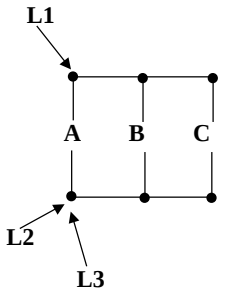
  Wert T, falls der Wert der beiden Argumente die gleiche externe Repräsentation besitzt, NIL sonst

Prädikate und Strukturermittlung

Beispiele zu Listenvergleich

- ------► (SETQ L1 (LIST 'A 'B 'C))
  (A B C)
  ------► (SETQ L2 (LIST 'A 'B 'C))
  (A B C)
  ------► (SETQ L3 L2)
  (A B C)

- ------► (EQ 'A 'A)
  T
  ------► (EQ 'A 'B)
  NIL
  ------► (EQ L1 L2)
  NIL
  ------► (EQ L2 L3)
  T
  ------► (EQUAL L1 L2)
  T
  ------► (EQUAL L2 L3)
  T
  ------► (EQ (CAR L1) (CAR L2))
  T



haben die gleiche Struktur, sind aber nicht die gleichen Speicherstrukturen

haben beide das Atom A als Wert; das hat eindeutige Speicherrepräsentation

Prädikate und Strukturermittlung

Leere Liste

- **Bemerkung**

  NIL und leere Liste -darg. ( )- sind äquivalent

- **Beispiel**

```
------>(EQ   NIL '( ))
  T                ; leere Liste bei Angabe stets durch NIL

------>(ATOM   '( ))
  T

                   ; Sei  a Ausdruck mit Wert als Liste
------>(ATOM a)
  T                ; g. d. w.   Liste leer
```

NIL ist der einzige Ausdruck der gleichzeitig Liste und Atom ist.

Prädikate und Strukturermittlung

Listenauskunftsfunktionen

- **(MEMBER Ausdruck Liste)**

  Wert: Liste, falls in der Liste (auf oberstem Niveau) ein Element mit dem Wert des Ausdrucks auftaucht; die Restliste ab erstem Auftauchen dieses Elements (incl.) NIL sonst

- **Beispiele:**

```
------>(SETQ   FUENF-X   '(X1 X2 X3 X4 X5))
(X1 X2 X3 X4 X5)

------>(MEMBER  'X6  FUENF-X)
NIL

------>(MEMBER  'X3  FUENF-X)
(X3 X4 X5)

------>(SETQ  A 'X4)
X4

------>(MEMBER A  FUENF-X)
(X4 X5)
```

Prädikate und Strukturermittlung

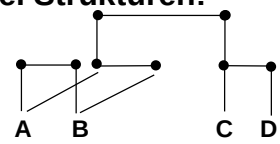
Weitere Beispiele - 1

```
-------> (MEMBER  A  '(( X1 X2)  (X3 X4 X5)))
NIL                              ; X4 nicht El. der zweielementigen Liste

-------> (MEMBER  A  (CDR  '(( X1 X2)  (X3 X4 X5))))
NIL                              ; X4 nicht El. von ((X3  X4 X5))

-------> (MEMBER  A  (CADR  '(( X1 X2)  (X3 X4 X5))))
(X4 X5)                          ; X4 ist El. von (X3 X4 X5)
```

Bemerkung 1:
MEMBER verwendet üblicherweise EQ bei Strukturen!
Problem, wenn erstes Argument eine Struktur ist.




```
-------> (MEMBER    '(A B)   '((A B) (C D)))
NIL
```

Erstes El. der zweiten Liste hat zwar dieselbe Struktur aber nicht dieselbe Speicherstruktur

Prädikate und Strukturermittlung

Weitere Beispiele - 2

- **Abhilfe**
  vorschreiben von EQUAL Strukturvergleich

  ```
  (MEMBER '(A B)   '((A B) (C D)) : Test  'EQUAL)
  ((A B) (C D))
  ```

  Systemprädikat kann auch selbstdefiniertes Prädikat sein

- **MEMBER ist kein Prädikat, liefert im positiven Fall Liste, kann aber als Prädikat verwendet werden, da für wahr irgendein Wert ≠ NIL genommen wird**

- **Probleme:**

```
---> (EQ   T   (MEMBER   'Y   '(X  Y  Z)))
NIL
---> (EQUAL  T   (MEMBER  'Y   '(X  Y  Z)))
NIL
```

Prädikate und Strukturermittlung

Längenbestimmung

- **(LENGTH Liste)**

  Wert: Anzahl der Elemente der Liste

- **Beispiel:**

```
------> (SETQ  L  '(A B))
(A B)
------> (LENGTH  L)
2
------> (LENGTH '((A B)  (C D)))
2
------> (LENGTH  (APPEND  L '(A B)))
4
------> (LENGTH  LIST  L  '((A B)))
2
------> (LENGTH  (MEMBER  'X3  '(X1 X2 X3 X4 X5)))
3
```


Prädikate und Strukturermittlung

Überprüfung/Vergleich von Atomen

- **Wertvergleiche**
  - Zahlen ganze Zahlen
    Gleitpunktzahlen verschiedener Genauigkeit
    komplexe Zahlen
  - Ergebnis von EQ nicht definiert, da Zahlen des gleichen Typs und mit gleichem Wert nicht notwendigerweise im gleichem Speicherbereich untergebracht sind.
  - Vergleich mit EQUAL liefert für Zahlen des gleichen Typs und Werts passendes Ergebnis
- **Beispiele:**

```
--->(EQ  2.0  2)              ; Interne Darstellung i.a. verschieden
NIL
--->(EQ  2.13  2.13)          ; nicht def.
NIL oder T                    ; systemabhängig
--->(EQUAL 2.0  2)            ; untersch.Typ
NIL
--->(EQUAL  4.5  (+ 2.3  2.2))
T
--->(EQUAL  4.5  '(+ 2.3  2.2)) )
NIL
```

Resumeé : EQ ungeeignet
EQUAL bedingt geeignet

Prädikate und Strukturermittlung

Spezielle Prädikate und Beispiele - 1

- **Spezielle Prädikate**

  (= Ausdr1 Ausdr2 ... AusdrN) alle Werte Zahlen sind gleich
  (< Ausdr1 Ausdr2 ... AusdrN) „ „ „ aufsteigend geordnet
  (> Ausdr1 Ausdr2 ... AusdrN) „ „ „ absteigend geordnet
  sonst NIL

- **Aufruf mit einem Argument, Ergebnis T**

  =,<, > haben keine Seiteneffekte, die Auswertung der Ausdrücke kann welche produzieren

- **Beispiele:**

```
---> (=  2.0  2.00  2)     ; Typen müssen nicht notwendigerweise übereinstimmen
 T
---> (< -1.8   0   2.3  4)
 T
---> (>  17)
 T
---> (= 17.0  (SETQ  X ( +  13  4)
 T
```


Prädikate und Strukturermittlung

Weitere Prädikate zu Vergleich

weitere Prädikate

(/= a1 a2 ... an)
(<= a1 a2 ... an)
(>= a1 a2 ... an)

} **Semantik analog zu oben**

Prädikate und Strukturermittlung

Artvergleiche

- **(NUMBERP Ausdr)**

  Wert T, falls Wert von Ausdruck numerisch ist, d.h. externe Zahlendarstellung besitzt, NIL sonst

```
-->(NUMBERP 'X)
NIL
-->(NUMBERP (+ 17 4))
T
```

- **(SYMBOLP Ausdr)**

  Wert T, falls Wert von Ausdruck ein Symbol ist
  NIL sonst

- **(STRINGP Ausdr)**

  Wert T, falls Wert von Ausdruck Zeichenkette als Wert hat
  NIL sonst

- **Beispiele:**

```
----> (SYMBOLP 3.0)        ----> (SYMBOLP 'AB)
NIL                        T
----> (STRINGP 'AB)        ----> (STRINGP "A0")
NIL                        T
```


Prädikate und Strukturermittlung

Komplexe Prädikate

- **Zusammengesetzt aus vordef. Prädikaten (Struktur- und Wertevergleich) und Junktoren AND, OR, NOT: spez. Formen**

- **(AND A1 A2 . . . An)**
  Wert: Anw. A_i
  NIL, falls ein A_i NIL, sonst Wert An
  n=0 Wert T

- **(OR A1 A2 . . . An)**
  Wert: Anw. A_i
  falls ein A_i Wert ≠ NIL, dieser Wert
  NIL sonst
  n=0 Wert NIL

- **(NOT A)**
  Wert: NIL, falls Wert von A ≠ NIL
  T sonst

n≥0

Auswertung von li nach re

Kurzschlußauswertung:
Gefahr Seiteneffekte, nicht durchgeführt

Prädikate und Strukturermittlung

Beispiele - 1

```
------► (AND (NUMBERP 3) (SETQ ACHTUNG 'SEITENEFF) (NULL 'T))
NIL

------► (AND (NUMBERP 3) (NULL 'T) (SETQ ACHTUNG 'K- SEITENEFF))
NIL

------► (AND)
T

------► (AND (NULL '() ) (MEMBER 'X3 '(X1 X2 X3 X4 )))
(X3 X4)

------► (OR (MEMBER 'X6 '(X1 X2 X3 )) (NULL '0))
NIL

------► (OR (MEMBER 'X3 '(X1 X2 X3 )) (NULL 'NIL))
(X3)

------► (OR (NULL 'NIL) (MEMBER 'X1 '(X1 X2 X3 )))
T

------► (OR)
NIL
```

Applikative Programmierung in LISP II

- **Bedingte Anweisung**
- **Anonyme Funktionen**
- **Bindung und Umgebung**
- **Variablenarten**
- **Funktionale Argumente**

Bedingte Anweisung

- **Mit COND können bedingte Anweisungen realisiert werden.**
  - (COND (T1 Programm-Stück1)
    (T2 PS-2)
    (Tn PS-n)) mit n>= 1
  - PS-i sind Folgen f1 .. fm von Formen, m>= 0
  - (Ti PS-i) heißen „Klauseln“
- **Auswertung:**
  - Auswertung der Tests in der vorgegebenen Reihenfolge bis ein Test-i - Wert ungleich NIL ergibt. In diesem Fall Auswertung des Rests der Klausel, COND-Auswertung beendet.
  - Der Wert des PS-i ist der Wert der COND-Form.
  - Sind die Werte aller Tests NIL, so ist der Wert der Bedingung NIL.
  - Ist das Programmstück der zutreffenden Klausel leer, so ist der Wert der Klausel der Wert des Tests.
- **Seiteneffekt:**
  - keine durch COND, aber durch Auswertung der T-i und PS-i möglich

Systemfunktionen
Boolesche/bedingte
Ausdrücke

Beispiele / Ausführung

- **Beispiele:**

```
(COND   ((MEMBER  EL LISTE)  LISTE)
        (T  (CONS  EL  LISTE) )
)
(COND   ((NULL  L)  NIL)
        ((EQUAL  EL  (CAR  L))  L)
        (T  (MEMBER  EL (CDR  L)))
 )
```

- **Ausführung:**

```
--->(SETQ L '(AB))
(AB)
--->(COND (( MEMBER (LAST '((AB) (CD)))  L)  'FALSCH)
          (( < (LENGTH (APPEND  '(L)  L)) 4)
                          (REVERSE  '(G I T H C I R))))
(RICHTIG)
```


Funktionen
und
Rekursion

Funktionen

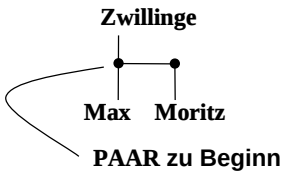
- **Klassifikation**
  - **Systemfunktionen**
    - spezielle Funktionen
    - normale Funktionen
  - **benutzerdefinierte Funktionen**
    - benannte Funktionen (DEFUN)
    - anonyme Funktionen (LAMBDA-Ausdruck)

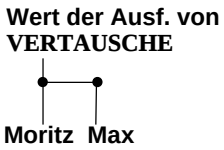
- **Wiederholung Benannte Funktionen**
  - **Definition**
    - `(DEFUN  FName  [Parlist]  [Rumpf] )`
      Auswertung: Wert ist der Funktionsname
  - **Aufruf**
    - `(FName` a_1 a_2 .. a_n `)` a_i Ausdrücke
      Auswertung: a_1, a_2 ... ausgewertet
      Bindung am Formalparameter
      Auswertung des Rumpfs

Funktionen: Beispiele

- **Beispiel:**

```
-->(DEFUN  VERTAUSCHE   (PAAR)
        (LIST (CADR  PAAR) (CAR  PAAR)))
VERTAUSCHE
-->(SETQ  ZWILLINGE  '(MAX   MORITZ))
(MAX  MORITZ)
-->(VERTAUSCHE  ZWILLINGE)
(MORITZ MAX)
```





- **Bemerkungen:**
  - Bindung (dynamisch) nur während Funktionsauswertung, aber s.u.
  - Neue Funktionsdefinition an bereits ex. Namen durch DEFUN möglich
    Systemfunktionen überschreibbar !
  - DEFUN ist spezielle Form: keiner ihrer Argumente wird ausgewertet

Unbenannte Funktionen, Lambda-Ausdrücke

- `(LAMBDA [Parlist] [Rumpf])` **analog zu DEFUN**
  - Definiert namenlose Funktion
  - Lambda-Ausdruck hat keinen Wert
  - `(Lambda-Ausdruck a1 a2 . . .an)` Auswertung wie bei DEFUN

- **Beispiel:**

```
----->(LAMBDA (PAAR)(LIST(CADR  PAAR)(CAR PAAR)))
   ; ist nicht auswertbar, jedoch
----->((LAMBDA (PAAR)(LIST(CADR PAAR)(CAR PAAR)))
            '(FRAU MANN))
(MANN  FRAU)
```

Funktionen und Rekursion

Bemerkungen: DEFUN, LAMBDA

- **DEFUN führt Namen ein**
  Aufruf und Definition beliebig weit auseinander
  ansonsten Ausführung gleich
  LAMBDA-Funktion nur an Stelle der Definition aufrufbar
- **LAMBDA historisch bedingt → Lambda-Kalkul (CHURCH 41)**
  besser AFUN für Anonymous Function
- **Auswertung eines Funktionsaufrufs heißt Lambda-Konversion**
- **Überall, wo Funktionsaufrufe stehen dürfen, dürfen auch Lambda-Ausdrücke stehen**

(Funktionsausdruck. a_1 a_n)

Name einer mit DEFUN definierten Funktion
oder Lambda-Ausdruck

Funktionen und Rekursion

Beispiele für Funktionsdeklarationen

- **Aufnehmen eines Objekts zu einer Liste, falls es noch nicht enthalten ist.**

```
(DEFUN OUR-ADJOIN (OBJEKT LISTE)
  (COND ((MEMBER  OBJEKT  LISTE)  LISTE)
        (T  (CONS  OBJEKT  LISTE))))
```

- **Erweiterung `OUR-ADJOIN`, so dass die Reihenfolge der Argumente beliebig sein darf:**

```
(DEFUN ANY-ORDER-ADJOIN (ARG-1 ARG-2)
  (COND ((LISTP  ARG-1)
         (OUR-ADJOIN  ARG-2  ARG-1))
        (T (OUR-ADJOIN  ARG-1  ARG-2))
))
```

Funktionen und Rekursion

Rekursion: Beispiel

- **Länge einer Liste:**

```
(DEFUN OUR-LENGTH(L)
  (COND ((NULL L) 0)          ; leere Liste
        (T (+1 (OUR-LENGTH (CDR L))))))
```

Laufzeitkeller auf der Listenhalde

- **Anzahl der Atome eines Ausdrucks:**

```
(DEFUN COUNT-ATOMS (L)
  (COND ((NULL L) 0)
        ((ATOM L) 1)
        (T (+ (COUNT-ATOMS (CAR L))
              (COUNT-ATOMS (CDR L))))))
```


Bindung

Parameterübergabe

- **Parameterübergabe bei Funktionsaufruf**

```
------> (SETQ FREI 3)
3

------> (DEFUN DEMO (GEBUNDEN)
          (SETQ GEBUNDEN (+ GEBUNDEN 10))
          (+ GEBUNDEN FREI))
```

- **Ausführung**

```
------> (DEMO  FREI)


------> FREI
```

Call by Value für Parameter-übergabe

Bindung

Bindung und Umgebung

- **Bindung**
  - Zuordnung von Symbolen zu Werten
- **Möglichkeiten**
  - Wertzuweisung mittels SETQ
  - Parameterbindung beim Funktionsaufruf (zeitlich begrenzt)
- **Umgebung**
  - Gesamtheit der zu einem Zeitpunkt zugänglichen Bindungen
  - Aktuelle Umgebung
- **Globale (Top-Level) Umgebung**
  - Umgebung zum Zeitpunkt des Systemstarts
  - Zuordnung zwischen Symbolen und Systemfunktionen
- **Globaler Wert eines Atoms**
  - Wert des Atoms in der globalen Umgebung

Bindung

Umgebung und Variablen

- **Definitionsumgebung einer Funktion**
  - Diejenige Umgebung, die zum Zeitpunkt der Definition aktuell ist.
- **Aufrufumgebung einer Funktion**
  - Diejenige Umgebung, die zum Zeitpunkt des Aufrufs aktuell ist.
- **Gebundene Variable**
  - Variable, die in der Parameterliste einer Funktion auftaucht.
  - Lokale Variable
- **Freie Variable**
  - Variable, die im Rumpf einer Funktion steht, aber keinen Parameter bezeichnet.

Bindung

Beispiel

```
→ (SETQ FREI  3)
3
→ (DEFUN DAMO (GEBUNDEN)
    (SETQ GEBUNDEN (+ GEBUNDEN 10))
    (+ FREI (SETQ FREI GEBUNDEN))) ; Seiteneff.
DAMO
→ (DAMO  FREI)

→ FREI
```


Bindung

Namengleichheit freier und gebundener Variablen

```
→(SETQ FREI 3)
3
→(DEFUN  DEMO1  (FREI)
    (SETQ  FREI (+ FREI 10))
    (+ FREI FREI))

DEMO1
→(DEMO1   2)
24
→FREI
3
```

- **Ablauf**
  - Zuerst Bindung FREI an 2, dann Bindung FREI an 12.
  - Nach Auswertung von DEMO1 gilt die alte Bindung wieder
  - Während der Auswertung von DEMO1 ist die alte Bindung verdeckt

Bindung

Beispiele: Bindung / Umgebung

```
⟶(DEFUN SUCCESSOR (N)
        (+ 1 N))
SUCCESSOR
⟶(SUCCESSOR 17)
18
⟶(SETQ N 10)
10
⟶(DEFUN FUNNY-SUCC (N)
        (SETQ A (+ 1 N))
        (AUX-SUCC))
FUNNY-SUCC
⟶(DEFUN AUX-SUCC ( )
        (+ 1 N))
AUX-SUCC
⟶(FUNNY-SUCC 17)
⟶A
```

Bindung N → 17 (s.u.) gilt nicht während der Auswertung von AUX-SUCC: dort ist N eine freie Variable

Bindung aus Def. Umg. von AUX-SUCC

COMMON-LISP : „statischer" Gültigkeitsbereich (Bindung)

alte LISP-Dialekte: dynamische Bindung

Bindung

LET-Ausdrücke und neue Bindungen

- **LET-Ausdruck**
  - führt neue lokale Variable ein
  - entspricht in etwa einem Block
- `(LET ((Par-1 Ausdr-1) (Par-n Ausdr-n)) ⋮ Rumpf )` **lok. Bindungen, mit denen der folg. Rumpf auszuwerten ist**

  `; Formen` $f_1 \ldots fm,\ m \geq 0$
- **Auswertung**
  - Auswertung der Ausdr-I
  - Bindung an Par-i „parallel"
  - danach Auswertung des Rumpfs
  - Wert des LET-Ausdrucks ist Wert der letzten Form, NIL für m=0
  - anschließend Aufhebung der Bindungen
  - keine Seiteneffekte durch LET, aber durch Auswertung von Ausdr-i bzw. fj

Bindung

LET vs. LAMBDA

- **LET-Ausdruck übersichtlicher als der entsprechende Lambda-Ausdruck**
  Im LISP-System werden LET-Ausdrücke auf LAMBDA-Ausdrücke zurückgeführt
- **((LAMBDA (Par-1 . . . Par-n)**
  Rumpf)
  Ausdr-1 . . . Ausdr-n)

  Bedeutung von LET-Ausdr. als funktionale Argumente (s.u.)
- **---▸(SETQ X 2)**
  2

  ---▸(LET ((X (+ 1 X))
  (Y (+ 2 X)))
  (LIST X Y))
  (3 4)

Bindung

Schachtelung und Gültigkeitsbereich

- **Schachtelung von LET-Ausdrücken bzw. Funktionen und Gültigkeitsbereich**
- ----▸
```
(LET ((X 2)
      (Z 5))
     (LET ((X (+ 1 X))
           (Y (+ 2 X)))
          (LIST X Y Z)))
(3 4 5)
```
- ----▸
```
(DEFUN FU1 (X Z)
  (DEFUN FU2 (X Y)
     (LIST X Y Z))
  (FU2 (+ 1 X)(+ 2 X)))
FU1
----▸ (FU1 2 5)
(3 4 5)
```

Funktionale Argumente

Funktionale Argumente

- **LISP- Funktionen**
  - haben die gleiche Gestalt wie LISP-Daten
  - können wie Daten behandelt und verarbeitet werden
  - insbesondere können Ausdrücke Funktionsbeschreibungen als Wert haben
-
```
(SETQ FN (COND((< X Y) 'ADJOIN)
               (T 'MEMBER)))
```
- **Funktionales Objekt**
  - Ein Ausdruck, dessen Wert eine Funktionsbeschreibung liefert.
  - Auswertung muss immer in einer Argumentposition geschehen
- **Auswertungsformen**
  - FUNCALL
  - APPLY

Funktionale Argumente

Aufruf mit verschiedenen Funktionen

- (FUNCALL Funkt-Obj a_1 . . . a_n)

Ausdruck mit Funktionsbeschreibung als Wert

Argumente für Funktionsaufruf

keine Seiteneffekte durch FUNCALL aber durch Ausw. von fO, a_i möglich

```
→(SETQ  L '(A B C)   X  3   Y  5)
5
→(LET ((FN (COND ((< X Y)'ADJOIN)
                 (T     'MEMBER))))
        (FUNCALL FN 'D  L))
(D A B C)
→(FUNCALL  'ADJOIN  'D  L)
(D A B C)
```

Funktionale Argumente

Auswertung durch APPLY

- **Auswertung funktionaler Objekte durch APPLY**
  - flexibler als FUNCALL
  - Argumente der Funktion einzeln oder zusammen in einer Liste

(APPLY fO a_1 . . . a_n a_{n+1})

Liste

falls n =0 alle Argumente in der Liste
Argument a_1, .. a_n + Elemente der Liste

- **Beispiel:**

```
→(APPLY  '+ (1 2 3 4 5 6))  ; Standardfall
21
→(APPLY  '+  1 2 3 '(4  5  6))
21
```


Was haben wir gelernt?

- **COMMON-LISP Ausschnitt für wertorientierte/applikative Programmierung**
- **LISP-Programm: Folge von auswertbaren Ausdrücken (Formen) mit oder ohne Seiteneffekt (Zuordnung von Variablen zu Atomen oder Listen)**
- **Interne Listenstruktur als spezielle Haldenstruktur, Auswertung durch rekursiven Abstieg (Compilation!), Gargabe Collection nötig, Listen in leftmost-son-right-sibling-Darstellung**
- **Auswertung oder Quotierung**
- **Listenoperationen: Aufbau mit CONS, Zusammenfügen APPEND, LIST**
  Zerlegen mit CAR, CDR, abgekürzte
  Zusammenfassung wie CADDR
  Spiegelung mit REVERSE
  Zugriff auf Teile mit LAST, MEMBER
- **Strukturprädikate ATOM, CONSP, NULL, LISTP, EQ, EQUAL, LENGTH, <, =, etc. Artvergleiche NUMBERP, SYMPOLP, STRINGP, TYPEP, n-äre Junktoren AND, OR, unäres NOT**
- **Bedingte „Anweisungen" COND**
- **Funktionen: Definition und Aufruf, unbenannte Funktionen mit LAMBDA-Ausdrücken, Rekursionshandhabung, Parameterübergabe, freie/gebundene Variable, Bindung und Umgebung, LET-Ausdrücke, Schachtelung, Gültigkeitsbereich, funktionale Argumente, funktionale Objekte**

Glossar

- **Atome, Listen, Funktionen, Funktionssymbole, Argumente**
- **Variable als Verweise auf Atome oder Listen, Listenaufbau aus CONS-Zellen**
- **Seiteneffekte als Änderung der Variablenzuordnung mit SETQ, Top-Level-Schleife, Interpretation als Systemfunktion, Compilation äquivalenten Codes wegen Effizienz**
- **Listenfunktionen, Listenprädikate (Unterscheidung nicht sauber), Strukturprädikat für Atome, Ausdrücke, Listen, Bedingungen, Klauseln**
- **DEFUN, LAMBDA, LET, APPLY**
- **Namensgleichheit, freie/gebundene Variable, Bindung, Umgebung, Schachtelung, Gültigkeit**
- **funktionale Argumente, Bindung verschiedener Funktionen**

Logische Programmierung in Prolog I

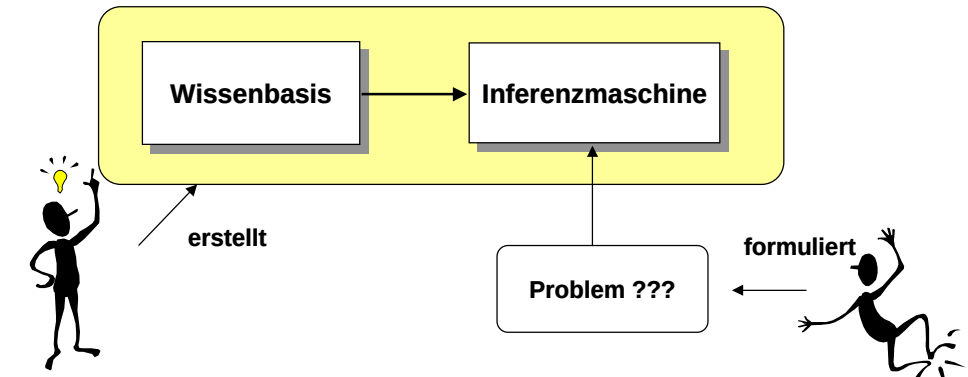
- **Programmiermodell**
- **Grundkonzepte**
- **Sprachelemente von Prolog**
- **Terminologie**

Literatur

- Bratko, I.
  Prolog programming for artificial intelligence. 2nd ed.
  Addison-Wesley, 1990.
- Sterling, L. & Shapiro, E.
  The art of Prolog. MIT Press, 1986.
- Clocksin, W.F. & Mellish, C.S.
  Programming in Prolog, 2nd Ed,
  Springer , 1984.
- Lloyd, J.W.
  Foundations of logic programming. 2nd Ed,
  Springer , 1987.

Einführung

Logikbasierte Programmierung



nicht-algorithmisches, sondern deklaratives Programmieren !!

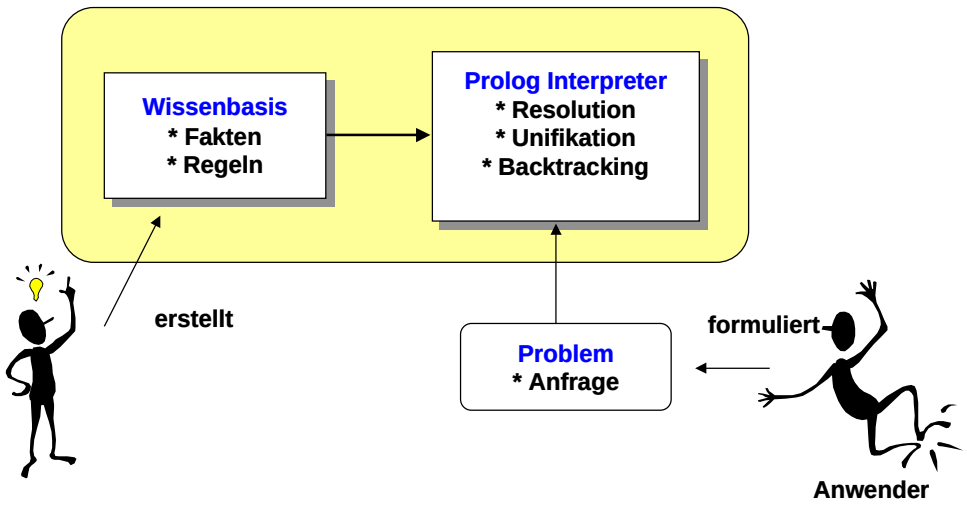
Einführung

Prolog - Einordnung

- **PROLOG**
  - steht für PROgramming in LOGic
  - Logik wird als Programmiersprache benutzt, basiert auf der **Prädikatenlogik**
  - Ist eine Programmiersprache für symbolische, nicht-numersiche Programmierung
  - Ist gut geeignet für Probleme, die Objekte und Relationen zw. Objekten behandeln
  - Syntax der Sprache ist eine modifizierte **Hornklausel-Notation**
- **Entwicklung**
  - Robert Kowalski (Edinburgh): theoretische Entwicklung
  - 1972: erste Implementierung von Colmerauer
  - Mitte der 70er Jahre : effiziente Implementierungen (Warren)
  - 1980: Sprache des japanischen "Fifth Generation Project"

Einführung

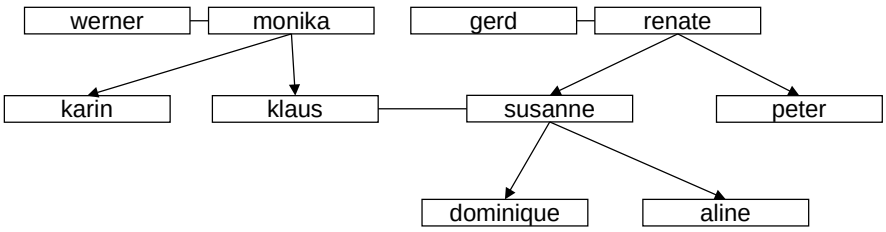
Prolog Programmiermodell



Einführung

Problemwelt

- **Verwandtschaftliche Beziehungen**
  - Wir wollen Fragen in dieser Problemwelt beantworten!
- **Beispiele**
  - "Wer ist die Mutter einer Person X"
  - "Wer ist Großvater einer Person Y"



Grund-konzepte

Tatsachen (Fakten)

Tatsachen

- sind Aussagen über **Objekte**, die als richtig angenommen werden.
- Sie beschreiben
  - Eigenschaften einzelner Objekten
  - Aussagen, die gleichzeitig mehrere Objekte betreffen (Relationen)

Beispiele:

```
weiblich(monika).        maennlich(werner).
weiblich(susanne).       maennlich(klaus).
weiblich(aline).         maennlich(dominique).
weiblich(karin).         maennlich(peter).
weiblich(renate).        maennlich(gerd).
```

```
verheiratet(werner, monika).
verheiratet(gerd, renate).
verheiratet(klaus, susanne).
```

Relationen sind gerichtet

```
istMuttervon(susanne, aline).
istMutterVon(susanne, dominique)
istMutterVon(monika, karin).
istMutterVon(monika, klaus).
istMuttervon(renate, peter).
istMuttervon(renate, susanne).
```


Grund-konzepte

Arbeitsweise von Prolog, Fragen

Prinzip

- Zuerst wird die **Wissensbasis** erstellt und dem Prolog-System mitgeteilt.
- Dann werden **Fragen** (goals, queries) an die Wissensbasis gestellt, deren Richtigkeit bewiesen werden soll.
- Prolog erzeugt die **Antwort**, indem es einen **logischen Beweis** auf der Basis des vorhandenen Wissens durchführt.

Beispiel:

```
?- maennlich(gerd).
yes

?- verheiratet(klaus, susanne).
yes

?- verheiratet(gerd, monika).
no
```

Grund-konzepte

Schlußfolgerungen, Regeln

- **Regeln dienen dazu, um aus bekanntem Wissen neues Wissen herzuleiten.**
- **Beispiel**
  - Vater-Kind-Beziehung
    - "Eine Person V ist Vater eines Kindes K, wenn er mit einer Frau F verheiratet ist und diese Mutter des Kindes K ist."
  -
```
istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```
- **Bemerkungen**
  - Alle Voraussetzungen stehen rechts in der Regel.
  - Diese werden durch Komma getrennt.
  - Variablen sind Platzhalter für beliebige konkrete Objekte

Grund-konzepte

Regeln

- **Regeln bestehen aus**
  - einem Bedingungsteil (rechte Seite, Rumpf)
  - und einem Folgerungsteil (linke Seite, Kopf)
- **Eine Regel p :- q, r kann gelesen werden**
  - **Deklarative** Bedeutung im Sinne von WAS
    - IF rechte Seite der Regel THEN linke Seite der Regel
  - **Prozedurale** Bedeutung im Sinne von WIE
    - Um ein Ziel p zu lösen, müssen zuerst die Unterziele q und r gelöst werden
- **Prolog wendet die Regeln rückwärts an**
  - d.h. Prolog startet mit der linken Seite
  - Um zu zeigen, dass die linke Seite gilt, muss gezeigt werden, dass die rechte Seite gilt.
  - backward chaining

Grund-konzepte

Variablen in Fragen

```
mag(georg, karin).
mag(georg, peter).
mag(georg, susi).
mag(susi, wein).
mag(karin, gin).
mag(susi, karin).
mag(susi, peter).

gemFreund(A,B,C) :- mag(A, C),
                    mag(B, C).
```

?- mag (karin, gin).
YES

?- mag(susi, WAS).
WAS = wein ;
WAS = karin

?- gemFreund (georg, susi, WER).
WER = karin ;
WER = peter ;
NO

?- mag (susi, susi)
NO

Es ist nicht beweisbar!

- **Prolog antwortet**
  - mit einer ***Instanziierung*** der Variablen in der Frage
  - ***Weitere*** Instanziierungen können durch ; abgerufen werden

Grund-konzepte

Rekursive Definition von Regeln

- **Prolog erlaubt die rekursive Definition von Regeln**
  - Rekursion ist eine wichtige Programmiertechnik in Prolog
- **Beispiel:**
  - Definition der Vorfahre-Relation mit Hilfe der Relation `elternteil`
  - **Fall1:**
    Für alle X und Z:
    - X ist Vorfahre von Z falls X Elternteil von Z ist
    - `vorfahre(Vorf,X):- elternteil(Vorf,X).`
  - **Fall2:**
    Für alle X und Z:
    - X ist Vorfahre von Z falls es ein Y gibt mit
    - (1) X ist Elternteil von Y und
    - (2) Y ist Vorfahre von Z.
    - ```
      vorfahre(Vorf,X):- elternteil(Vorf,Y),
                         vorfahre(Y,X).
      ```
      ```

Grund-konzepte

Rekursive Definition von Regeln

Beispiel:

- `vorfahre(Vorf,X):- elternteil(Vorf,X).`
- `vorfahre(Vorf,X):- elternteil(Vorf,Y), vorfahre(Y,X).`

```
?-vorfahre(X,aline).
X = klaus;
X = susanne;
X = werner;
X = monika,
X = gerd;
X = renate;
```


Grund-konzepte

Prolog-Programme

- **Prolog-Programme bestehen aus *Klauseln***
  - Fakten und Regeln

```
weiblich(monika).
weiblich(susanne).
weiblich(aline).
weiblich(karin).

verheiratet(werner, monika).
verheiratet(gerd, renate).

istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```

- **Anmerkungen**
  - Jede Klausel wird mit einem **Punkt** abgeschlossen.
  - Jede der ersten 4 Klauseln drückt ein **Fakt** über die `weiblich`-Relation aus (Prädikat)
  - Die Anzahl der Argumente wird als **Stelligkeit** der Relation (des Prädikats) bezeichnet.

Grund-konzepte

Universelle Fakten

- **Variablen in Fakten**
  - `mag(susi,X).` „Susi mag alles“
  - `mag(X,X).` „Jedes mag sich selbst“
  - `mag(X,Y).` „Jedes mag alles“
- **Anmerkungen**
  - Gleiche Variable in gleichem Fakt: Gleichheit
  - Gleiche Variable in verschiedenen Fakten: Nichtgleichheit
  - Variablen in Fragen sind existenzquantifiziert:
    `?-mag (susi,X).` **Existiert** etwas, das Susi mag“
  - Variablen in Fakten sind allquantifiziert:
    `mag (susi,X).` „Susi mag **alles**“

Grund-konzepte

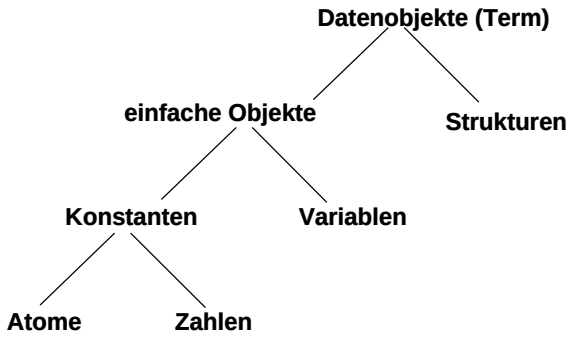
Konjunktiv verknüpfte Fragen

- **Prolog kann verschiedene Fragen kombinieren**
  - Beispiel: Frage: "Wer ist Großvater von Aline?“
  - Vorgehen:
    - (1) Wer ist Vater von Aline? Annahme, es gibt ein Y.
    - (2) Wer ist Vater von Y? Annahme, es gibt ein X.
- **Frage:**
  -
```
?- istVaterVon(Y,aline), istVaterVon(X,Y).
Y = klaus
X = werner
```
- **Ordnung der Teilfragen kann geändert werden, ohne dass sich das Ergebnis ändert.**
- **Beispiel:**
  -
```
?- istVaterVon(X,Y), istVaterVon(Y,aline)
X = werner
Y = klaus
```

Syntax

Objekte in Prolog

- **Objekte**
  - repräsentieren Dinge der Vorstellungswelt oder sind Abstraktionen von realen Gegenständen.
- **Prolog kennt folgende Datenobjekte**
  - Prolog erkennt den Typ eines Objekts an seiner syntaktischen Struktur



Syntax

Atome und Zahlen

- **Atome sind**
  - Zeichenketten aus Buchstaben, Zahlen und dem Zeichen _
  - Atome beginnen mit einem Kleinbuchstaben

    z.B. `renate` `x_25` `frau_holle` `peter`

  - Zeichenketten aus speziellen Zeichen,
    z.B. `<--->` `===>`
  - in einfache Anführungszeichen eingeschlossene Zeichenketten
    z.B. `'Frau Holle'`
  - Spezielle Zeichen: `,` `;` `!` `[ ]`
- **Zahlen in Prolog sind Integer- oder Real-Zahlen, wobei**
  - Real-Zahlen einen Dezimalpunkt enthalten,
    z.B. 2.03
  - Integer-Zahlen nicht,
    z.B. -22, 3, 6

Syntax

Objekte in Prolog

- **Objekte**
  - repräsentieren Dinge der **Vorstellungswelt** oder sind Abstraktionen von realen **Gegenständen**.
- **Terme beschreiben Objekte**
  - **Konstanten** und **Variablen** sind Terme,
  - Ist f ein Funktionssymbol (Funktor) und sind t_1 , ..., t_n Terme, so ist auch $f(t_1, ..., t_n)$ ein Term (**Struktur**).
  - Ein Term heißt **Grundterm**, wenn er keine Variablen enthält.
- **Konstante**
  - Zahlen
  - Atome

Syntax

Variablen

- **Bedeutung von Variablen in Prolog**
  - In prozeduralen Sprachen repräsentieren Variablen Speicherzellen des Rechners. Variablen kann ein Wert zugewiesen werden.
  - In Prolog ist eine Variable ein **Platzhalter** für ein Prolog-Objekt
    - Anstelle einer Variablen kann ein **beliebiges Prolog-Objekt** eingesetzt werden.
    - Bspl: `mag(susi, WAS)`
- **Syntax**
  - Bezeichner einer Variablen beginnt mit Großbuchstabe oder mit Unterstrich "_"
- **Anonyme Variable**
  - Drückt aus, dass an **einer Stelle** ein beliebiges Objekt eingesetzt werden kann und wir uns nicht dafür interessieren.
  - `istEhemann(Person) :- verheiratet(Person, _ )`

Syntax

Strukturen

- **Eine Struktur repräsentiert ein Objekt, das aus mehreren Objekten zusammengesetzt ist.**
- **Aufbau**
  - **Name** der Struktur (Funktor)
  - **Komponenten** der Struktur werden geklammert und durch Komma getrennt
  - Funktor muss ein Atom sein
  - Komponenten können Konstanten oder wiederum Strukturen sein.
- **Beispiele**
  - `datum (24, 11, 1999)`
  - `name (herr, max, mueller)`
  -
```
geboren(name (herr, max, mueller),
        datum (24, 11, 1999))
```


Syntax

Beispiel Datenstruktur: Terme und Fragen

```
buch('The Art of Prolog',author (sterling,shapiro)).
buch('Programming in Prolog',author(clocksin,mellish)).
buch('Foundations of LogicProgramming',author(lloyd)).


?-buch(X,Y).
X= 'The Art of Prolog' Y=author (sterling, shapiro);
X= 'Programming in Prolog' Y=author(clocksin,mellish);
X= 'Foundations of LogicProgramming' Y=author(lloyd);
no

?-buch(X,author(Y,Z)).
X= 'The Art of Prolog' Y=sterling Z=shapiro;
X= 'Programming in Prolog' Y=clocksin Z=mellish;
no
```

Konzepte und Terminologie

Definitionen nach Sterling/Shapiro: Programm, Regel, Prädikat

- Ein **Logikprogramm** ist eine endliche Regelmenge.
- Statt Regel sagt man auch **Horn-Klausel** oder bei Eindeutigkeit auch einfach nur **Klausel.**
- Eine **Regel** hat die Form.

  $A <- B_1, B_2 , ... B_n.$ mit $n >= 0$

  A ist der **Regelkopf** und die B_i's sind der **Regelrumpf.**
  Eine Regel mit $n=0$ wird **Fakt** genannt. A und B_i's werden auch Ziele genannt.
- Ein **Ziel** hat die Form eines Prädikats (Prädikatenlogik 1. Stufe)

  $pred (t_1, t_2, ..., t_m)$ mit $m >=0$

  wobei pred Prädikatnamen, m Stelligkeit und die tj Argumente.
  Prädikate beschreiben Objekte und Beziehungen zwischen diesen Objekten.
- Argumente für Prädikate sind **Terme**; induktiv definiert:
  - eine Konstante ist ein Term,
  - eine Variable ist ein Term,
  - sind $t_1, t_2, ..., t_k$ Terme und ist f ein Funktionssymbol, so ist auch $f (t_1, t_2, ..., t_k)$ ein Term.

Konzepte und Terminologie

Semantik von Regeln und Programmen

- **Konventionen:**
  - hans, h1 Konstante,
  - X, Haus Variable,
  - _ anonyme Variable,
  - mag Prädikatname, plus Funktionssymbol
- **Bedeutung von Regel A <- B1, B2 , ..., Bn:**
  - B1 & B2 & ... Bn - >A,
  - umgangssprachlich „Wenn B1 gilt und B2 gilt und ... und Bn gilt, dann gilt auch A“.
- **Prozedurale Interpretation einer Regel:**
  - „Um A zu beantworten, beantworte B1 und B2 und ... und Bn“.
  - Diese Interpretation ist die Grundlage von PROLOG–Systemen.
- **Eine Frage definiert ein Ziel oder eine Konjunktion von Zielen. Die Ausführung des Programms hat zu zeigen, ob alle Ziele erfüllt werden. Ein Ziel ist erfüllt, wenn:**
  - entweder ein „passendes“ Fakt existiert,
  - oder ein „passender“ Regelkopf existiert und jedes Ziel im Regelrumpf erfüllt wird.

Logische Programmierung in Prolog II

- **Termgleichheit**
- **Beweisstrategie**
- **Listen in Prolog**
- **Arithmetik**

Unifikation

- **Problem**
  - Wird eine Frage gestellt, so muss das Prolog-System entscheiden, ob der Term der Frage zu einem Term der Wissensbasis gleich ist oder nicht.
  - Nach Einführung von Termen mit Funktionssymbolen ist ein Mechanismus zum Auffinden „passender“ Regelköpfe zu definieren.
- **Prolog benutzt die *Unifikation*, um zu prüfen, ob zwei Terme miteinander übereinstimmen**
  - Zwei Terme sind ***unifizierbar***, falls es eine Substitution gibt, die die Terme gleich macht
  - Diese Substitution wird ***Unifikator*** genannt.
- **Beispiele**
  - t1 = `datum(D,M,1983)`, t2 = `datum(D1,may,Y1)`

    Die Substitution S = {D = D1,M = may, Y1= 1983} unifiziert t1 und t2

Termgleichheit

Regeln für Termgleichheit

- **Zahlen und Atome**
  - sind nur zu sich selbst gleich
- **Variable**
  - ist gleich zu einer anderen Variablen, wenn die Namen gleich sind.
  - Variablen unifizieren mit jedem Term.
  - Wird eine Variable durch einen Term substituiert, dann ist diese mit dem Term instanziiert.
  - Alle Vorkommen der Variablen werden durch den Term ersetzt.
- **Strukturen**
  - Zwei Strukturen sind gleich, wenn
    - sie den gleichen Funktor haben
    - die gleiche Anzahl von Komponenten haben
    - die entsprechenden Komponenten der ersten und zweiten Struktur paarweise gleich sind.

Termgleichheit

Beispiele Unifikation

- **Beispiele**
  - T1 = f (X, a(b, c))
    T2 = f(d, a (Z, c))
  - T1 = f(X, a(b, c))
    T2 = f(Z, a(Z, c))
  - T1 = f(c, a(b, c))
    T2 = f(Z, a(Z, c))
  - T1 = g(Z, f(A, 17, B), A+B, 17)
    T2 = g(C, f(D, D, E), C, E)

Termgleichheit

Substitution

- **Substitution**
  - Eine Substitution ist eine endliche Menge von Paaren der Form X=t
  - X ist eine Variable, t ein Term
  - Es gilt: Keine zwei Paare haben dieselbe Variable als linke Seite
- **Instanz eines Terms**
  - Für einen Term t und seine Substitution S ={X1 = t1 , ... Xn = tn} bezeichnet tS das Ergebnis der simultanen Ersetzung aller Vorkommen aller Xi durch ti (1<=i<=n).
  - tS heißt Instanz von t.
- **Anmerkungen**
  - Ein Ziel und ein Fakt (ein Regelkopf) „passen“, wenn sie eine gemeinsame Instanz besitzen.
  - Eine Instanz heißt Grund-Instanz, wenn sie keine Variablen enthält.
  - Die Berechnung eines PROLOG-Programms liefert, ausgehend von der Frage, die Substitutionen, für die alle Ziele der Frage erfüllt sind.

Termgleichheit

Unifikator

- **Unifikator**
  - Ein Unifikator zweier Terme ist eine Substitution, die die beiden Terme identisch macht, also eine Instanz erzeugt, die Instanz beider Terme ist. In diesem Fall sagt man, dass die beiden Terme unifizierbar sind.
- **Beispiel:**

p(X,	f(g(c,d),	Y),	Z,	W)
p(g(c,d),	f(X,	b),	b,	V)

haben als Unifikator die Substitution

S = { X/g(c,d), Y/b, Z/b, W/V }

das heißt, es gilt:

S(T1) = S(T2)

Termgleichheit

Allgemeinster Unifikator

- **Eine Instanziierung, mit der die Terme gleich werden, kann allgemeiner sein als eine andere**
  - z.B. für `date(D,M,1983)` und `date(D1,may,Y1)` ist die Instanziierung
    - S1 = {D /D1, M/ may, Y1/1983}
  - allgemeiner als die Instanziierung
    - S2 = {D/3 , M/may, Y1/1983}
  - da nicht alle Komponenten festgelegt sind.
  - Term t_1 ist allgemeiner als Term t_2, wenn t_2 eine Instanz von t_1 ist.
- **Der Allgemeinste Unifikator**
  - für zwei Terme ist ein Unifikator, der die allgemeinste Instanz erzeugt.
- **Satz**
  - Wenn zwei Terme unifizierbar sind, dann existiert ein eindeutiger allgemeinster Unifikator, ggfs. in alphabetischen Varianten (Umbenennung von Variablen).

Termgleichheit

Algorithmus MGU

```
Eingabe: Terme t1 und t2
Ausgabe: MGU oder "nicht unifizierbar"

IF t1 und t2 gleiche Variablen THEN  U = {}

IF t1 eine Variable THEN
  IF t2 enthält nicht t1 THEN  U = { t1/t2 }
  ELSE nicht unifizierbar

IF t2 eine Variable THEN
  IF t1 enthält nicht t2 THEN  U = { t2/t1 }
  ELSE nicht unifizierbar

IF t1 und t2 sind Konstanten THEN
  IF t1 = t2 THEN              U = {}
  ELSE nicht unifizierbar

IF t1 und t2 Strukturen THEN
  IF Funktoren und Anzahl Komponenten sind gleich THEN
    Sei U1 = MGU (t11, t21)
    FOR alle Komponentenpaare t1i, t2i 2<=I<=n DO
      Ui := MGU( Ui-1(..(U1(t1i))..), Ui-1(..(U1(t2i))..) )
    IF alle Ui existieren THEN      U = Un(Un-1(..(U1))..)
    ELSE nicht unifizierbar
ELSE nicht unifizierbar
```

Beispiele MGU

- **Beispiel**
  - T1 = g(f(1), h(X))
    T2 = g(Y, Y)
  - T1 = p(1, A, f(g(X)))
    T2 = p(X, f(Y), f(Y))
  - T1 = p(X, f(g(c, d), Y), Z, W)
    T2 = p(g(c, d), f(X, b), V, h(X,Z))

Variablen

- **Variablen in Prolog werden anders gebraucht als in herkömmlichen Programmiersprachen**
  - Variablen referenzieren Objekte, nicht auf Speicherplätze
  - sie werden bei der Unifikation instanziiert
- **Der Operator = bedeutet nicht Zuweisung, sondern Unifikation**
  Bsp:
  - X = 3. bedeutet, dass die Variable X unifiziert wird mit der Konstante 3 und dabei mit 3 instanziiert wird.
  - = kann nicht verwendet werden, um den Wert einer Variable zu erhöhen
    d.h. X = X + 1. erzeugt einen Fehler

Termgleichheit

Gleichheit von Termen

- **Gleichheit**
  - von Termen bedeutet, dass diese unifizierbar sind
- **Systemprädikat =**
  - = (X,X)
  - dieses Faktum ist jedem Prolog-System bekannt
  - Es ist nicht erlaubt, neue Klauseln zu diesem Prädikat hinzuzufügen
  - Schreibweise:
    - = (t1, t2)
    - t1 = t2
  - Bedeutung;
    - t1= t2 ist dann beweisbar, wenn die Terme unifizierbar sind
- **Beispiel**
  - 3+2=5 ist nicht beweisbar
  - weiblich(susanne) = weiblich(X) ist beweisbar

Beweisstrategie

Resolutionsprinzip - 1

- **Prolog versucht eine Anfrage auf der Menge der vorhandenen Fakten und Klauseln zu beweisen.**
  - Ein Faktum ist eine beweisbare Aussage
  - Es gilt
    - wenn jedes der Literale L1 , L2, L3, …, Ln beweisbar ist
    - und es eine Regel L :- L1 , L2, L3, …, Ln gibt
    - dann ist auch das Literal L beweisbar.
  - Beispiel:

```
istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```

  - Modus Ponens (Abtrennregel)
- **Resolutionsprinzip**
  - ist die Umkehrung des Modus Ponens
  - liefert ein Verfahren, um zu zeigen, ob eine Aussage beweisbar ist oder nicht.

Beweisstrategie

Resolutionsprinzip - 2

- **Eine Aussage ist beweisbar,**
  - wenn sie ein Faktum ist
  - wenn mindestens eine Regel existiert, deren linke Seite identisch mit der zu beweisenden Aussage ist
  - und deren Literale der rechten Seite alle beweisbar sind.
- **Beispiel**
  - `istVaterVon(klaus, aline)` V/klaus, K/aline
    wird zurückgeführt auf
  - `verheiratet(klaus,F), istMutterVon(F,aline)` F/susanne
    wird zurückgeführt auf
  - `verheiratet(klaus,susanne)` ist ein Faktum, bewiesen
  - `istMutterVon(F,aline)` ist ein Faktum, bewiesen
    wird zurückgeführt auf
  - leere Aussage

Beweisstrategie

Resolutionsprinzip - 3

- **Satz**
  - Kann eine Aussage in mehreren Schritten unter Verwendung des Resolutionsprinzips auf die leere Aussage zurückgeführt werden, dann ist diese Aussage beweisbar.
- **Folgende Ergebnisse sind möglich**
  - yes
    - Die gegebene Aussage ist aufgrund der Fakten und Regeln beweisbar.
  - No
    - Die gegebene Aussage ist aufgrund der Fakten und Regeln nicht beweisbar. Möglicherweise ist die Aussage korrekt, nur sind die Regeln und Fakten unvollständig.
  - hält nicht an
    - Die Aussage könnte aufgrund der Regeln und Fakten beweisbar sein.

Beweisstrategie

- **Resolution ist keine eindeutige Vorschrift**
  - Welches Literal einer Anfrage soll als nächstes bewiesen werden?
  - Durch welche Klausel soll ein Literal einer Anfrage abgeleitet werden?
- **Strategie**
  - A) Die Literale einer Anfrage werden von links nach rechts abgeleitet.
  - B) Die Klauseln werden von "oben" nach "unten" nach einer passenden durchsucht.
  - C) Falls das Prolog-System beim Beweisen in eine Sackgasse läuft, wird der letzte Ableitungsschritt rückgängig gemacht und die nächste passende Klausel zu einem neuen Ableitungsschritt verwendet (backtracking).

    Bem backtracking müssen Bindungen von Variablen, die beim Beweisschritt stattgefunden haben, rückgängig gemacht werden, damit neue Variablenbindung im alternativen Beweisschritt versucht werden kann.

Backtracking -Beispiel - 1

```
verheiratet(werner, monika).
verheiratet(gerd, renate).
verheiratet(klaus, susanne).

istMuttervon(susanne, aline).
istMutterVon(susanne, dominique)
istMutterVon(monika, karin).
istMutterVon(monika, klaus).
istMuttervon(renate, peter).
istMuttervon(renate, susanne).

istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```

Beweisstrategie

Backtracking -Beispiel - 2

istVaterVon(X,aline)

```
istVaterVon(X,aline) :- verheiratet(X,F),    K/aline, V/X
                        istMutterVon(F,aline).


verheiratet(X,F)                          X/werner , F/monika
istMutterVon(monika, aline)               FAIL


verheiratet(X,F)                          X/gerd , F/renate
istMutterVon(renate, aline)               FAIL


verheiratet(X,F)                          X/klaus , F/susanne
istMutterVon(susanne, aline)
```

```
X=klaus
```


Listen

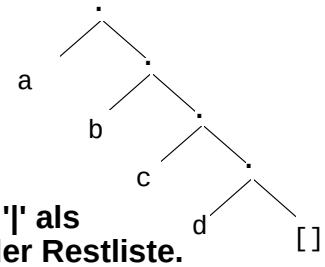
Listen in Prolog

- **Eine Liste ist eine Folge von Objekten.**
  - Die Anzahl der Objekte einer Liste ist nicht festgelegt.
- **Eine Liste in Prolog ist entweder**
  - die leere Liste (dargestellt durch das Atom [])
  - die Struktur mit dem Funktor **.** und zwei Komponenten
    - die erste Komponente wird head genannt
    - die zweite Komponente ist wiederum eine Liste (tail)
- **Beispiele für Listen:**
  - []
  - . (10, [])
  - . (x, . (y, []))
  - . (a, . (b, . (c, [])))

Listen

Listen repräsentiert als Bäume

- **Listen können als Bäume repräsentiert werden.**
  - . (a, . (b, . (c, .(d, [])))) Listennotation
  - Die Elemente werden durch Komma getrennt und mit eckigen Klammern umschlossen.
  - [a, b, c, d]



- **Prolog verwendet den vertikalen Strich '|' als Trennsymbol zw. dem Listenkopf und der Restliste.**
- **alternative** Repräsentationen können verwendet werden:
  - [elem1,elem2,elem3] oder
    [head | tail] oder
    [elem1 , elem2, … | rest] oder …
  - z.B. [a,b,c] = [a | b,c] = [a , b | c] = [a,b,c, []]

Listen

Listen in Prolog

- **Unifikation**
  - Listen sind spezielle Terme
  - Länge der Liste entspricht der Stelligkeit
  - Listen können miteinander unifiziert werden.
- von Listen ergibt:

L1	L2
[a,b,c]	[X,Y,Z]
[a,b,c]	[X|Y]
[a]	[X|Y]
[[a, Y] | Z]	[[X, b], [c, d]]
[X, Y, X]	[a, Z, Z]
[[X], [Y], [X]]	[[a], [Z], [Z]]

Operationen auf Listen: member

- **Prolog-Prädikate (Prozeduren) werden oft durch**
  - einen Basisfall und
  - einen einfachen rekursiven Fall definiert:
- **Beispiel:** die Relation member
  - Ein Element ist in einer Liste, falls
    - es das erste Listenelement ist **(Basisfall)** oder
    - es in der Restliste ist **(rekursiver Fall)**
-
```
member(X,[X|_]).

member(X,[_|Y]) :- member(X,Y).
```


Operationen auf Listen: member

```
?- member(1,[1,2,3]).
yes
```

```
member(X,[X|_]).
member(X,[_|Y]):- member(X,Y).
```

```
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

```
?- member(1,[a,b]).
   (3) 0 Call: member(1,[a,b]) ?
   (4) 1 Call: member(1,[b]) ?
   (5) 2 Call: member(1,[]) ?
   (5) 2 Fail: member(1,[]) ?
   (4) 1 Fail: member(1,[b]) ?
   (3) 0 Fail: member(1,[a,b]) ?
no
```

Listen

Operationen auf Listen: member

```
?- member(X,[a,c]),member(X,[b,c]).
   (5) 0 Call: member(\_35,[a,c]) ?
   (5) 0 Exit: member(a,[a,c]) ?
   (6) 0 Call: member(a,[b,c]) ?
   (7) 1 Call: member(a,[c]) ?
   (8) 2 Call: member(a,[]) ?
   (8) 2 Fail: member(a,[]) ?
   (7) 1 Fail: member(a,[c]) ?
   (6) 0 Fail: member(a,[b,c]) ?
   (5) 0 Redo: member(a,[a,c]) ?
   (9) 1 Call: member(\_35,[c]) ?
   (9) 1 Exit: member(c,[c]) ?
   (5) 0 Exit: member(c,[a,c]) ?
   (10) 0 Call: member(c,[b,c]) ?
   (11) 1 Call: member(c,[c]) ?
   (11) 1 Exit: member(c,[c]) ?
   (10) 0 Exit: member(c,[b,c]) ?
X = c
```

```
member(X,[X|_]).

member(X,[_|Y]):-
   member(X,Y).
```


Listen

Operationen auf Listen: append

Beispiele:

append([a,b],[c,d],[a,b,c,d]). append([a,b,d],[c],[a,b,d,c]).

append([],[c,d],[c,d]).

- **Idee:** Unterscheidung nach dem ersten Argument von append
  - **Basisfall:** falls das erste Argument von append die **leere Liste** ist, dann enthält das dritte Argument die gleiche Liste wie das zweite Argument
  - **rekursiver Fall:** falls das erste Argument von append **nicht** die leere Liste ist, dann muß es die Form **[X|L1]** haben.

Daher hat das dritte Argument die Form [X|L3] , wobei L3 die aneinander gehängten Listen L1 und L2 enthält.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Listen

Operationen auf Listen: append

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

?- append([a,b],[c,d],L3).
L3 = [a,b,c,d]

?- append(L1,L2,[a,b]).
L1 = [],
L2 = [a,b] ;
L1 = [a],
L2 = [b] ;
L1 = [a,b],
L2 = [] ;
no
```


Arithmetik

Operatoren & Arithmetik

- **Prolog erlaubt die Verwendung von Operatoren.**
- **Operatoren werden in Infix-Notation geschrieben.**
  - z.B. 1+3 statt +(1,3)
- **Grundlegende arithmetische Operatoren sind**

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
mod	Modulo

- **Vergleichsoperatoren**

X > Y	X ist größer als Y
X >= Y	X ist größer als oder gleich Y
X < Y	X ist kleiner als Y
X =< Y	X ist kleiner als oder gleich Y
X =:= Y	X ist gleich Y
X =\= Y	X ist ungleich Y

Arithmetik

Arithmetik: 'is' und '='

- **Der Operator =**
  - Er bedeutet, daß Unifikation verwendet wird.
  - Die Unifikation gelingt, wenn der Term auf der rechten Seite von = mit dem Term der linken Seite unifiziert werden kann.
    ?- X = 3+1.
    X = 3+1.
    ?- 4 = 3+1.
    no
    ?- 3+1 = 3+1.
    yes
- **Der Operator is**
  - ruft die arithmetische Evaluierung der rechten Seite auf und versucht, das Ergebnis mit der linken Seite zu unifizieren.
    ?- X is 3+1.
    X = 4
    ?- 4 is 3+1.
    yes
    ?- 3+1 is 3+1.
    no

Arithmetik

Arithmetik

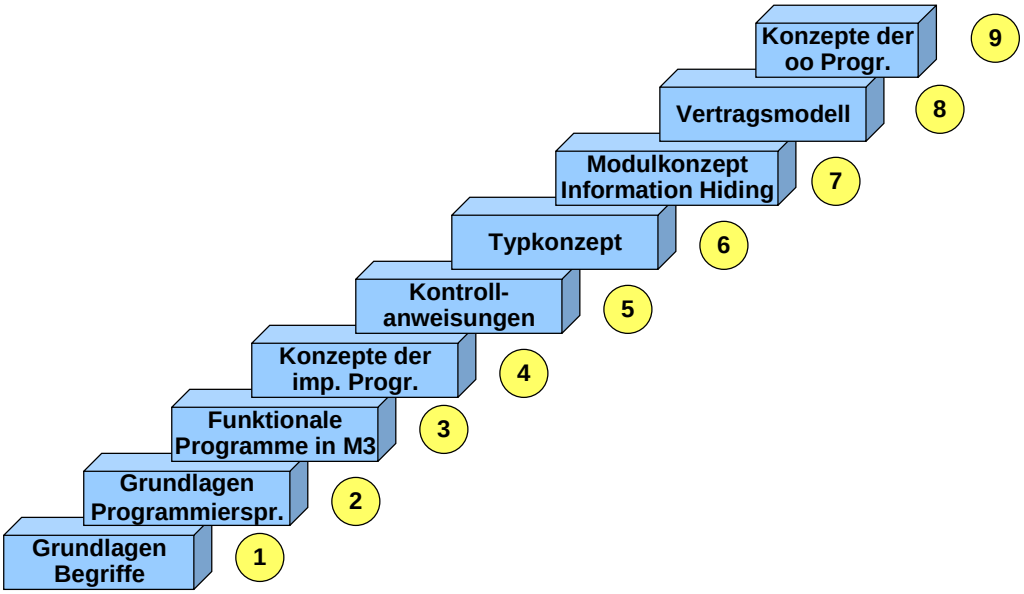
- **Beispiel: Länge einer Liste length(L1,N),d.h.**
  - falls L1 leer ist,dann hat N den Wert 0,
  - sonst ist N gleich der Länge des Tails von L1 + 1
- **Prädikat length**
  - `length([],0).`
  - `length([H|T],N1) :- length1(T,N),`
    `N1 is N + 1.`
- **Beispiele**
  - `?- length([a, b, c, d], X).`
    `X = 4`
  - `?- length([a,b], X), Y is 4 + X.`
    `X = 2`
    `Y = 6`

Deklarative vs. Prozedurale Interpretation

- **Deklarative Bedeutung**
  - Sie bezieht sich auf die logischen Relationen, die durch das Programm beschrieben werden, d.h. WAS ist das Ergebnis des Programms
- **Prozedurale Bedeutung**
  - Sie bezieht sich darauf, WIE das Ergebnis berechnet wird.
- **Deklarative Programmierung ist einer der Vorteile von Prolog, denn die Idee ist einfach:**
  - Gebe Relationen an, die für eine Applikation gelten müssen, und kümmere dich nicht darum, WIE die Lösung gefunden wird.
- **Aber: der prozedurale Aspekt ist wichtig,**
  - z.B. die Regel p:-p. ist deklarativ korrekt, aber prozedural sinnlos, da sie eine endlose Schleife bewirkt.

Zusammenfassung
Teile I - IV

Aufbau der Vorlesung - Teile I - IV



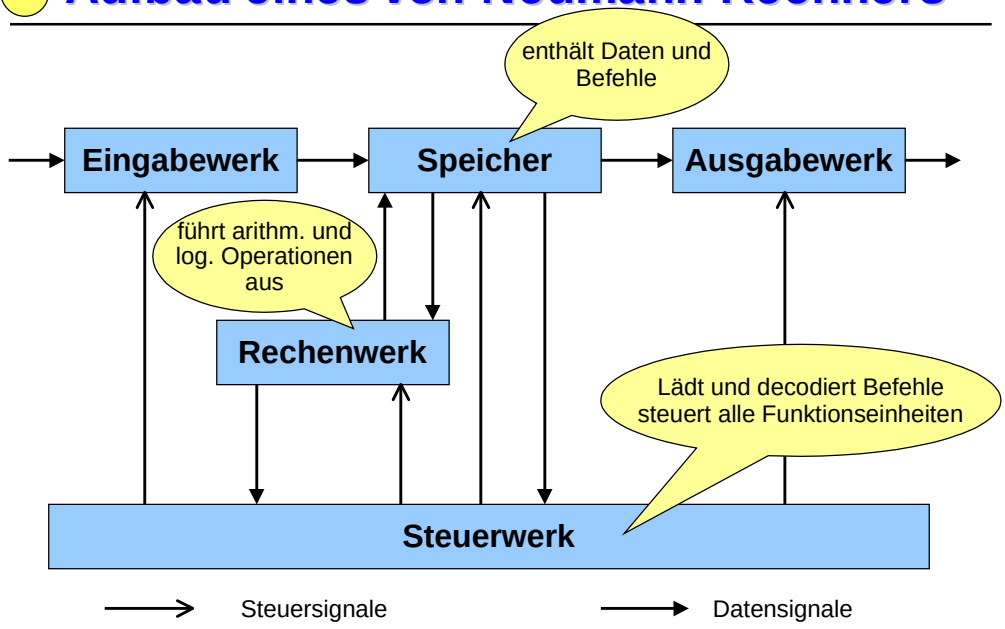
1

Begriffe

- **Ein Algorithmus ist ein Verfahren, welches**
  - in einem ***endlichen*** Text niedergelegt werden muß
  - ***effektiv*** ausführbar ist,
  - durch eine *Maschine* ***ausgeführt*** werden kann
  - Anzahl und Ausführungszeit der Elementaroperationen sind beschränkt
  - Ein Algorithmus wird entsprechend seiner Vorschrift schrittweise ausgeführt
- **Definition: Software**
  - "Computer ***programs***, ***procedures***, and possibly associated ***documentation*** and ***data*** pertaining to the operation of a computer system."
- **Definition: Programm**
  - "A combination of ***computer instructions*** and ***data definitions*** that enable computer hardware to ***perform*** computational or control functions".
- **Unter dem Begriff Programmieren versteht man**
  - das **Lösen von Problemen** unter Zuhilfenahme eines **Rechners**

1

Aufbau eines von-Neumann-Rechners



2

Alphabet, Wort, Formale Sprache

Alphabet

- Ein Alphabet ist eine ***nichtleere endliche*** Menge von ***unterscheidbaren*** Zeichen ("Buchstaben")
  A = {a_1, a_2, a_3, …} mit einer ***Ordnungsrelation*** ≤ ($a_1 \leq a_2 \leq a_3$ ….)

Wort über einem Alphabet

- ***endliche Folge*** von Buchstaben, die auch ***leer*** sein kann (ε leere Wort)
- A* bezeichnet die ***Menge aller Wörter*** über dem Alphabet A (inkl. dem leeren Wort).

Formale Sprache

- Sei A ein Alphabet. Eine (formale) Sprache (über A) ist ***jede beliebige Teilmenge von A****.

2

Grammatik

Definition:

- Eine Grammatik G für eine Sprache L ist definiert durch
- ein ***Viertupel (N, T, P, S)***
- N: Menge der ***Nichtterminalsymbole***
- T: Menge der ***Terminalsymbole***
- P: Menge von ***Produktionsregeln***
- S: das ***Startsymbol***

Typ-0-Grammatik

- Gestalt der Produktionen ist nicht eingeschränkt

Typ-1 oder kontextsensitive Grammatik

- Produktionen sind beschränkt oder kontextsensitiv

Typ-2 oder *kontextfreie* Grammatik

- die linke Seite einer Produktion ist immer ein Nichtterminal

Typ-3 oder reguläre Grammatik

- Produktionen sind terminierend, links- , rechtslinear

2

EBNF u. Syntaxdiagramme

- **EBNF**
  - **E**xtended **B**ackus-**N**aur-**F**orm
  - ***Meta-Sprache*** zur Beschreibung der Syntax formaler Sprachen
  - ***Metasymbole*** von EBNF sind
    - = „definiert als“
    - (...|...) genau eine Alternative aus der Klammer muß stehen
    - [...] Inhalt der Klammer kann stehen oder nicht
    - { ... } Inhalt der Klammer kann n-fach stehen, n >= 0
    - . Ende der Produktion
    - Terminalsymbole werden in " " eingeschlossen
- **Syntaxdiagramme**
  - beschreiben Produktionen ***grafisch***
  - Nichterminalsymbole sind Rechtecke
  - Terminalsymbole sind Langrunde

3

Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
- **Was benötigt man dazu**
  - Daten / ***Datentypen*** und ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereich)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift*** , ist frei von ***Seiteneffekten***

3

Formale & aktuelle Parameter

- **Parameter**
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - haben eine ***Typ***
  - dadurch werden Funktionen ***flexible einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden
  - diese werden dann im Rumpf ***verwendet***.

3

Deklaration, Anweisung, Ausdruck

- **Deklarationen:**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***.
- **Anweisungen:**
  - Sind in Blöcken enthalten.
  - Die Folge der Anweisungen eines Blocks wird bei Ausführung in der ***Reihenfolge der Aufschreibung*** abgearbeitet.
- **Ausdrücke**
  - werden ***ausgewertet***
  - liefern einen Wert (Ergebnis)
  - Viele Anweisungen erlauben, daß Ausdrücke verwendet werden können

3

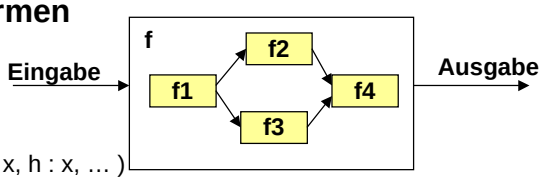
Datentypen

- **Typbegriff**
  - im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung
- **Man unterscheidet grob:**
  - ***einfache*** Datentypen
  - ***zusammengesetzte*** Datentypen
- **Einfachen Datentypen in Modula-3**
  - **Ganze Zahlen**
  - **Zeichen**
  - **Texte**
  - **Wahrheitswerte**
  - **Gleitkommazahlen**

3

Vernetzung von Funktionen

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, …] : x = (f : x, g : x, h : x, …)
  - **Bedingung**
    - if t then f else g : x =



f : x , falls t : x = true
g : x , falls t : x = false
? , sonst

3

Rekursion

- **Idee:**
  - allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens "***durch sich selbst***"
- **Rekursive Funktion**
  - darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen***
- **Indirekte Rekursion:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Anmerkung:**
  - Es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung***
  - Rekursionsabbruch

4

Variable und Wertzuweisung

- **Variable**
  - **logischer** Speicherplatz mit seinem Wert
  - besitzt einen **Namen**, unter dem man die Variable ansprechen kann
  - Datentyp legt fest, welche **Werte** eine Variable annehmen kann und welche **Operationen** darauf ausgeführt werden können
  - Variable kann als Behälter betrachtet werden
    - hat einen Wert und einen Typ
- **Wertzuweisung**
  - dient dazu, den Wert einer Variablen zu verändern
  - Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variable zugewiesen
    - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
    - (right-hand side - RHS, left-hand side - LHS).
  - LHS und RHS müssen Typkompatibel sein

4

Abstraktion durch Prozeduren

- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.
- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.

4

Parameterübergabearten

- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
  - ***Ausgabeparameter***
  - ***Ein- / Ausgabeparameter***
- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
  - realisieren Eingangsparameter
  - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
- **Der formale Parameter beim *Call by Reference* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
  - realisieren Ausgangs und Ein- / Ausgangsparameter
  - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.

4 Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - wird auch ***Sichtbarkeitsbereich*** oder ***Namensraum*** genannt
  - Bezeichner ist in seinem Sichtbarkeitbereich gültig
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

5 Arten von Kontrollanweisungen

- **Sequenz von Aktionen:**
  - Eine Aktion wird ***nach der anderen*** abgearbeitet.
- **Auswahlanweisungen kommen vor als:**
  - Einwegauswahl (one-way selection) IF-THEN
  - Zweiwegauswahl (two-way-selection) IF-THEN-ELSE
  - Mehrfachselektion (multiple selection) CASE
- **Wiederholungsanweisungen werden benötigt,**
  - um ***iterative Algorithmen*** zu formulieren.
  - Schleife mit vorheriger Prüfung WHILE und FOR
  - Schleige mit nach nachfolgender Prüfung REPEAT-UNTIL
  - Schleife ohne Prüfung LOOP (EXIT)

6

Einteilung von Typen

- **Einfache und zusammengesetzte Datentypen:**
  - ***Einfache Datentypen*** erlauben ***keinen*** Zugriff auf ihre innere Struktur. Ihre Werte können unmittelbar notiert werden.
  - **Zusammengesetzte Datentypen** sind aus anderen Datentypen aufgebaut. Auf ihre einzelnen Elemente kann zugegriffen werden.
- **Vorgegebene und benutzerdefinierte Datentypen:**
  - **Vorgegebene Datentypen** haben einen vordeklarierten Namen und können unmittelbar zur Deklaration von Variablen verwendet werden.
  - **Benutzerdefinierte** Datentypen haben einen selbst definierten Namen und müssen deklariert werden.
- **Statische und dynamische Datentypen**
  - **Statisch**: Größe der Typobjekte ist von vornherein ***bekannt***
  - **Dynamisch**: Größe ist während der Laufzeit ***veränderbar***

6

Benutzerdefinierte Typen

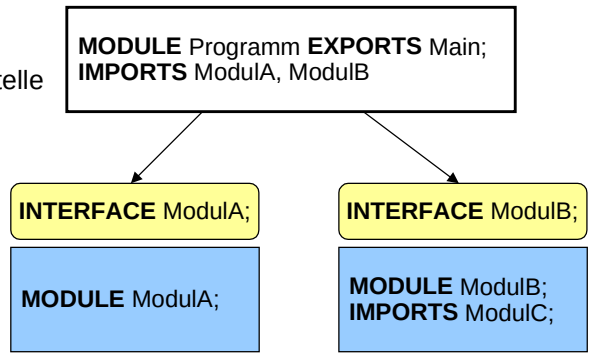
- **Benutzdefinierte einfache Typen,**
  - `TYPE Zeit = REAL;`
- **Unterbereichstyp (subrange type)**
  - `TYPE Index = [1..10];`
- **Aufzählungstyp (emumeration type)**
  - `TYPE Ampelfarbe = {rot, gelb, gruen};`
- **Array-Typ:**
  - Aneinanderreihung von ***gleichartigen Elementen***, wobei auf die Komponenten mit Hilfe eines ***Index*** zugegriffen wird.
- **Record-Typ**
  - Darstellung ***inhomogener***, aber ***zusammengehöriger*** Informationen.
- **Mengen-Typ**
  - Mengen sind ***ungeordnete*** Sammlungen von Elementen

6 Dynamische Datentypen/ Prozedurtyp

- **Eigenschaften von Objekten dynamischer Datentypen**
  - ***Lebensdauer*** ist nicht an Prozedur oder Moduls gebunden
  - werden ***explizit erzeugt*** und eventuell auch wieder beseitigt
  - sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
  - können in prinzipiell ***beliebiger*** Menge geschaffen werden
  - haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
  - sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur.***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.

7 Modulkonzept

- **Entstanden aus der Notwendigkeit,**
  - große Programmtexte in für den ***Übersetzer faßliche Einheiten*** zu zerlegen,
  - Modulkonzept ist zum zentralen ***Organisationskonzept*** für Entwürfe und Programmtexte geworden.
- **Konzept:**
  - Trennung von Schnittstelle und Implementierung
  - IMPORT-Beziehung zwischen Modulen



7

Information Hiding

- **Prinzip:**
  - Es werden nur die Informationen zur Verfügung gestellt, die ***absolut notwendig*** sind!
  - Alle anderen, insbesondere die ***wichtigen Informationen*** werden ***versteckt***!
  - Der Zugriff auf diese Informationen geschieht über "***Vermittler***".
- **Information Hiding wird realisiert durch**
  - Datenkaspelung in Objektmodulen
  - Abstrakte Datentypen

7

Objektmodul - Datenkapsel

- **Die zentrale Idee:**
  - ***Trenne*** die konkrete Realisierung (i.e. Implementation) einer Datenstruktur von ihren sichtbaren Eigenschaften.
- **Merkmale:**
  - Eine Datenstruktur wird in einem Modul ***eingekapselt***.
  - An der Schnittstelle des Moduls sind nur ***Operationen sichtbar***, die den allgemeinen Umgang mit der Datenstruktur beschreiben.
  - Die Datenstruktur selbst ist ***verborgen***.
- **Jedes Objektmodul**
  - beschreibt und realisiert nur eine ***einzige sog. abstrakte Datenstruktur***.

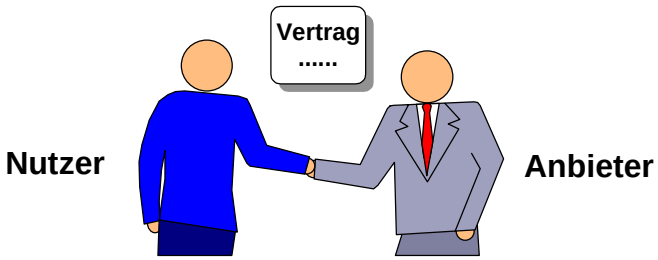
7

Konzept Abstrakter Datentyp

- **Kann als *Generalisierung* des Objektmoduls (Datenkapsel) betrachtet werden.**
  - Anstatt eines Objektes wird ein Typ (für diese Objekte) definiert.
- **Betrachten wir den ADT als (formale) Spezifikation eines Typs,**
  - dann entwerfen wir ihn durch Angabe von
  - ***Typnamen***
  - ***Signaturen*** (Operationen)
    - zum Erzeugen von Objekten, zum Verändern etc.
  - ***Axiome***
    - formulieren den semantischen Zusammenhang der Operationen.
  - ***Vorbedingungen***
    - geben an, in welchem Zustand welche Operationen gültig sind.

8

Idee des Vertragsmodells



- **Vertrag**
  - zwischen Nutzer und Anbieter einer Operation regelt, wer der beiden Partner welche ***Verpflichtungen*** einhalten muß (und welchen ***Nutzen*** er dadurch hat)
- **Operation**
  - arbeitet korrekt, wenn sie vertragsgemäß ***benutzt*** bzw. ***realisiert*** wird.

8

Zusicherungen

- **Zusicherungen**
  - sind eine Technik, um eine bestimmte Art von Verträgen zwischen Anbieter und Nutzer zu formulieren.
- **Zusicherungen werden formuliert als**
  - ***Vorbedingungen*** für Operationen
  - ***Nachbedingungen*** von Operationen
  - ***Invarianten*** von abstrakten Datentypen
- **Zusicherungen**
  - erhöhen die ***Benutzbarkeit***, indem sie diese formaler definieren
  - verbessern die ***Testbarkeit***
  - verbessern die ***Fehlersuche*** (debugging)
  - verlangen vom Entwickler ein ***abstraktes*** Denken

8

Realisierung von Zusicherungen in M3

- **Mit Hilfe des Pragmas ASSERT**

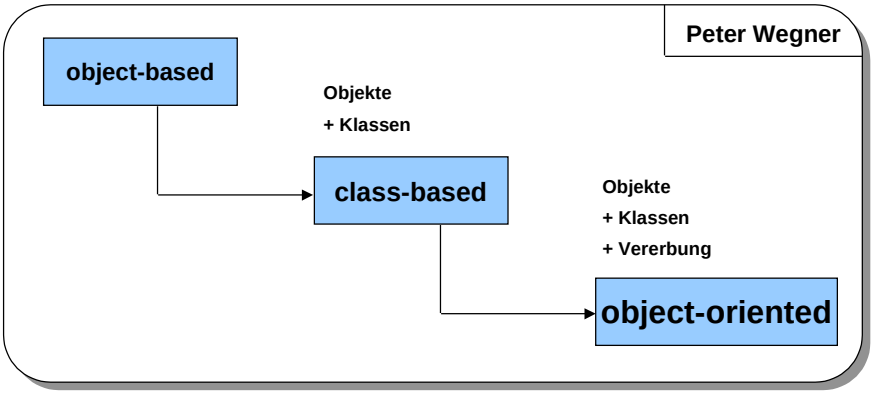
```
PROCEDURE EntnehmeText (VAR o: Ordner): TEXT =
VAR t : TEXT;
BEGIN
 <* ASSERT NOT IstLeer(o) *>
 . . .
 <* ASSERT NOT IstVoll(o)  *>
 <* ASSERT Invariante(o)   *>
END EntnehmeText;
```

- **Mit Hilfe von Ausnahmen**

```
PROCEDURE EntnehmeText (VAR o: Ordner) : TEXT
                        RAISES {As. Violated} =
VAR t : TEXT;
BEGIN
  As.Require (NOT IstLeer(o), "OrdnerADT.EntnehmeText");
  . . .
  As.Ensure (NOT IstVoll(o), "OrdnerADT.EntnehmeText");
  As.Ensure (Invariante(o), "OrdnerADT.EntnehmeText");
END EntnehmeText;
```

9

Merkmale objektorientierter Sprachen



Ada
Modula-2

CLU
(Ada)
(Modula-2)

Simula
Smalltalk-80
Eiffel
C++
Java
Modula-3

9

Objekt, Nachricht, Klasse, Vererbung

- **Ein Objekt ist eine Datenkapsel, die aus zwei Teilen besteht**
  - Der Wert der Daten repräsentiert den **Zustand** des Objekts
  - Daten können nur mithilfe von **Operationen** verändert werden
- **Objekte kommunizieren miteinander,**
  - dadurch daß sie **Nachrichten** versenden und Nachrichten empfangen.
- **Gleichartige Objekte werden in einer Klasse beschrieben**
  - **Speicherstruktur**
  - **Operationen** (Methoden, Routinen)
  - konkrete Klassen, abstrakte Klassen
- **Gemeinsame Eigenschaften verschiedener Klassen**
  - werden in einer **eigenen Klasse** zusammengefaßt und definiert, und anschließend an diese vererbt.
  - Einfachvererbung - Mehrfachvererbung

9

Das dynamische Binden

```
g : Geo_Object;
r : Rectangle;
c : Circle;

r.Create; c.Create

g := r;
g.contains (p);

g := c;
g.contains (p);
```

```
class Geo_Object

feature
  contains (p : Point) : Boolean is
    deferred
    end;
end -- class Geo_Object
```

```
class Rectangle

feature
   contains (p : Point) :Boolean is
     ...
end -- class Rectangle
```

```
class Circle

feature
  contains (p : Point) :Boolean is  ...
end -- class Circle
```

Dynamisches Binden heißt:

die **richtige** Implementierung zur **Laufzeit** finden

9

Klassen in Modula-3 - Objekttypen

```
INTERFACE Geo_Object;
IMPORT Point;

TYPE T =  ROOT OBJECT
  METHODS
    moveTo (p : Point.T);
    contains (p : Point.T) : BOOLEAN;
  END;
END Geo_Object.
```

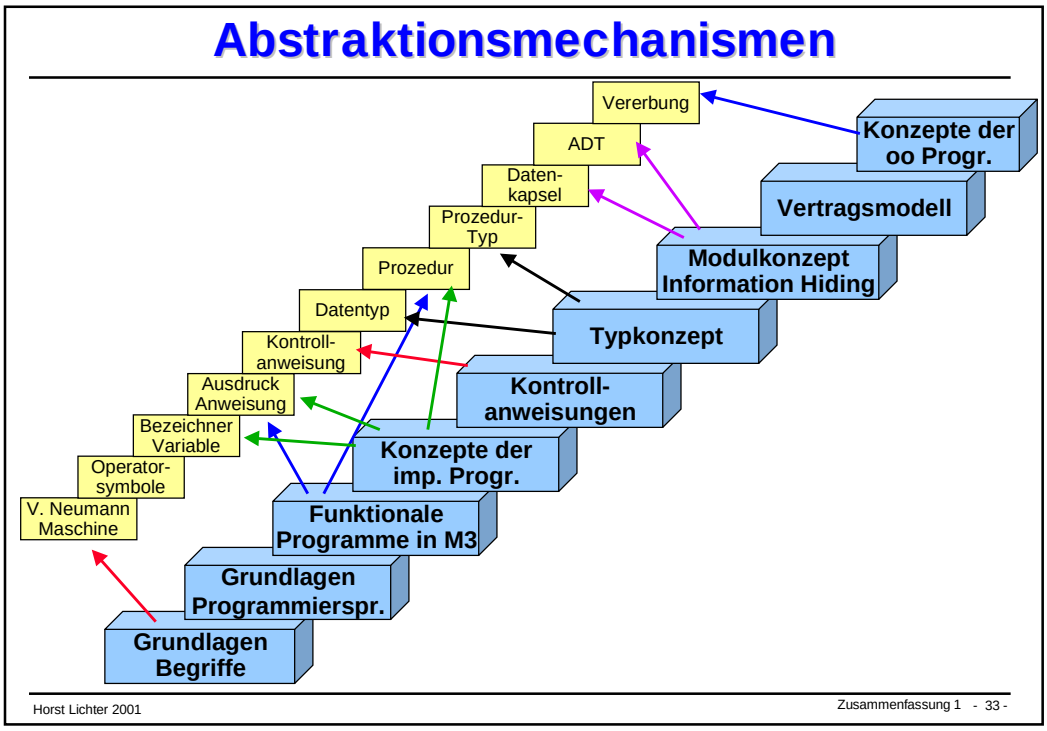
Abstrakte Klasse

```
INTERFACE Rectangle;
IMPORT Geo_Object, Point;

TYPE T <: Public;
     Public = Geo_Object.T OBJECT
              METHODS
                area() : INTEGER;
                height () : INTEGER;
                setOrigin (p : Point.T);
                getOrigin(): Point.T;
                setCorner(p : Point.T);
                getCorner(): Point.T;
                asText () : TEXT;
              END;
END Rectangle.
```

```
INTERFACE DisplayableRectangle;
IMPORT Rectangle;

TYPE T <: Public;
     Public = Rectangle.T OBJECT
              METHODS
                setColor(c : TEXT);
                getColor() : TEXT;
                draw(): TEXT;
              END;
END DisplayableRectangle.
```


Abstraktionsmechanismen
Vererbung
ADT
Daten-kapsel
Prozedur-Typ
Prozedur
Datentyp
Kontroll-anweisung
Ausdruck Anweisung
Bezeichner Variable
Operator-symbole
V. Neumann Maschine
Konzepte der oo Progr.
Vertragsmodell
Modulkonzept Information Hiding
Typkonzept
Kontroll-anweisungen
Konzepte der imp. Progr.
Funktionale Programme in M3
Grundlagen Programmierspr.
Grundlagen Begriffe
Horst Lichter 2001
Zusammenfassung 1 - 33 -

Zusammenfassung
Teile V - VI

LISP-Programm

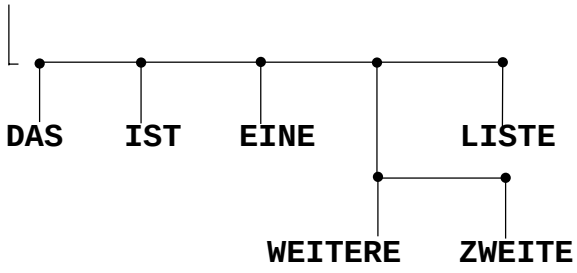
- **LISP – Programm:** **Folge von Ausdrücken, die nacheinander ausgewertet werden**
- **Ausdrücke**
  - Atome oder Liste
- **Form**
  - auswertbarer Ausdruck
- **Atome**
  - Zahlenliterale
  - Symbole (Literale, Konstante, Variablen)
  - Zeichenketten(literale) „a b c d“ „~ A“
- **Liste**

  (listelem listelem ... listelem)

  NIL

Liste für Daten und Funktionen

- **(DAS IST EINE (WEITERE ZWEITE) LISTE)**



- `(DEFUN VERTAUSCHE (PAAR)`
  `(LIST(CADR PAAR)(CAR PAAR))))`

Funktionen

- **Definition**
  - `(DEFUN FName [Parlist] [Rumpf] )`
- **QUOTE ist die Identitätsfunktion**
  - QUOTE wertet seine Argumente nicht aus!
  - `(QUOTE A)` kann kürzer dargestellt werden als `'A`
- **Die Wertzuweisung an ein symbolisches Atom geschieht durch die Systemfunktion** SETQ
  - `(SETQ L '( A B ))`
  - Ergebnis ist ein Seiteneffekt
- **Systemfunktion EVAL**
  - (eval (- 4 3))
  - Die Auswertung eines Ausdrucks kann damit explizit angestoßen werden.

Grundlegende Funktionen

- **CONS**
  - Aufbau einer Liste
- **CAR, CDR**
  - erstes Element einer Liste, Restliste
- **APPEND**
  - Die Liste, die durch Aneinanderfügen der Elemente der Listen gewonnen wird.
- **LIST**
  - Liste der Werte der Argumente in der gegebenen Reihenfolge
- **Prädikate**
  - ATOM, CONSP, NULL, LISTP. NUMBERP, EQ EQUAL

Funktionen

- **Klassifikation**
  - **Systemfunktionen**
    - spezielle Funktionen
    - normale Funktionen
  - **benutzerdefinierte Funktionen**
    - benannte Funktionen (DEFUN)
    - anonyme Funktionen (LAMBDA-Ausdruck)
- `(LAMBDA [Parlist] [Rumpf])` **analog zu DEFUN**
  - Definiert namenlose Funktion
  - Lambda-Ausdruck hat keinen Wert
  - `(Lambda-Ausdruck a1 a2 . . .an)`

Bindung

Bindung und Umgebung

- **Bindung**
  - Zuordnung von Symbolen zu Werten
- **Umgebung**
  - Gesamtheit der zu einem Zeitpunkt zugänglichen Bindungen
  - Aktuelle Umgebung
- **Definitionsumgebung einer Funktion**
- **Aufrufumgebung einer Funktion**
- **Gebundene Variable**
  - Variable, die in der Parameterliste einer Funktion auftaucht.
  - Lokale Variable
- **Freie Variable**
  - Variable, die im Rumpf einer Funktion steht, aber keinen Parameter bezeichnet.

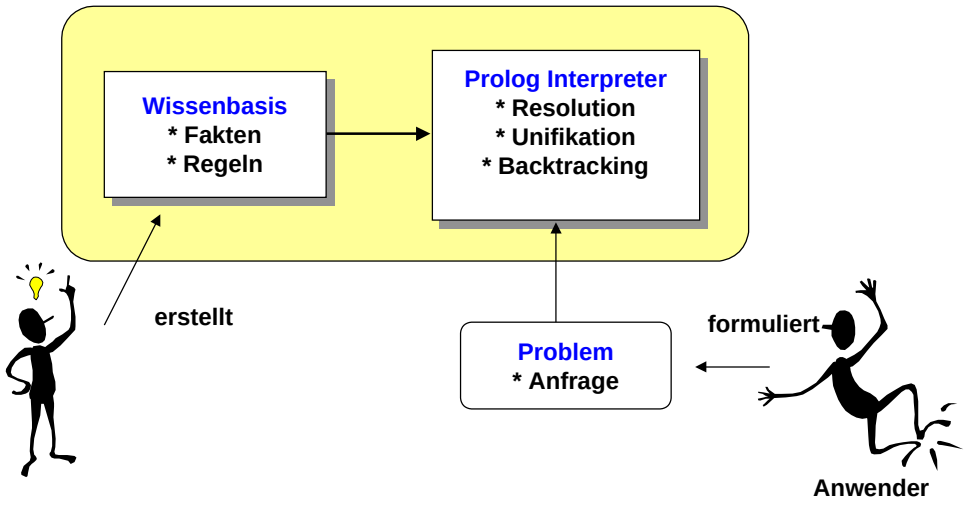
Funktionsobjekte

- **Die Funktion symbol-function**
  - erwartet einen Funktionsnamen als Argument und liefert ein Funktionsobjekt
- **Die Funktion function**
  - erwartet einen Lambda-Ausdruck und liefert ebenfalls ein Funktionsobjekt
  - Das Ergebnis ist eine sog. Closure
- **Closures sind (anonyme) Funktionen,**
  - die Variablen ihrer Umgebung referenzieren.
- **Ist das Argument von symbol-function eine selbstdefinierte Funktion,**
  - dann liefert diese auch eine Closure.
- **Als Kurzschreibweise**
  - für symbol-function und function kann #' benutzt werden.

Listendurchwanderung

- **Bei der Durchwanderung (mapping)**
  - geht man die Liste durch und wendet eine Funktion auf die Elemente der Liste an.
- **Mapping-Funktionen**
  - verlangen ein funktionales Argument; sind also Funktionen höherer Ordnung
  - Die Funktion wird entweder auf das erste Element der aktuellen Liste (car) oder auf die gesamte aktuelle Liste angewendet.
- **Das Gesamtergebnis der Durchwanderung der Liste**
  - kann gesammelt und als Liste zurückgegeben werden.
  - Sind nur Seiteneffekte von Interesse, kann die Liste unverändert zurückgegeben werden.
- **MAPC, MAPCAR**

Prolog Programmiermodell



Definitionen nach Sterling/Shapiro: Programm, Regel, Prädikat

- Ein **Logikprogramm** ist eine endliche Regelmenge.
- Statt Regel sagt man auch **Horn-Klausel** oder bei Eindeutigkeit auch einfach nur **Klausel.**
- Eine **Regel** hat die Form.

  $A <- B_1, B_2 , ... B_n.$ mit $n >= 0$

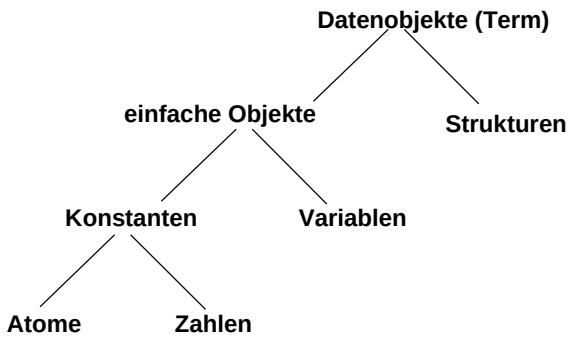
  A ist der **Regelkopf** und die B_i's sind der **Regelrumpf.**
  Eine Regel mit $n=0$ wird **Fakt** genannt. A und B_i's werden auch Ziele genannt.
- Ein **Ziel** hat die Form eines Prädikats (Prädikatenlogik 1. Stufe)

  $pred (t_1, t_2, ..., t_m)$ mit $m >=0$

  wobei pred Prädikatnamen, m Stelligkeit und die t_j Argumente.
  Prädikate beschreiben Objekte und Beziehungen zwischen diesen Objekten.
- Argumente für Prädikate sind **Terme**; induktiv definiert:
  - eine Konstante ist ein Term,
  - eine Variable ist ein Term,
  - sind $t_1, t_2, ..., t_k$ Terme und ist f ein Funktionssymbol, so ist auch $f (t_1, t_2, ..., t_k)$ ein Term.

Objekte in Prolog

- **Objekte**
  - repräsentieren Dinge der Vorstellungswelt oder sind Abstraktionen von realen Gegenständen.
- **Prolog kennt folgende Datenobjekte**
  - Prolog erkennt den Typ eines Objekts an seiner syntaktischen Struktur



Unifikation

- **Problem**
  - Wird eine Frage gestellt, so muss das Prolog-System entscheiden, ob der Term der Frage zu einem Term der Wissensbasis gleich ist oder nicht.
  - Nach Einführung von Termen mit Funktionssymbolen ist ein Mechanismus zum Auffinden „passender“ Regelköpfe zu definieren.
- **Prolog benutzt die *Unifikation*, um zu prüfen, ob zwei Terme miteinander übereinstimmen**
  - Zwei Terme sind ***unifizierbar***, falls es eine Substitution gibt, die die Terme gleich macht
  - Diese Substitution wird ***Unifikator*** genannt.
- **Beispiele**
  - t1 = `datum(D,M,1983)`, t2 = `datum(D1,may,Y1)`

    Die Substitution S = {D = D1,M = may, Y1= 1983} unifiziert t1 und t2

Resolutionsprinzip

- **Prolog versucht eine Anfrage auf der Menge der vorhandenen Fakten und Klauseln zu beweisen.**
  - Ein Faktum ist eine beweisbare Aussage
  - Es gilt
    - wenn jedes der Literale L1 , L2, L3, …, Ln beweisbar ist
    - und es eine Regel L :- L1 , L2, L3, …, Ln gibt
    - dann ist auch das Literal L beweisbar.
  - Beispiel:

```
istVaterVon(V,K) :- verheiratet(V,F) ,
                    istMutterVon(F,K).
```

  - Modus Ponens (Abtrennregel)
- **Resolutionsprinzip**
  - ist die Umkehrung des Modus Ponens
  - liefert ein Verfahren, um zu zeigen, ob eine Aussage beweisbar ist oder nicht.

Beweisstrategie

- **Resolution ist keine eindeutige Vorschrift**
  - Welches Literal einer Anfrage soll als nächstes bewiesen werden?
  - Durch welche Klausel soll ein Literal einer Anfrage abgeleitet werden?
- **Strategie**
  - A) Die Literale einer Anfrage werden von links nach rechts abgeleitet.
  - B) Die Klauseln werden von "oben" nach "unten" nach einer passenden durchsucht.
  - C) Falls das Prolog-System beim Beweisen in eine Sackgasse läuft, wird der letzte Ableitungsschritt rückgängig gemacht und die nächste passende Klausel zu einem neuen Ableitungsschritt verwendet (backtracking).

    Bem backtracking müssen Bindungen von Variablen, die beim Beweisschritt stattgefunden haben, rückgängig gemacht werden, damit neue Variablenbindung im alternativen Beweisschritt versucht werden kann.

Listen in Prolog

- **Eine Liste ist eine Folge von Objekten.**
  - Die Anzahl der Objekte einer Liste ist nicht festgelegt.
- **Eine Liste in Prolog ist entweder**
  - die leere Liste (dargestellt durch das Atom [])
  - die Struktur mit dem Funktor **.** und zwei Komponenten
    - die erste Komponente wird head genannt
    - die zweite Komponente ist wiederum eine Liste (tail)
- **Beispiele für Listen:**
  - []
  - . (10, [])
  - . (x, . (y, []))
  - . (a, . (b, . (c, [])))

Operatoren & Arithmetik

- **Prolog erlaubt die Verwendung von Operatoren.**
- **Operatoren werden in Infix-Notation geschrieben.**
  - z.B. 1+3 statt +(1,3)
- **Grundlegende arithmetische Operatoren sind**

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
mod	Modulo

- **Vergleichsoperatoren**

X > Y	X ist größer als Y
X >= Y	X ist größer als oder gleich Y
X < Y	X ist kleiner als Y
X =< Y	X ist kleiner als oder gleich Y
X =:= Y	X ist gleich Y
X =\= Y	X ist ungleich Y

```
INTERFACE Patient;

IMPORT Behandlung;

TYPE T <: REFANY;

PROCEDURE GetName(p: T): TEXT;

PROCEDURE SetName(p: T; n: TEXT);

PROCEDURE GetKrankenkasse(p: T): TEXT;

PROCEDURE SetKrankenkasse(p: T; kk: TEXT);

PROCEDURE AddBehandlung(p: T; behandl: Behandlung.T);

PROCEDURE RemoveBehandlung(p: T);

END Patient.
```

```
MODULE Patient;

IMPORT Behandlung;

TYPE Laenge = [1..10];
     Behandlungsfeld = ARRAY Laenge OF Behandlung.T;

REVEAL T = BRANDED REF RECORD
              name: TEXT;
              krankenkasse: TEXT;
              behandlungen: Behandlungsfeld;
              index: Laenge := 1;
           END;


PROCEDURE GetName(p: T): TEXT =
BEGIN
  RETURN (p^.name);
END GetName;

PROCEDURE SetName(p: T; n: TEXT) =
BEGIN
  p^.name := n;
END SetName;

PROCEDURE GetKrankenkasse(p: T): TEXT =
BEGIN
  RETURN (p^.krankenkasse);
END GetKrankenkasse;
```

```
PROCEDURE SetKrankenkasse(p:T; kk: TEXT) =
BEGIN
  p^.krankenkasse := kk;
END SetKrankenkasse;


PROCEDURE AddBehandlung(p: T; behandlung:
 Behandlung.T) =
BEGIN
  IF (p^.index <= 10) THEN
    p^.behandlungen[p^.index] := behandlung;
    p^.index := p^.index + 1;
  END;
END AddBehandlung;


PROCEDURE RemoveBehandlung(p: T) =
BEGIN
  IF (p^.index > 1) THEN
    p^.index := p^.index - 1;
  END;
END RemoveBehandlung;


BEGIN
END Patient.
```

Erinnerung: Felder

- **Definition**
- **Felder allgemein**
- **Feldoperationen**
- **Initialisierung**

Einleitung

- Ein Feld ist eine geordnete Sammlung oder auch Aneinanderreihung von Elementen des selben Typs.
- Die Anzahl der Elemente in einem Feld ist fest.
- Der Name einer Array-Variablen bezeichnet das gesamte Array.
- Ein einzelnes Array-Element wird durch einen Index bzw. mehrere Indizes identifiziert.
- Zur Identifizierung kann jeder Ordinaltyp verwendet werden.

Felder - allgemein

- Der Typkonstruktor, der in Deklarationen verwendet wird, ist in Modula-3 (vereinfacht):

```
<ArrayType> = ARRAY Index OF Komponenten
```

- Als Selektor für ein einzelnes Element wird die Indexangabe verwendet:

```
val := a1[3]
val := a2[4,5]
```

- Die Indexangabe wird auch für die selektive Zuweisung verwendet:

```
a1[4] := 42
```


Initialisierung von Feldern

- Mit Hilfe eines sog. **Feldaggregats** können konstante ARRAY-Objekte erzeugt und Feldobjekte initialisiert werden
  - `VAR x := Array_Type {e1, ..., en}`

    `e1` bis `en` sind Ausdrücke; ihr Wert wird den Feldelementen initial zugewiesen.

Operationen mit Feldern

Zuweisung

- Arrays können einander zugewiesen werden, wenn sie
  - den gleichen Basistyp und
  - die gleich Gestalt, d.h. gleiche Anzahl von Elementen in jeder Dimension haben
- zwei solche Arrays heißen zuweisungskompatibel

Vergleich

- Zwei zuweisungskompatible Arrays können auf Gleichheit und Ungleichheit geprüft werden.

Erinnerung: Mengen

- **Definition**
- **Mengenoperationen**
- **Mengenrelationen**

Einführung

- Modula-3 bietet einen eigenen vordefinierten Mengentyp.
- Mengen sind **ungeordnete** Sammlungen von Elementen.
- Der Elementtyp (Universum) muß ein **Ordinaltyp** sein.
- Elemente einer Menge können **nicht indiziert** werden.
- Ein Element einer Menge kann man nur „durch sich selbst“ bezeichnen.
- Wertebereich eines Mengentyps ist die **Potenzmenge**.
- Initialisierung möglich mit Mengenaggregat

Mengenoperationen - 1

- **Vereinigung (+)**
  - $S + T = \{x \mid (x \in S) \text{ oder } (x \in T)\}$
  - S+T ist die Menge aller Elemente, die in S oder in T oder in beiden Mengen enthalten sind.

- **Mengendifferenz (-)**
  - $S - T = \{x \mid (x \in S) \text{ und } (x \notin T)\}$
  - S-T ist die Menge aller Elemente, die in S sind aber nicht in T.

Mengenoperationen - 2

- **Durchschnitt (*)**
  - $S * T = \{x \mid (x \in S) \text{ und } (x \in T)\}$
  - S*T ist die Menge aller Elemente, die in S als auch in T enthalten sind.

- **Symmetrische Differenz (/)**
  - $S / T = \{x \mid ((x \in S) \text{ und } (x \notin T)) \text{ oder } ((x \notin S) \text{ und } (x \in T))\}$
  - S/T ist die Menge aller Elemente, die entweder in S oder in T aber nicht in beiden enthalten sind.

Mengenrelationen - 1

- **Gleichheit (=)**
  - S = T ist wahr gdw. S und T die gleichen Elemente enthalten
- **Ungleichheit (#)**
  - S # T gdw. NOT(S=T)
- **Teilmenge (<=)**
  - S <= T gdw. alle Elemente von S auch in T enthalten sind
- **echte Teilmenge (<)**
  - S < T gdw. (S <= T) AND (S # T)

Mengenrelationen - 2

- **Obermenge (>=)**
  - S >= T gdw. T <= S
- **echte Obermenge (>)**
  - S > T gdw. T < S
- **Enthalten (IN)**
  - e IN S gdw. $e \in S$
  - ergibt wahr, wenn das Element e in der Menge S enthalten ist.
  - e muß mit dem Basistyp kompatibel sein.
  - Beachte: Die IN-Relation unterscheidet sich von den anderen Relationen, da sie nicht zwei Operanden des gleichen Typs miteinander verbindet.

UML-Notation für objektorientierte Programme

Die Unified Modeling Language (UML) erlaubt es, die Struktur objektorientierter Programme zu visualisieren.

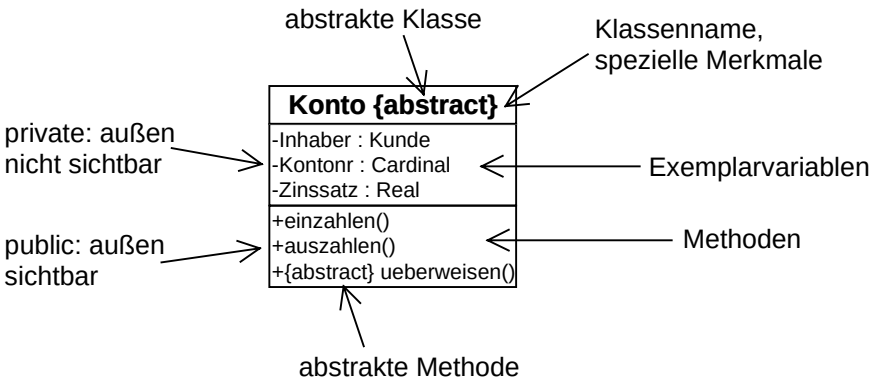
Merkmale der UML:

- Visuelle Modellierung von Systemen durch verschiedene Diagrammarten
- Unterschiedliche Sichten können modelliert werden:
  - statische Sicht: Ein System besteht aus Klassen. Die Klassen sowie die Beziehungen zwischen den Klassen müssen identifiziert und modelliert werden.
  - dynamische Sicht: Beschreibt das Verhalten der Objekte entlang der Zeitachse
- Diagrammarten:
  - Klassendiagramm (statisch)
  - Interaktionsdiagramm (dynamisch)

UML-Klassendiagramme

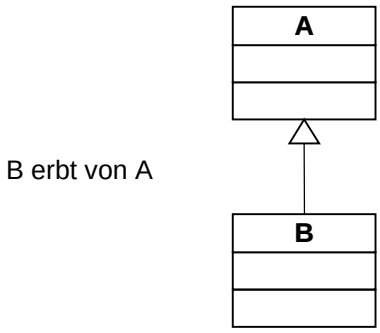
Bestehen aus Symbolen für Klassen und Beziehungen zwischen den Klassen.

Klassensymbol



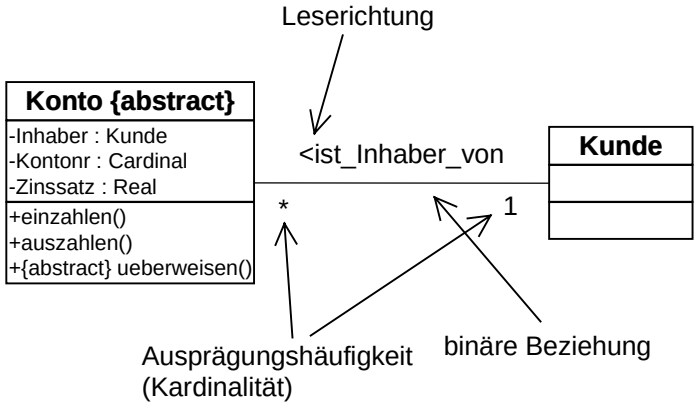
Beziehungen zwischen Klassen

Generalisierung

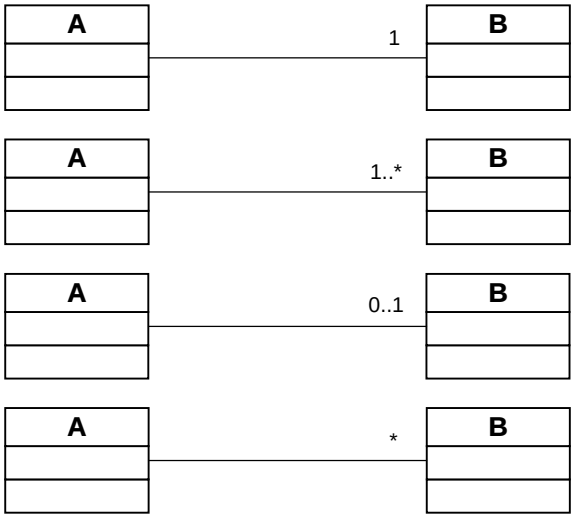


Allgemeine Beziehungen

Beschreiben gemeinsame Struktur einer Menge von statischen Beziehungen zwischen Objekten.



Kardinalitäten



Genau ein B-Objekt steht in Beziehung zum A-Objekt

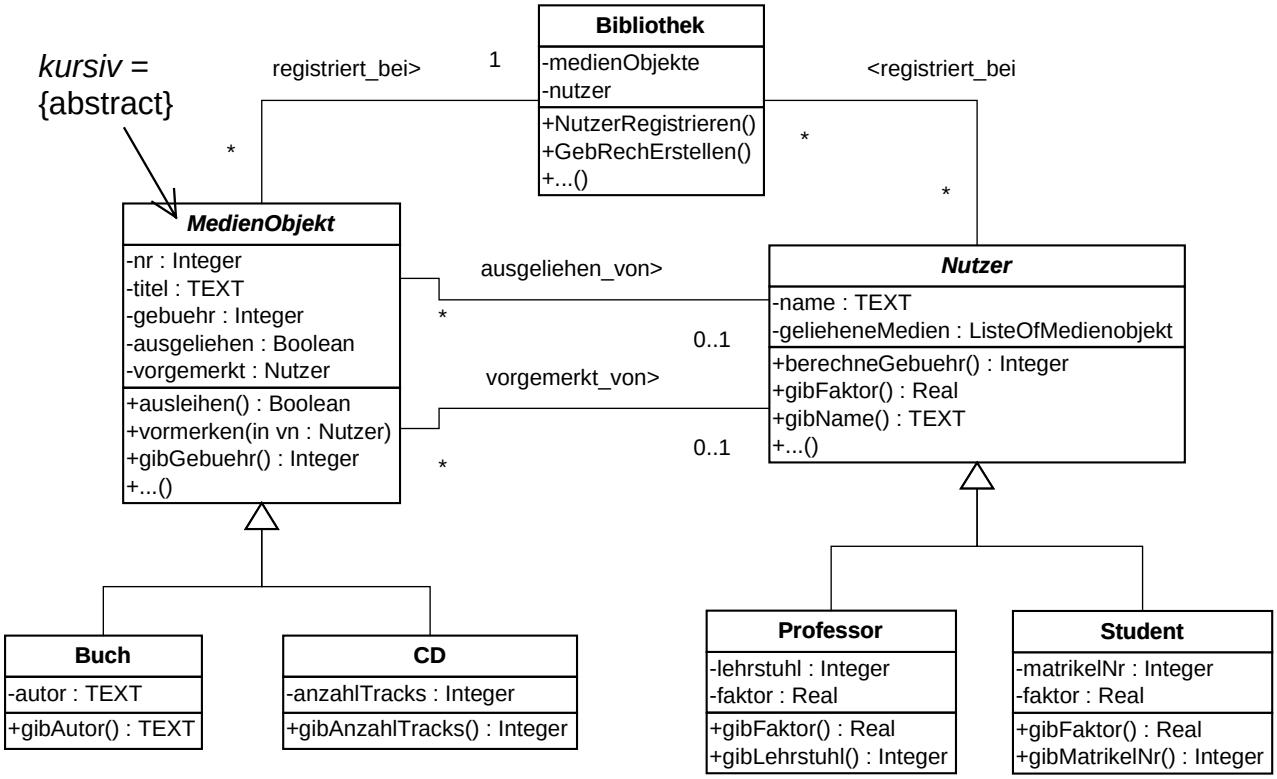
Jedes A-Objekt steht in Beziehung zu wenigstens einem B-O**bjekt**

Jedes A-Objekt steht in Beziehung zu **keinem** oder genau **einem** B-Objekt

Keine Einschränkung bzgl. Anzahl der B-Objekte

Beispiele:

Beispiel Klassendiagramm:



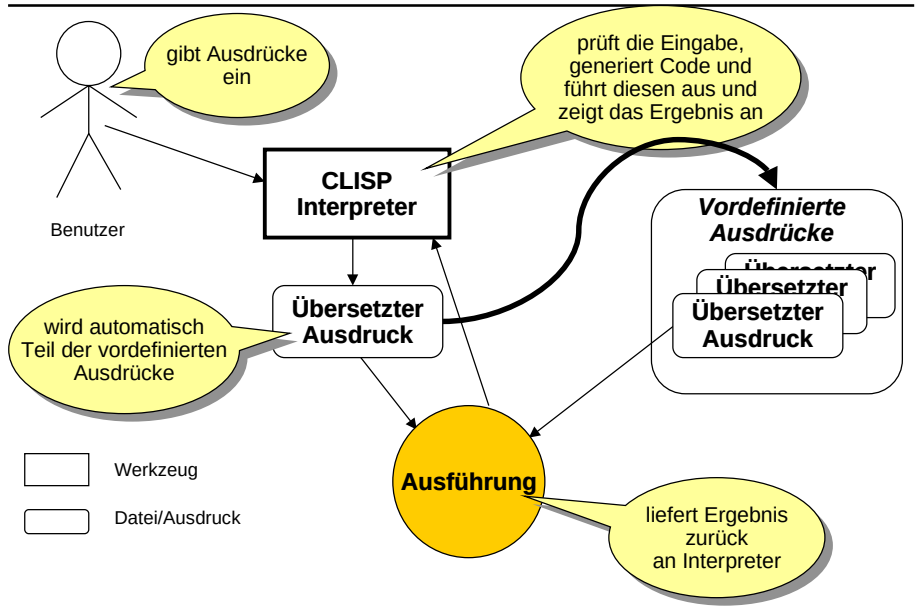
LISP Programmierung mit CLISP

- **CLISP**
- **Arbeiten mit CLISP**
- **Laden einer Datei**
- **Fehler - Was tun?**
- **Zusammenfassung**

CLISP

- **LISP**
  - Sprache für das ***Lis***t ***P***rocessing (LISP)
    - Basis sind Atome und Listen bzw. Listen von Listen
  - Verarbeitung von Symbolen und symbolischen Ausdrücken
  - erste Version von John McCarthy 1958 entwickelt
  - danach verschiedene LISP-Dialekte (MacLISP, Scheme, ...)
- **Common LISP ist heute Industriestandard**
- **CLISP**
  - implementiert Common LISP
  - ist freie Software und unter GNU-Lizenz frei verteilbar
  - besteht aus einem Interpreter und Compiler

Arbeiten mit CLISP - Interpreter

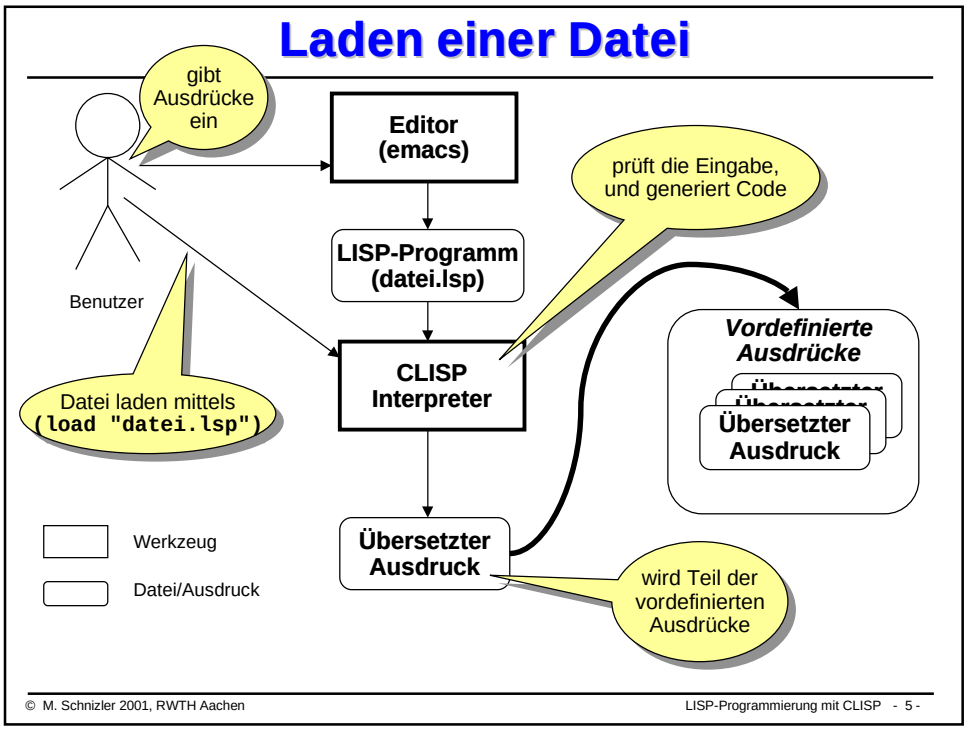


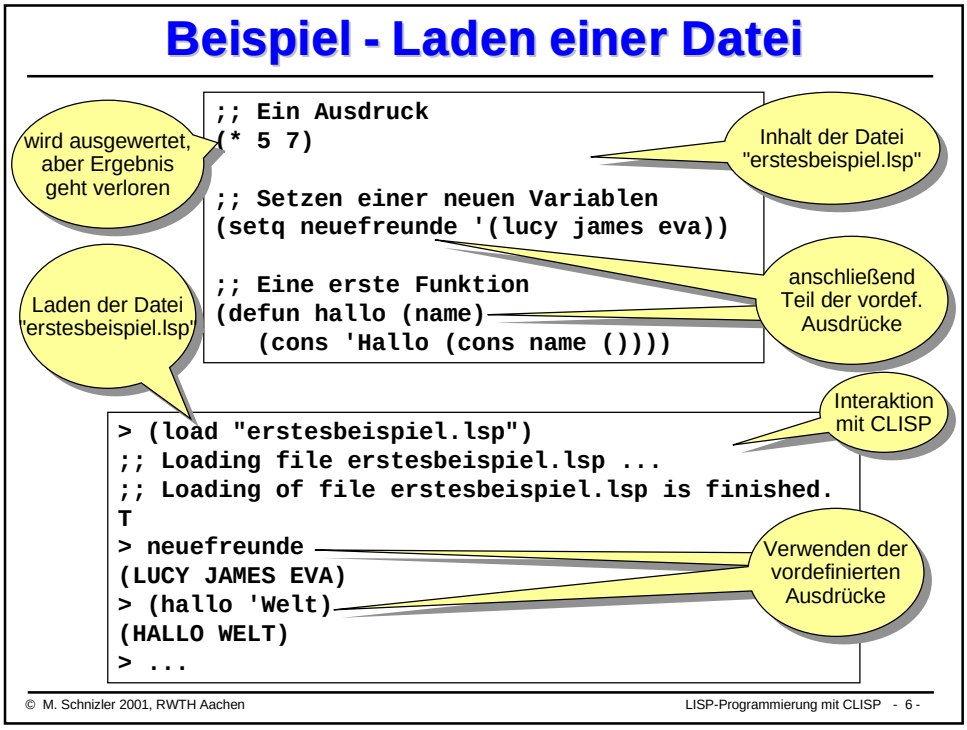
CLISP starten

- **Installieren der CLISP-Umgebung**
  - von Erstsemester-CD (z.B. E:\Software\lisp\win32\clispw32.zip)
  - bspw. ins Verzeichnis: C:\Programme\clisp
- **Starten der CLISP-Umgebung**
  - Wechseln ins Verzeichnis: C:\Programme\clisp
  - Ausführen: lisp.exe -M lispinit.mem
- **Danach Eingabe von Ausdrücken möglich:**

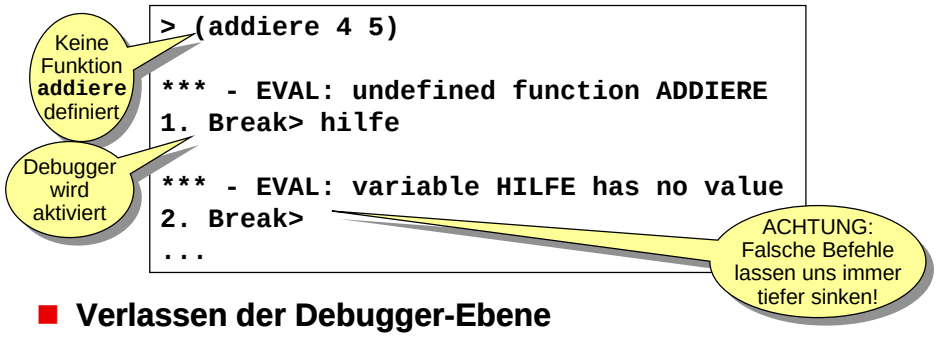
```
...
> (+ 2 3)
5
> (setq freunde '(dick jane sally))
(DICK JANE SALLY)
> freunde
(DICK JANE SALLY)
...
```

Problem: Eingaben sind nach Beenden des Interpreters verloren!


Laden einer Datei
gibt Ausdrücke ein
Editor (emacs)
prüft die Eingabe, und generiert Code
LISP-Programm (datei.lsp)
Benutzer
Vordefinierte Ausdrücke
CLISP Interpreter
Übersetzter Ausdruck
Datei laden mittels (load "datei.lsp")
Werkzeug
Datei/Ausdruck
Übersetzter Ausdruck
wird Teil der vordefinierten Ausdrücke
© M. Schnizler 2001, RWTH Aachen
LISP-Programmierung mit CLISP - 5 -


Beispiel - Laden einer Datei
;; Ein Ausdruck
(* 5 7)
;; Setzen einer neuen Variablen
(setq neuefreunde '(lucy james eva))
;; Eine erste Funktion
(defun hallo (name)
(cons 'Hallo (cons name ())))
wird ausgewertet, aber Ergebnis geht verloren
Inhalt der Datei "erstesbeispiel.lsp"
anschließend Teil der vordef. Ausdrücke
Laden der Datei 'erstesbeispiel.lsp'
Interaktion mit CLISP
> (load "erstesbeispiel.lsp")
;; Loading file erstesbeispiel.lsp ...
;; Loading of file erstesbeispiel.lsp is finished.
T
> neuefreunde
(LUCY JAMES EVA)
> (hallo 'Welt)
(HALLO WELT)
> ...
Verwenden der vordefinierten Ausdrücke
© M. Schnizler 2001, RWTH Aachen
LISP-Programmierung mit CLISP - 6 -

Fehler - Was tun?



- **Verlassen der Debugger-Ebene**
  - durch wiederholtes Verwenden von **unwind** oder **abort**
  - bei **unwind** und **abort** keine Klammern!!!
- **Debugger bietet Inspektionsbefehle**
  - **backtrace**: zeigt aktuellen Inhalt des Stacks an
  - **where,up,down**: erlauben Navigation im Auswertungsstack
  - **help** zeigt weitere Befehle

Zusammenfassung

- **CLISP**
  - frei verfügbar und unter GNU Lizenz verteilbar
  - implementiert den Industriestandard CommonLISP
  - ist auch auf der Erstsemester-CD
- **Arbeit mit CLISP**
  - Ausdrücke werden interpretiert
  - können direkt eingegeben
  - oder auch aus einer Datei geladen werden ⇒ Übungsabgabe!!!
- **Fehler**
  - führen automatisch in den Debugger
  - **unwind** und **abort** (ohne Klammern!!) führen wieder raus!
  - weitere Befehle erlauben Analyse des Fehlers

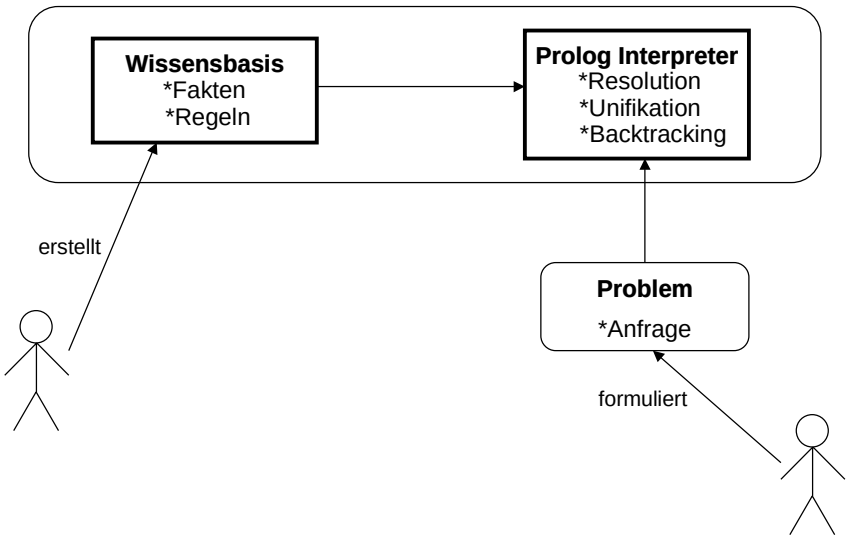
Prolog Programmierung mit SWI Prolog

- **SWI Prolog**
- **Arbeiten mit SWI Prolog**
- **Laden einer Datei**
- **Fehler**
- **Zusammenfassung**

SWI Prolog

- **Prolog**
  - steht für **Prog**ramming in **Log**ic
  - Logik wird als Programmiersprache benutzt, basiert auf der **Prädikatenlogik**
  - ist gut geeignet für Probleme, die Objekte und Relationen zwischen Objekten behandeln
  - Syntax der Sprache ist eine modifizierte **Hornklausel-Notation**
- **SWI Prolog**
  - ist freie Software und unter GNU-Lizenz frei verteilbar
  - besteht aus einem Interpeter

Prolog Programmiermodell



Arbeitsweise von Prolog, Fragen

- Zuerst wird die **Wissensbasis** erstellt und dem Prolog-System mitgeteilt.
- Dann werden die **Fragen** an die Wissensbasis gestellt, deren Richtigkeit bewiesen werden soll.
- Prolog erzeugt die **Antwort**, indem es einen logischen Beweis auf der Basis des vorhandenen Wissens durchführt.

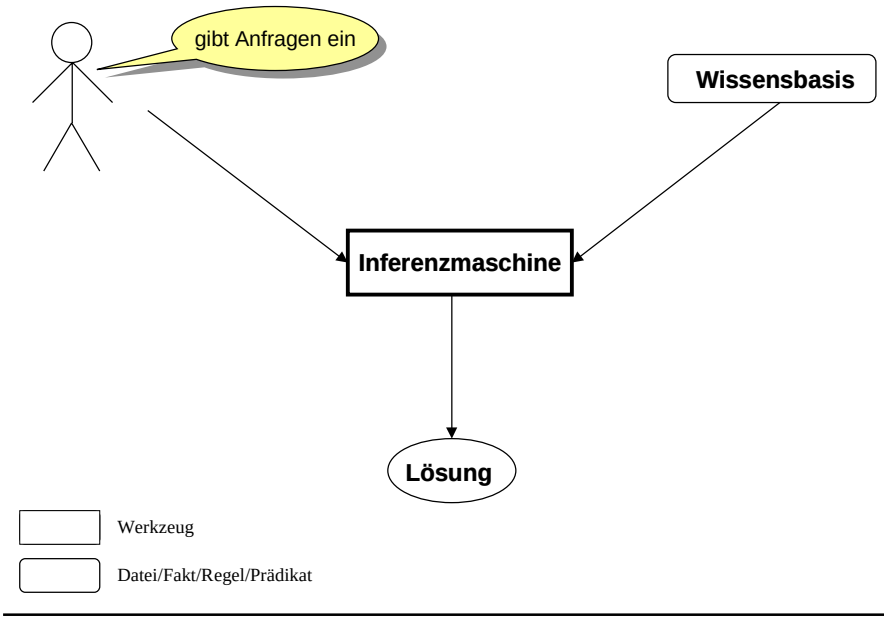
Fakten, Regeln, Programme (1)

- **Regeln** dienen dazu, aus bekanntem Wissen neues Wissen herzuleiten
- **Regeln** bestehen aus
  - einem Bedingungsteil (rechte Seite, Rumpf)
  - und einem Folgerungsteil (linke Seite, Kopf)
- Eine Regel **p:-q, r** kann gelesen werden
  - Deklarative Bedeutung im Sinne von WAS
    - IF rechte Seite der Regel THEN linke Seite der Regel
  - Prozedurale Bedeutung im Sinne von WIE
    - Um ein Ziel p zu lösen, müssen zuerst die Unterziele q und r gelöst werden

Fakten, Regeln, Programme (2)

- Prolog wendet die Regeln rückwärts an
  - d.h. Prolog startet mit der linken Seite
  - um zu zeigen, daß die linke Seite gilt, muß gezeigt werden, daß die rechte Seite gilt
- Prolog-**Programme** bestehen aus Fakten und Regeln
- Treten Variablen in Fragen auf, so antwortet Prolog
  - mit einer Instanziierung der Variablen in der Frage
  - weitere Instanziierungen können mit ; abgerufen werden

Arbeiten mit dem SWI-Prolog - Interpreter

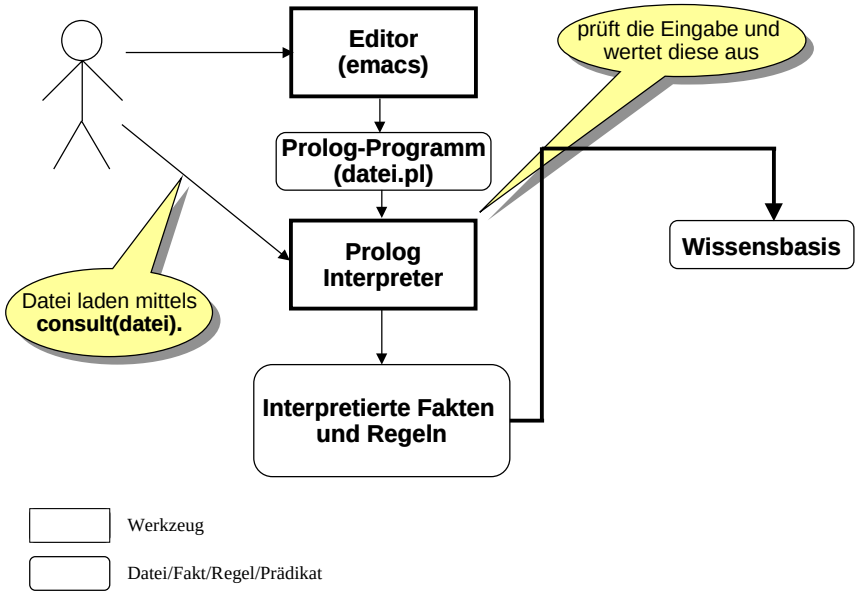


SWI-Prolog starten

- **Installieren von SWI-Prolog (unter Windows)**
  - z.B. von der Erstsemester-CD
    - selbstentpackendes Archiv
    - den Dialogen folgen
- **Starten der SWI-Prolog-Umgebung**
  - z.B. über den entsprechenden Menü-Eintrag im Programmverzeichnis
- **Danach Eingabe von Anfragen möglich:**

```
...
?- <(3,2).
No
?- ...
```

Laden einer Datei



Beispiel - Laden einer Datei

```
/* Ein paar Fakten. */
mutter(karin,susanne).
mutter(karin,monika).

/* Eine Regel. */
schwester(X,Y) :- mutter(Z,X), mutter(Z,Y).
```

Inhalt der Datei "erstesbeispiel.pl"

```
?-consult(erstesbeispiel).
%erstesbeispiel compiled 0.00sec, 840bytes
Yes
?-schwester(susanne, monika).
Yes
?-schwester(susanne, X).
X= susanne;
X= monika;
No
?- ...
```

Laden der Datei "erstesbeispiel.pl"

Verwenden der Prädikate aus "erstesbeispiel.pl"

Anmerkungen

- Jede Klausel (d.h. Fakt, Regel, Anfrage) wird mit einem **Punkt** abgeschlossen.
- **Kommentare** beginnen mit /* und enden mit */
- Laden einer Datei unter Windows ist auch direkt über den Windows-Explorer möglich („Doppelklick“).
  - Das **Verzeichnis**, auf welches der Interpreter dann beim Laden weiterer Dateien mit Hilfe von consult zugreift, ist das der geladenen Datei.
  - Sonst ist dies das voreingestellte Verzeichnis.

Fehler (1)

- Bei einem syntaktischen Fehler wird die **Ausführung abgebrochen** und eine entsprechende Fehlermeldung produziert.
- Es können jedoch nicht nur syntaktische Fehler auftreten sondern auch Fehler folgender Art:
  - eine (gemäß der Aufgabenstellung) fehlerhafte Antwort
  - unendlicher Beweis
- Bei solchen (nicht syntaktischen) Fehlern können **vordefinierte Prädikate** die Fehlersuche unterstützen:
  - **trace**
    - zur Ausgabe des Beweisverlaufes
    - beim Beweis des Literals **notrace**
      - wird als Nebeneffekt der trace-Modus ausgeschaltet
    - wird hier genauer erläutert

Fehler (2)

- **spy**
  - setzen von Beobachtungspunkten auf vom Benutzer definierte Prädikate (das Prädikat muß übergeben werden)
  - beim Beweis des Literals **nospy P**
    - werden als Nebeneffekt die Beobachtungpunkte von dem in P spezifizierten Literal entfernt
- **debugging**
  - Ausgabe der gerade gesetzten Beobachtungspunkte
  - beim Beweis des Literals **nodebug** werden als Nebeneffekt
    - alle gesetzen Beobachtungspunkte gelöscht
    - der debug-Modus ausgeschaltet

Fehleranalyse mit trace

- Beim Beweis des Literals **trace** wird als Nebeneffekt der trace-Modus eingeschaltet.
- Im **trace-Modus**
  - wird jeder einzelne Beweisschritt des Prologsystems ausgegeben
  - danach auf eine Reaktion des Benutzers gewartet
- Beim eventuellen Backtracking wird der trace-Modus nicht ausgeschaltet.
- Beim Beweis des Literals **notrace** wird als Nebeneffekt der trace-Modus ausgeschaltet

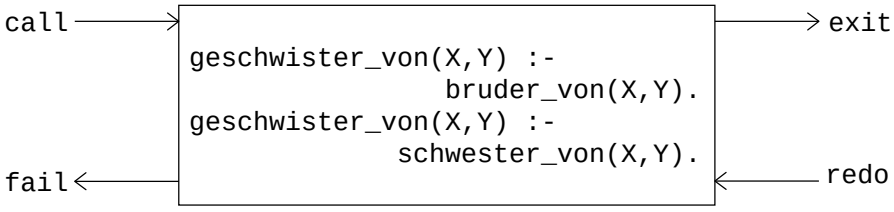
Ausgabe des Beweisverlaufs

- Bei der Ausgabe des Beweisverlaufs liegt das sogenannte **Box-Modell** zugrunde.
- Wenn ein Literal bewiesen werden soll, dann wird dieses **Literal als Box** dargestellt.
  - zwei **Eingänge**: call und redo
  - zwei **Ausgänge**: exit und fail
  - in der Mitte: die in der Datenbank vorhandenen Klauseln für das entsprechende Literal

Beispiel für das Box-Modell

- Alle Klauseln des Beipiels der Verwandtschaftsbeziehungen seien bekannt.
- Zu beweisendes Literal:

  `geschwister_von(hanna, wilhelm)`
- zugehöriges Box-Modell:



Bedeutung der Pfeile

- **call**
  - **erster Beweisversuch** des Literals
- **exit**
  - **erfolgreicher Beweis** des Literals
- **redo**
  - durch diesen Eingang betritt das Prolog-System die Box, wenn es **schon einen Beweis für das Literal gefunden** hat, aber ein **nachfolgendes Literal nicht bewiesen** werden konnte.
  - Ein **neuer Beweis** für das Literal muß gesucht werden.
- **fail**
  - **Verlassen** der Box, wenn
    - **keine** in der Box stehende **Klausel zum Beweis** des Literals **verwendet werden kann** oder
    - **bei allen verschiedenen Beweisen** des Literals **ein nachfolgendes Literal nicht bewiesen werden konnte**

Steuerung des trace-Modus

- **creep**
  - nächstes Betreten oder Verlassen einer Box soll ausgegeben werden
  - Benutzer wird erneut aufgefordert, den trace-Modus zu steuern
- **skip**
  - Es soll erst wieder eine Ausgabe erfolgen, wenn die gerade aktuelle Box betreten oder verlassen wird.
- **leap**
  - Es soll erst wieder eine Ausgabe erfolgen, wenn ein Beobachtungspunkt erreicht wird oder wenn die aktuelle Box betreten oder verlassen wird.
- **abort**
  - Der Beweis soll abgebrochen werden.

Zusammenfassung

- **Prolog**
  - ist ein Beispiel einer logischen, deklarativen Programmiersprache
  - beantwortet Anfragen unter Zuhilfenahme einer Wissensbasis
- **SWI-Prolog**
  - frei verfügbar und unter GNU-Lizenz verteilbar
  - ist auch auf der Erstsemester-CD
- **Arbeit mit SWI-Prolog**
  - Fakten und Regeln werden aus einer Datei geladen
  - Anfragen werden direkt eingegeben
- **Fehler**
  - Ausgabe des Beweisverlaufes mit Hilfe des Literals trace ermöglicht Analyse

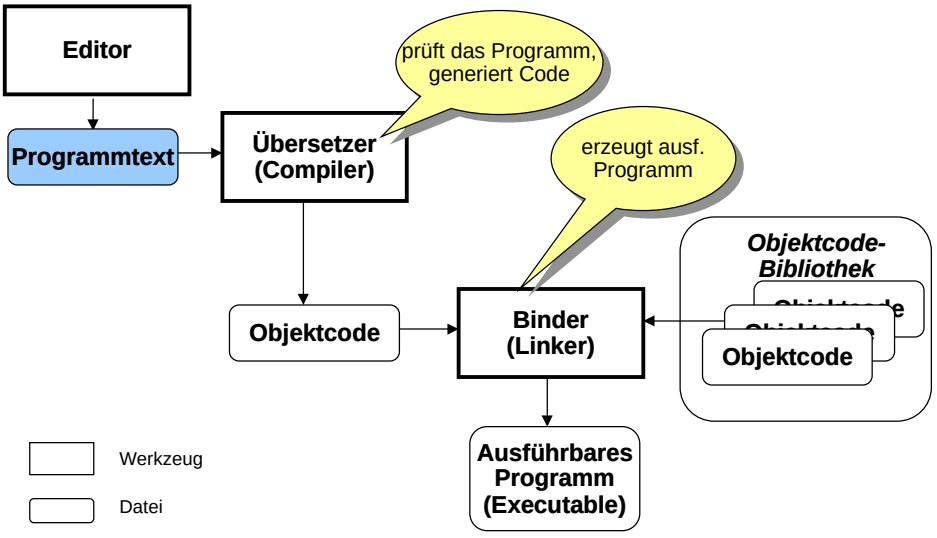
Infrastruktur für die Modula-3 Programmierung

- **Hilfsmittel**
- **Editor und PM 3**
- **Das erste Modula-3 Programm**
- **Ein interaktives Programm**
- **Fehler und Warnungen**
- **Zusammenfassung**

Hilfsmittel bei der Programmierung

- **Bei der Programmierung verwenden wir Hilfsmittel.**
  - Diese nennt man auch **Programmentwicklungswerkzeuge**.
  - Sie sind selbst wieder Programme.
- **Editor**
  - Erstellen des Programmtextes.
- **Übersetzer (Compiler)**
  - Dieser **prüft** den Programmtext auf Fehler und **übersetzt** den Programmtext in eine ausführbare Form.
- **Binder (Linker)**
  - Werden große Programme erstellt, bestehen diese nicht nur aus einer Programmdatei, sondern aus vielen.
  - Der Binder erzeugt daraus das **ausführbare** Programm.

Vom Programmtext zum ausf. Programm



Editor

- **Notwendig, um den Programmtext einzugeben**
- **Eigenschaften eines “guten” Programmeditors**
  - stellt Zeichen mit fester Breite dar
  - erzeugt reine Textdatei
  - Anzeige der Zeilennummer (z. B. Fehlersuche)
  - hebt syntaktische Elemente hervor
- **Geeignete Editoren**
  - Unix: xemacs, vi, emacs, xedit, ...
  - Windows NT: xemacs, vi, zur Not auch Wordpad, ...
  - SCHLECHT: Word, Framemaker, ...

PM 3

- **Polytechnique (Montreal) Modula-3**
  - aktuellste Modula-3 Freeware-Implementierung
  - Teil der Erstsemester-CD
  - basiert auf dem ursprünglichen DEC SRC Modula-3
- **Umfaßt den Compiler (Übersetzer)**
  - prüft die Syntax des Programmtexts
  - übersetzt in durch den Rechner ausführbaren Code
- **Linker (Binder) und vorgefertigte Bibliotheken**
  - für die Ein-/Ausgabe von Daten (z.B. das Modul IO)
  - um Zeichenketten zu manipulieren (z.B. das Modul Text)
- **Laufzeitumgebung**
  - stellt den Rahmen für ein ausführbares Programm
  - übernimmt die Speicherverwaltung

Erstes MODULA-3 Programm

- **Aufgabenstellung:**
  - Erstellen Sie ein Programm, das den folgenden Willkommensgruß auf dem Bildschirm anzeigt und dann beendet!

```
---------------------------------
Willkommen zum Studium in Aachen!
---------------------------------
```

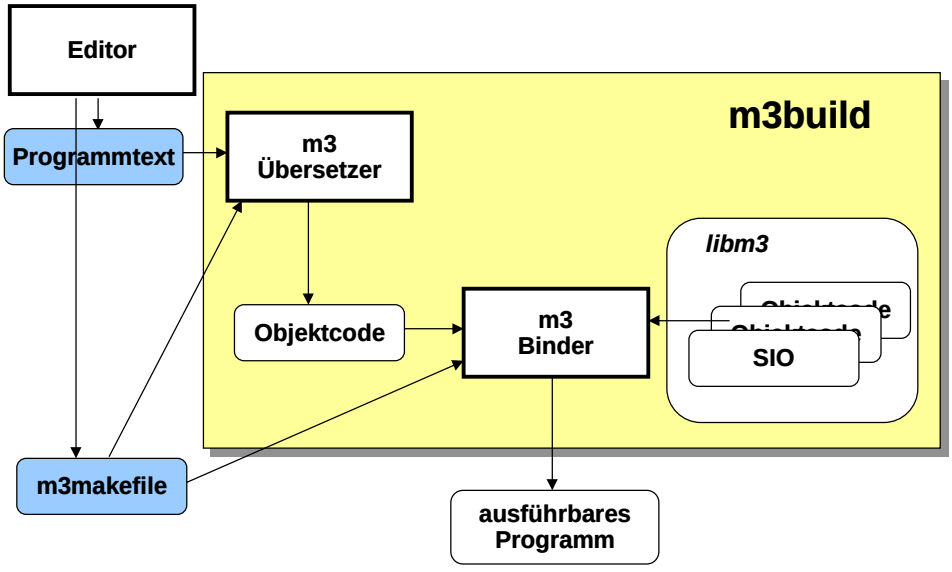
- **Lösungsidee:**
  - Da MODULA-3 Möglichkeiten anbietet, um Texte am Bildschirm auszugeben, können wir die Lösung zu dieser Aufgabenstellung direkt als Programm formulieren.
    - Modul SIO (Simple Input Output)
    - Stellt u.a. die Operationen
      - `SIO.PutText ()` und
      - `SIO.Nl()` zur Verfügung

Das erste M3-Programm

```
MODULE WillkommenInAachen EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Autor           : Horst Lichter, RWTH Aachen
   Erstellt        : 16.08.98
   Letzte Aenderung: 20.08.98
*)

IMPORT SIO;
BEGIN
  SIO.Nl();
  SIO.PutText("--------------------------------");
  SIO.Nl();
  SIO.PutText("Willkommen zum Studium in Aachen!");
  SIO.Nl();
  SIO.PutText("--------------------------------");
  SIO.Nl();
END WillkommenInAachen.
```


Entwicklungswerkzeuge für Modula-3



m3build

- **Kommando auf der Ebene des Betriebssystems,**
  - um ein Modula3-Programm zu übersetzen und, wenn möglich,
  - daraus eine ausführbare Datei zu erzeugen
- **m3build erwartet,**
  - die Datei **m3makefile** im aktuellen Verzeichnis
  - Fehlt diese, so wird eine Fehlermeldung ausgegeben.
- **m3build**
  - verarbeitet den Inhalt von m3makefile
  - übersetzt das dort angegebene Programm
    - gibt eventuell Fehlermeldungen aus
  - erzeugt aus dem übersetzten Programm eine ausführbare Datei

Die Datei m3makefile

- **Liste der Teile**
  - um ein ausführbares Programm zu erstellen
  - gibt an, welche "Teile" benötigt werden
  - und wo sich diese befinden
- **Beispiel:**

```
% Makefile fuer Modula-3
% Programm WillkommenInAachen

import("libm3")
import("libSIO")

implementation("WillkommenInAachen")
program("WillkommenInAachen")
```

Standardmodul für Modula-3

Modul SIO

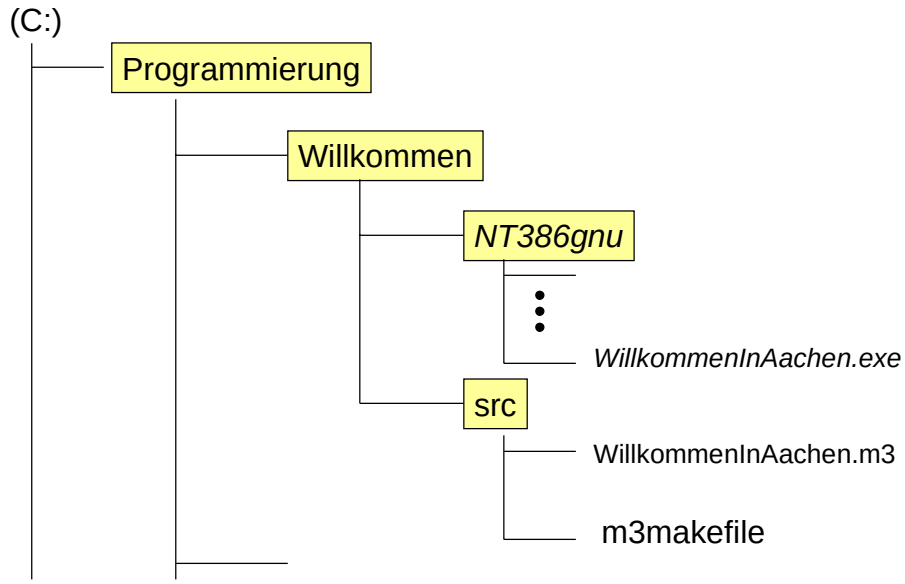
In der Datei <x>.m3 ist ein Modul implementiert

Verwende <x>.exe als Dateiname für das ausführbare Programm

Verzeichnisstruktur für m3build

- **src-Verzeichnis**
  - Hier muss die Datei, die den Programmtext enthält, abgelegt sein.
  - Diese muss die Endung "**.m3**" haben. Beispiel: Willkommen.m3
  - In diesem Verzeichnis sucht m3build die Datei m3makefile. Dementsprechend muss diese dort vorhanden sein.
  - Muss vom Programmierer angelegt werden.
  - Name unter Windows und UNIX: "**src**"
- **exe-Verzeichnis**
  - Wird automatisch von m3build angelegt (auf der selben Ebene wie das src-Verzeichnis!)
  - Nimmt alle Ergebnisse von m3build auf, insbesondere die ausführbare Datei. Diese hat z.B. die Endung "**.exe**"
  - Verzeichnisname unter Windows: "NT386gnu"
    Solaris: "SOLgnu"
    LINUX: "LINUXLIBC6"

Beispiel für Verzeichnisstruktur



Ein interaktives Programm

```
MODULE WillkommenInX EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
   Autor           : Horst Lichter, RWTH Aachen
*)

IMPORT SIO;

VAR ort : TEXT;
BEGIN
  SIO.PutText ("Bitte Ort angeben: ");
  ort := SIO.GetLine();
  SIO.Nl();
  SIO.PutText("---------------------------------");
  SIO.Nl();
  SIO.PutText("Willkommen zum Studium in ");
  SIO.PutText(ort);
  SIO.Nl();
  SIO.PutText("---------------------------------");
  SIO.Nl();
END WillkommenInX.
```


Fehler

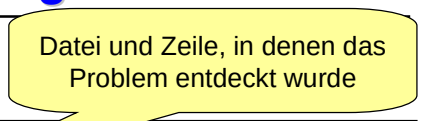
Datei und Zeile, in denen das Problem entdeckt wurde

```
--- building in ..\NT386GNU ---
new source -> compiling ..\src\WillkommenInX.m3
"..\src\WillkommenInX.m3", line 19: Initial module name
doesn't match final name (WillkommenInY)

compilation failed => not building program "WillkommenInX"
m3build: quake error:
```

- **Übersetzer gibt einen Fehler aus**
  - wenn eine Unstimmigkeit entdeckt wird
  - und die vollständige Übersetzung daher nicht mehr möglich ist
- **Ein guter Übersetzer lokalisiert Fehler sehr genau**
  - dennoch muss die angezeigte Stelle nicht Problemursache sein

Warnung




```
--- building in ..\NT386GNU ---
new source -> compiling ..\src\WillkommenInY.m3
"..\src\WillkommenInY.m3", line 19: warning: file name
(WillkommenInY.m3) doesn't match module name (WillkommenInX)

new "WillkommenInY.mo" -> linking WillkommenInX.exe
```

- **Der Compiler gibt eine Warnung aus**
  - wenn eine Unstimmigkeit entdeckt wird
  - die vollständige Übersetzung jedoch möglich ist
- **Warnungen**
  - zeigen an, daß etwas nicht stimmt und sollten geprüft werden
  - können aber oft ignoriert werden

Zusammenfassung

- **Editor und Compiler (PM 3) sind das Rüstzeug für die Programmierung**
- **Modula-3 erwartet eine feste Verzeichnisstruktur, beachten Sie diese!**
- **Für Ein-/Ausgaben verwenden wir das Modul SIO**
  - Import im Programm: `IMPORT SIO;`
  - im m3makefile mit: `import("libSIO")`
- **Achten Sie auf ordentlichen Programmierstil:**
  - Einrückungen
  - Wahl aussagekräftiger Bezeichner
  - Kommentare!!

Modula-3
Programmierkonventionen

Die wichtigsten Bestandteile der Programmierkonventionen für Modula-3 werden hier erläutert. Dies sind:

- Schreibweise von Bezeichnern
- Verwendung von Leerzeichen
- Einrückkonventionen
- Kommentare

Einleitung

- **Spielraum** bei der Gestaltung der **Details**
- **Richtlinien** sollen dazu beitragen, daß die **Programme stets leicht zu verstehen** sind.
- **Lesbarkeit** und **Wartbarkeit** sind wichtig.
- Es gibt zwar keinen "korrekten" Programmierstil, aber die folgenden Konventionen haben sich erfolgreich durchgesetzt.

Bezeichner - 1

- **Bezeichner** müssen (bis auf Laufvariablen in Schleifen, etc...) **aussagekräftig** sein**.**
  - Negativbeispiele:

```
MODULE A ...
PROCEDURE B ...
```

```
MODULE Gaius ...
PROCEDURE Julius ...
```


Bezeichner - 2

- Bei **Bezeichnern**, die **aus mehreren Worten** zusammengesetzt werden, muß der erste Buchstabe eines Wortes jeweils groß geschrieben oder durch "_" getrennt werden.
  - Beispiele:
    roemischeZahl, gebeTextEin, gebe_Text_ein
  - Negativbeispiel:
    diesisteinschlechterbezeichnerfuereinezahl

Bezeichner - 3

- **Bezeichner für folgende Komponenten müssen mit einem Großbuchstaben beginnen:**
  - Modulnamen
  - Prozeduren und Funktionen
  - Typen
  - Konstanten
  - Elemente von Aufzählungstypen
- **Bezeichner für alle anderen Komponenten müssen mit einem Kleinbuchstaben beginnen. Dies sind:**
  - Variablen
  - Formale Parameter
  - Record-Komponenten
  - Alias-Namen in WITH-Anweisung

Bezeichner - 4

- Bezeichner, die aus mehr als einem Buchstaben bestehen, sollten nicht nur aus Großbuchstaben bestehen. Dieses kann zu **Konflikten mit von Modula-3 als Bezeichner belegten Wörtern** führen.
- Derartige Wörter können jedoch benutzt werden, wenn Groß-/Kleinschreibung verändert wird.
  - Beispiel: begin, real dürfen als Variablennamen benutzt werden.
  - Negativbeispiel: BEGIN, REAL dürfen nicht als Variablennamen benutzt werden.

Verwendung von Leerzeichen

- **Leerzeichen stets vor und nach einem Operator**.
  - Beispiel: x := (a + b) * c + F(x);
  - Negativbeispiel: x:=(a+b)*c+F(x);

- **Ein Leerzeichen steht immer nach:**
  - geschlossener Klammer)
  - Doppelpunkt :
  - Semikolon **;**
    - Beispiel:

```
PROCEDURE P(x, y: REAL; int: INTEGER) =
```

    - Negativbeispiel:

```
PROCEDURE P(x,y:REAL;int:INTEGER)=
```


Einrückung - allgemein

- Jede **Ebene** wird **um 2 Leerzeichen eingerückt** (kein Tabulator!!!).

- Die Festlegung der Ebenen ist durch die **Einrückkonventionen** gegeben.

- Durch die Einhaltung der Einrückkonventionen wird die **Struktur** der Programme leichter erfaßbar.

Einrücken bei Modulen und Prozeduren

```
MODULE Test EXPORTS Test;
<Importe>
<Deklarationen>
BEGIN
 <Anweisungen>
END Test.
```

Die Anweisungen zwischen den Bezeichnern BEGIN und END sind eine Ebene tiefer als diese.

```
PROCEDURE P(x, y: REAL; int: INTEGER) =
 <Deklarationen>
 BEGIN
  <Anweisungen>
 END P;
```

Die Definition der Prozedur ist eine Ebene tiefer.

Nach den Deklarationen beginnt keine neue Ebene.

Einrücken bei Interface und Variablendeklarationen

```
INTERFACE Test;
<Importe>
<Deklarationen>
END Test.
```

Hier findet kein Ebenenwechsel statt.

```
VAR
 x, y, z: Typ := <Ausdruck>
 a: [1..10];
```

Die Deklaration der Variablen ist eine Ebene tiefer als der Bezeichner VAR.

Einrücken bei Schleifen

```
FOR i := a TO b DO
  <Anweisungen>
END;

REPEAT
  <Anweisungen>
UNTIL <Ausdruck>;

WHILE <Ausdruck> DO
  <Anweisungen>
END;
```

Die Anweisungen der Schleife sind eine Ebene tiefer als Schleifenkopf und -ende anzuordnen.

Einrücken bei if-Ausdrücken

```
IF <Ausdruck> THEN
  <Anweisungen>
ELSEIF <Ausdruck> THEN
  <Anweisungen>
ELSE
  <Anweisungen>
END;
```

Die Anweisungen sind eine Ebene tiefer als Bedingungsteile der if-Anweisung.

Kommentare - 1

- **(* steht zu Beginn eines Kommentars.**
- ***) steht am Ende eines Kommentars.**
- **Ein Leerzeichen steht nach (* und vor *).**
  - Beispiel: (* ein kurzer Kommentar in einer Zeile *)
- Beim **Auskommentieren von Quelltext** (d.h. der betreffende Quelltext soll nicht beachtet werden) sollte folgende Formatierung verwendet werden:

```
(* Die folgende(n) Codezeile(n) werden
auskommentiert.
          IF (x < a) THEN y := b;
*)
```


Kommentare - 2

- Für jeden Modul soll zu Beginn kurz die **Funktionalität** beschrieben werden.
- Zusätzlich soll der Kopfkommentar die Informationen über den **Autor**, die **Entwicklungsumgebung**, sowie das **Datum** der **Erstellung** und der **letzten Änderung** enhalten.
- Daher ist für jeden Modul ein Kopfkommentar der folgenden Form zu erstellen:

Kommentare - 3

```
MODULE Test EXPORTS Main;
(* Dieses Programm zeigt einen Willkommensgruss
    Autor : Horst Lichter, RWTH Aachen
    Umgebung : SRC-Modula-3 rel. 3.6, Windows NT 4.0
    Erstellt : 16.08.98
    Letzte Aenderung: 20.08.98
*)

IMPORT <Importe>
[...]
BEGIN
 <Anweisungen>
END Test.
```


Kommentare - 4

- Nur **sinnvolle Kommentare** sollten eingefügt werden.
- Kommentare sollten **keine redundante Information** enthalten.
  - Negativbeispiel:

```
(* weise c die Summe von a und b zu *)
c := a + b
```

- Kommentar **vor dem Quelltext**, der kommentiert wird.
- Geht aus den Bezeichnern nicht die Aufgabe der bezeichneten Variablen bzw. Parameter hervor, so sollte dieser im Prozedurkopf erläutert werden

Kommentare - 5

- Kommentare dienen dem Verständnis (ebenso wie der übrige Teil der Programmierrichtlinien) des Programmes. Entsprechend **verständlich** sollten die Kommentare verfaßt sein.
  - Beispiel:

```
(* wenn der Rechner nicht benutzt werden kann,
dann soll <Anweisung> ausgeführt werden
*)
IF (Stromausfall OR Stecker_draussen OR
Rechner_kaputt OR ... ) THEN
  <Anweisung>
```


Zusammenfassung - 1

- **Programme müssen gut verständlich sein**
  - "Wir programmieren nicht für uns, sondern für andere"
  - Lesbarkeit und Wartbarkeit
  - Programmierkonventionen müssen daher beachtet werden
  - dies gilt auch für die Vorlesung "Programmierung"
- **Bezeichnerwahl ist sehr wichtig**
  - Programme leichter verständlich
  - geringere Fehlerhäufigkeit
- **Leerzeichen**
  - helfen bei der Erfassung der Einheiten eines Ausdrurck

Zusammenfassung - 2

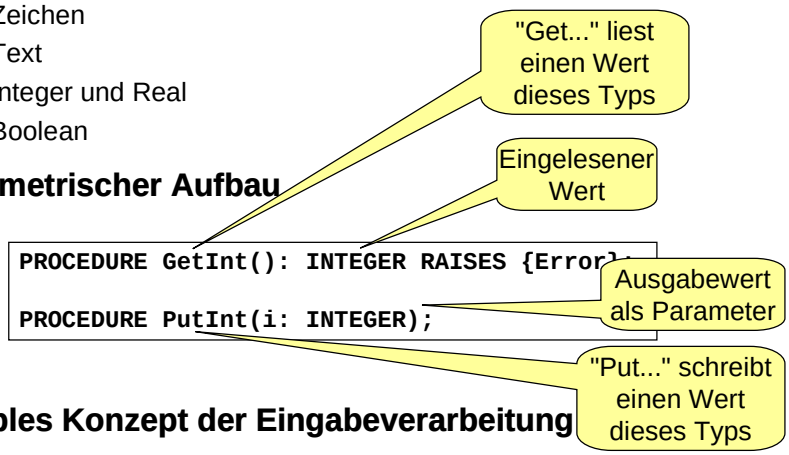
- **Einrückungen**
  - helfen, die Übersicht zu behalten
  - die Struktur des Programmes schneller zu erfassen
- **Aussagekräftige Kommentare**
  - helfen, schwierige Stellen zu verstehen
  - sind sehr wichtig für die Wartung

Ein-/Ausgabe mit dem Modul SIO

- Modul
- Konzept der Ein-/Ausgabe
- PutChar/GetChar
- Textein-/ausgabe
- Ein-/Ausgabe von Zahlen
- Zusammenfassung

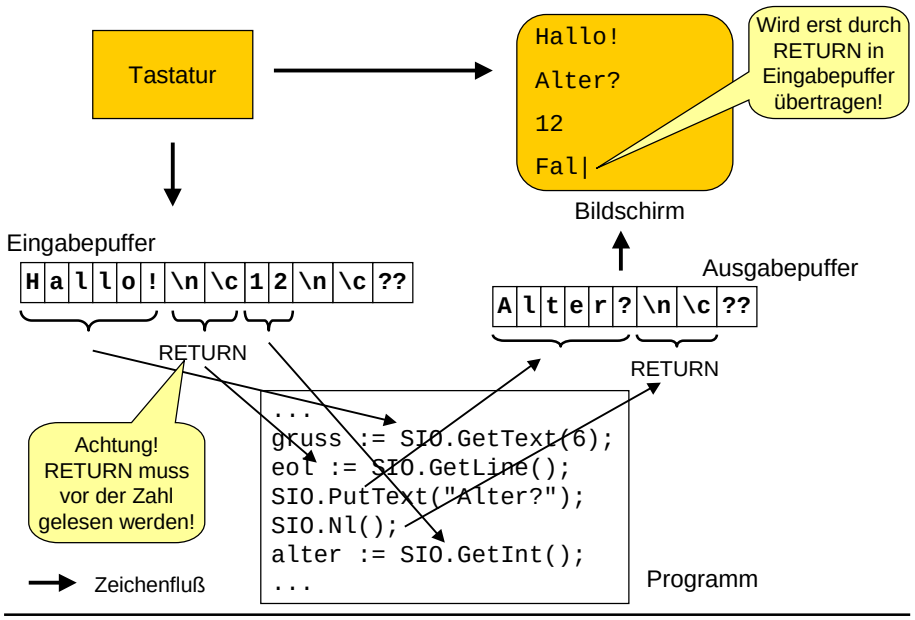
Das Modul SIO

- SIO = Simple Input/Output
- dient der Ein- und Ausgabe von Daten
  - Zeichen
  - Text
  - Integer und Real
  - Boolean
- Symmetrischer Aufbau



- Simples Konzept der Eingabeverarbeitung

Konzept der Ein-/Ausgabe



Put-/GetChar

```
VAR c: CHAR;

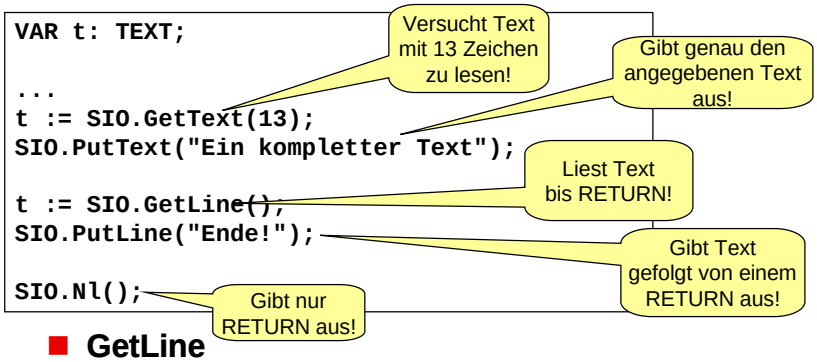
...
c:= SIO.GetChar();
...
SIO.PutChar('H'); SIO.PutChar('a'); ...
```

- **GetChar**
  - liest Zeichen einzeln aus dem Eingabepuffer
  - RETURN zählt ebenfalls als ein (oder zwei) Zeichen
- **PutChar**
  - schreibt die Zeichen einzeln in den Ausgabepuffer
- **Elementare Ein-/Ausgabeoperation**
  - restliche Operationen des SIO-Moduls bauen darauf auf
  - für Anwendungen in der Regel zu simpel

Solaris/Linux

Windows

Textein-/ausgabe



- **GetLine**
  - das RETURN wird aus dem Eingabepuffer entfernt
  - auch verwendbar, um nur das RETURN zu entfernen
- **PutText**
  - erlaubt Frage und Eingabe auf derselben Zeile
  - eine Zeile mit mehreren Aufrufen zusammensetzbar

Ein-/Ausgabe von Zahlen

```
VAR i: INTEGER; r: REAL;

...
i := SIO.GetInt();
SIO.PutInt(123);

r := SIO.GetReal();
SIO.PutReal(1.345E-2);
```

- **GetInt/GetReal**
  - erzeugen aus den Zeichen im Eingabepuffer eine gültige Zahl
  - es werden soviele Zeichen wie möglich interpretiert
  - falls keine gültige Zahl im Eingabepuffer $\Rightarrow$ Fehler!
- **PutInt/PutReal**
  - geben den übergebenen Wert aus
- **Analoge Operationen für weitere Elementartypen**
  - LongReal, Boolean, ...

Die Interna

- **Komplette Schnittstelle des SIO Moduls**
  - auf unseren WWW-Seiten (Abteilung Modula-3)
  - in der Bibliothek des PM3 (von der Erstsemester-CD)
    Pfad: **`C:\usr\pm3\lib\m3\pkg\libSIO\src\SIO.i3`**

```
PROCEDURE GetInt(rd: Reader := NIL): INTEGER RAISES {Error}
```

- **Modul kann mehr als wir benötigen**
  - neben Standardein-/ausgabe auch Dateien verwendbar
  - bietet weitere Operationen
- **Fehlerbehandlung**
  - es gibt zwei vorcompilierte Versionen des SIO-Moduls (PM3)
  - welche verwendet wird, hängt vom **`m3makefile`** ab
    - ohne Erzeugung von Ausnahmen: **`import("libSIO")`**
    - mit Ausnahmen: **`import("libinf1")`**
  - beide Versionen haben genau denselben Funktionsumfang

Zusammenfassung

- **Modul SIO**
  - dient der Ein-/Ausgabe von Daten
  - hat einen möglichst symmetrischen Aufbau
- **Konzept der Ein-/Ausgabe**
  - alle Zeichen landen im Eingabepuffer
    - auch das RETURN nach einer Zahleneingabe!
  - das Modul SIO liest die Zeichen, wie sie kommen!
- **Verschiedene Operationen**
  - helfen komplexere Daten ein-/auszugeben (z.B. REAL-Zahl)
  - werden im Rahmen der Vorlesung nicht alle benötigt
  - die Wichtigsten sind hier dargestellt

Das Modul Text

- **Modul**
- **Length und Empty**
- **GetChar**
- **FromChar und Sub**
- **SetChars und FromChars**
- **Zusammenfassung**

Das Modul Text - 1

- Für die Bearbeitung vieler Problemstellungen ist es notwendig, Operationen auf Text durchzuführen.
- In Modula-3 wird der Datentyp TEXT zur Verfügung gestellt. Objekte dieses Typs sind endliche Zeichenfolgen (Strings genannt). Sie müssen in Anführungsstrichen (“ “) geschrieben werden.
- Ein Großteil der Operationen auf dem Typ TEXT liefert das Bibliotheks-Modul Text.

Das Modul Text - 2

- Die wichtigsten Operationen ermöglichen:
  - Bestimmung der Länge eines Strings
  - Auswahl von Zeichen aus einem String
  - Auswahl von Teil-Strings
  - Umwandlung von Zeichen in Strings
  - Umwandlung zwischen Zeichen-Array und String

Das Text Interface

Ausschnitt aus dem Interface mit den wichtigsten Operationen:

```
INTERFACE Text;
TYPE T =TEXT; (* Text.T entspricht TEXT *)
PROCEDURE Cat(t, u: T): T;
PROCEDURE Equal(t, u: T): BOOLEAN;
PROCEDURE GetChar(t: T; i: CARDINAL): CHAR;
PROCEDURE Length(t: T): CARDNAL;
PROCEDURE Empty(t: T): BOOLEAN;
PROCEDURE Sub(t: T; start, length: CARDINAL): T;
PROCEDURE SetChars(VAR a: ARRAY OF CHAR; t: T);
PROCEDURE FromChar(c: CHAR): T;
PROCEDURE FromChars(READONLY a: ARRAY OF CHAR): T;
...
END Text.
```

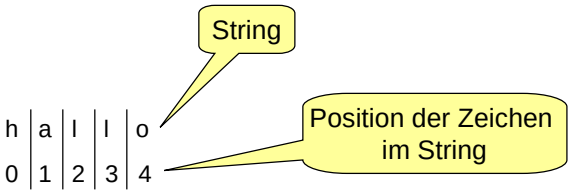

Length und Empty

```
VAR t: TEXT;
    i: CARDINAL;
...
t := SIO.GetLine();
...
IF Text.Length(t) = i THEN ... END;
...
IF Text.Empty(t) THEN ... END;
...
```

- **Length**
  - liefert zu einem String die Anzahl der enthaltenen Zeichen, d.h. die Länge des Strings
  - die gelieferte Zahl ist vom Typ CARDINAL
- **Empty**
  - liefert TRUE für einen gegebenen String str, wenn Text.Length(str) = 0

Position von Zeichen

- Die Position eines Zeichens in einem String wird durch eine Zahl vom Typ **CARDINAL** repräsentiert.
- Das erste Zeichen hat die Position 0, das zweite die Position 1, ...
- Wenn Test.Length(str) gleich n ist (n > 0), dann hat der das letzte Zeichen die Position (n-1).
- Beispiel:



Zeichenauswahl

```
VAR t: TEXT;
    c: CHAR;
    i: CARDINAL;
...
t := SIO.GetLine();
...
c := Text.GetChar(t, i);
...
```

- **GetChar**
  - erhält einen String und eine Zahl vom Typ CARDINAL
  - ein Fehler tritt auf, wenn die gegebene Zahl größer oder gleich der Länge des Strings ist oder kleiner als
  - liefert das Zeichen, welches im String an der Position der Zahl steht
  - Beispiel: Text.GetChar(“hallo”, 2)

FromChar

```
VAR s: TEXT;
    c: CHAR;
    ...
c := SIO.GetChar();
...
s := Text.FromChar(c);
...
```

- **FromChar**
  - erhält ein Zeichen (Typ: CHAR) als Eingabe
  - liefert einen String (Typ: TEXT) zurück, der nur aus dem angegebenen Zeichen besteht

Sub - 1

```
VAR s, t: TEXT;
    x, y: CARDINAL;
...
s := SIO.GetLine();
...
t := Text.Sub(s, x, y);
...
```

- **Sub**
  - erhält einen String und zwei Zahlen (erste Zahl: x, zweite Zahl: y) vom Typ CARDINAL als Eingabe
  - liefert den String zurück, der aus den y Zeichen besteht, die bei Position x beginnen
  - Beispiel: Text.Sub(“hallo“, 1, 2) liefert den String „al“.

Sub - 2

- **Die Operation Sub hat mehrere zu beachtende Spezialfälle:**
  - Wenn eine der beiden Zahlen kleiner als 0 ist, tritt ein Fehler auf.
  - Der leere String wird zurückgegeben, wenn x (die Position) größer gleich der Länge des Strings ist oder y = 0 (die Länge des auszugebenden Strings).
  - Wenn der eingegebene String nicht genügend Zeichen enthält (d.h., x + y > Text.Length(string)), dann wird der bei x beginnende (und bis zum Ende des Strings reichende) Teil-String ausgegeben.

SetChars und FromChars

- Diese beiden Prozeduren dienen der **Konvertierung** zwischen Strings und Arrays mit Elementen vom Typ CHAR.
- **SetChar**
  - erhält ein Array mit Elementen vom Typ Char (im Folgenden array genannt) und einen String (im folgenden string genannt)
  - füllt array so lange mit Elementen von string, bis array voll ist oder kein Zeichen mehr in string übrig
  - Füllt string array nicht komplett auf, so bleiben die übrigen Elemente unverändert
  - array muß veränderbar sein (d.h. nicht read-only)
- **FromChar**
  - erhält ein Array mit Elementen vom Typ CHAR
  - liefert einen String, der aus den Elementen des Arrays besteht

Zusammenfassung - 1

- **Modul Text**
  - dient der Bearbeitung von Text
  - hier wurde nur der wichtigste Teil vorgestellt
  - das komplette Interface befindet sich in der Datei Text.i3
- **Verschiedene Operationen**
  - ermöglichen die Betrachtung der Länge eines Strings
  - ermöglichen den Zugriff auf einzelne Zeichen oder Zeichenfolgen im String
  - ermöglichen die Konvertierung in / aus “verwandten” Typen

Zusammenfassung - 2

- **weitere Operationen**
  - liefern zu eine Hash-Funktion zu einem String
  - vergleichen zwei Strings bzgl. der lexikographischen Ordnung
  - liefern zu einem zu String und einem Zeichen die Position des ersten Auftretens des Zeichens in dem String
  - liefern zu einem zu String und einem Zeichen die Position des letzten Auftretens des Zeichens in dem String

Dateien in Modula-3

- **Dateisystem**
- **Operationen auf Dateien**
  - Lesen
  - Schreiben
- **Dateien in Modula-3**
  - wichtige Datei-Operationen
- **Konvertierung**

Dateien: Zweck

- **Verarbeiten von Daten**
  - Daten müssen in vielen Fällen dauerhaft (***persistent***) gespeichert werden
- **Bisher:**
  - Daten wurden im ***Arbeitsspeicher*** zur Laufzeit des Programms erzeugt
  - Nachdem das Programm beendet ist, sind diese Daten ***verloren***
  - "***flüchtiger***" Speicher
- **Hintergrundspeicher**
  - persistenter Speicher
  - Diskette, CD, Festplatte etc.
- **Frage:**
  - Wie können wir Daten aus dem Arbeitsspeicher in den Hintergrundspeicher bringen und umgekehrt?

Dateisystem

- **Betriebssystem:**
  - stellt ***Dienstleistungen*** zur Verfügung, damit der Umgang mit dem Rechner einfach und mit ***bestimmten Diensten*** möglich ist
  - eine angebotene Dienstleistung des Betriebssystems ist das ***Dateisystem***
- **Dateisystem**
  - erlaubt, den Hintergrundspeicher in ***einzelne Bereiche*** aufzuteilen
  - diese nennt man ***Dateien*** (file)
  - Dateien können in sogenannten ***Verzeichnissen*** gruppiert werden (directory)
  - jede Datei hat einen ***Namen***
  - aus der Sicht des Rechners ist eine Datei eine Folge von ***Informationseinheiten*** (Bytes)
  - ein Verzeichnis verwaltet für alle seine Dateien den Namen und die Position, wo die Dateien physisch auf dem Hintergrundspeicher liegen

Dateien: Zweck und Formen

- **Dateien können**
  - angelegt und gelöscht werden
  - gelesen und geschrieben werden
- **Dabei gibt es zwei Dateiformen:**
  - ***Sequentiell***
    - Daten werden von Anfang bis zum Schluß der Datei ***nacheinander*** gelesen (geschrieben).
    - Es gibt keine Möglichkeit, einen Ausschnitt der Datei zu ***überspringen***
  - ***Direkt***
    - Es kann eine ***Position*** angegeben werden, ab der gelesen (geschrieben) werden soll
    - Wechseln der Position ist ***beliebig*** möglich
- **Das Betriebssystem**
  - kennt für jede Datei einen Zähler, der die aktuelle Position kennzeichnet.

Arbeiten mit Dateien - allgemein

- **Programmiersysteme**
  - bieten in der Regel Möglichkeiten und Mechanismen, um vom Programm aus ***Dateioperationen*** ausführen zu können
  - dabei werden i.d.R. nicht die Funktionen des Betriebssystems benutzt
  - Programmiersysteme stellen eine ***abstrakte Schnittstelle*** zum Dateisystem zur Verfügung

Arbeiten mit Dateien - Wichtige Dateioperationen

- **Datei *öffnen***
  - öffnen zum Lesen
  - öffnen zum Schreiben (von vorne)
  - öffnen zum Schreiben (neue Zeichen werden hinten angehängt)
- **neue Datei *erzeugen***
- ***lesen* und *schreiben***
- **Abfragen des *Dateiendes* (end of file, EOF)**
- **Datei *schließen***
  - um sicherzugehen, daß alle Änderungen in einer Datei persistent sind

Sprachumgebung, E/A-Strom

- **Sprachumgebung von Modula-3**
  - stellt Module zur Verfügung, um Dateien manipulieren zu können
- **Ein- / Ausgabestrom**
  - Mit einem Ein- / Ausgabestrom kann auf eine Datei zugegriffen werden
  - In der Standardbibliothek sind diese definiert
    - `reader, writer`
  - Ein Ein- / Ausgabestrom kann bei seiner Initialisierung mit einer Datei verbunden werden

```
IMPORT IO, Rd, Wr;
VAR eingabestrom : Rd.T;
    ausgabestrom : Wr.T;
…
eingabestrom := IO.OpenRead ("eingabe.txt");
ausgabestrom := IO.OpenWrite("ausgabe.txt");
```


Dateioperationen in Modula-3 - Das Modul IO

- PROCEDURE **EOF** (rd: Rd.T := NIL): BOOLEAN;
  - Liefert TRUE gdw. das Ende der Datei rd erreicht wurde
- PROCEDURE **OpenRead** (f: TEXT): Rd.T;
  - Öffnet die Datei mit dem Namen f zum Lesen und liefert einen Eingabestrom auf ihren Inhalt.
  - Wenn die Datei nicht existiert oder sie nicht gelesen werden kann, so wird NIL geliefert.
- PROCEDURE **OpenWrite** (f: TEXT): Wr.T;
  - Öffnet die Datei mit dem Namen f zum Schreiben liefert einen Ausgabestom auf ihren Inhalt.
  - Wenn die Datei nicht existiert, so wird sie geschaffen.
  - Darf der Prozeß die Datei nicht modifizieren oder erschaffen, dann wird NIL zurückgeliefert.

Dateioperationen in Modula-3 - Das Modul Rd (1)

- Eine Variable vom Typ **Rd.T** identifiziert einen Eingabestrom.
- PROCEDURE **GetChar** (rd: T): CHAR RAISES {EndOfFile, Failure, Alerted};
  - Liest ein Zeichen vom Eingabestrom rd an der aktuellen Position und erhöht dessen aktuelle Position um 1.
- PROCEDURE **EOF** (rd: T): BOOLEAN RAISES {Failure, Alerted};
  - Liefert TRUE gdw. das Ende der Datei rd erreicht ist.
- PROCEDURE **UnGetChar** (rd: T): CHAR RAISES {};
  - Legt das zuletzt gelesene Zeichen auf Rd zurück.

Dateioperationen in Modula-3 - Das Modul Rd (2)

- PROCEDURE **Close** (rd: T) RAISES {Failure, Alerted};
  - Schließt rd,
  - d.h. alle mit rd verbundenen Ressourcen werden freigegeben und closed(rd) wird auf TRUE gesetzt.
- PROCEDURE **GetLine** (rd: T): TEXT RAISES {EndOfFile, Failure, Alerted};
  - Liest so viele Zeichen von rd, bis das Ende von rd erreicht wurde oder ein Zeilenvorschubzeichen gelesen wurde.
  - Die Ausnahme EndOfFile wird generiert, wenn das Ende von rd schon vor der Operation erreicht war.
  - Die gelesenen Zeichen werden zurückgegeben. Das Zeilenvorschubzeichen wird nicht im Ergebnis abgelegt.
- **Weitere Operationen zur**
  - Bestimmung der aktuellen Position von rd
  - Bestimmung der Länge von rd
  - ...

Dateioperationen in Modula-3 - Das Modul Wr (1)

- Eine Variable vom Typ **Wr.T** identifiziert einen Ausgabestrom.
- PROCEDURE **PutChar** (wr: T; ch: CHAR) RAISES {Failure, Alerted};
  - Schreibt das Zeichen ch auf den Ausgabestrom wr und erhöht dessen aktuelle Position um 1.
- PROCEDURE **Close** (wr: T) RAISES {Failure, Alerted};
  - Schließt wr,
  - d.h. alle mit wr verbundenen Ressourcen werden freigegeben und closed(wr) wird auf TRUE gesetzt.
- PROCEDURE **PutText** (wr: T; t: TEXT) RAISES {Failure, Alerted};
  - Schreibt alle Zeichen von t auf wr.

Dateioperationen in Modula-3 - Das Modul Wr (2)

- PROCEDURE **Flush** (wr: T) RAISES {Failure, Alerted};
  - Leert den Puffer von wr.
- **Weitere Operationen zur**
  - Bestimmung der Länge von wr
  - Bestimmung der aktuellen Position von wr
  - ...

Konvertierung

- Rd und Wr haben nur Prozeduren zum ***Lesen und Schreiben von Zeichen*** (Text ist eine Sequenz von Zeichen)
- Sollen Zahlen oder Boolesche Werte gespeichert werden, so müssen diese erst formatiert werden, d.h.
  - Konvertierung der Werte in jeweilige Text-Repräsentation
  - beim Lesen Konvertierung des Textes in jeweiligen Typ („scannen“)
- Dieses leisten die Module Fmt und Scan.

Das Modul Fmt

- Mit Hilfe der Prozeduren der Schnittstelle Fmt können Zahlen und andere Daten ***in TEXT umgewandelt*** werden.
- Dies wird u.a. geliefert für folgende Typen:
  - BOOLEAN (Prozedur Bool)
  - INTEGER (Prozedur Int)
  - CHAR (Prozedur Char)
  - REAL (Prozedur Real)
- Weitere Prozeduren zur Formatierung wie z.B. Längenanpassung, ...

Das Modul Scan

- Mit Hilfe der Schnittstelle Scan können Zahlen und andere Daten ***aus TEXT-Variablen gelesen*** werden.
- Die Prozeduren lesen den TEXT-Parameter und konvertieren dessen Inhalt in Werte des jeweiligen Typs.
- Führende Leerzeichen und bei Zahlen führende Nullen werden weggelassen.
- Dies wird u.a. geleistet für die folgenden Typen:
  - BOOLEAN (Prozedur Bool)
  - INTEGER (Prozedur Int)
  - REAL (Prozedur Real)
  - CHAR (Prozedur Char)

Was haben wir gelernt?

- **Dateien für die persistente Speicherung**
- **Dateisysteme als Teil des Betriebssystems**
- **Dateiformen (sequentielle / direkte Dateien), Modi (Lese-, Schreibdateien)**
- **interne Dateien, externe Dateien und Bindung / Auflösung der Bindung beim Öffnen oder Schließen**
- **Anschluß von Dateien an das Programmiersystem über vordefinierte E/A-Bausteine**
- **Dateioperationen für Dateiformen, Eingabe- oder Ausgabedateien**
- **Konvertierung**

Glossar

- **Hintergrundspeicherarten**
- **flüchtiger, persistenter Speicher**
- **Betriebssystem, Dateisystem, Anschluß von seiten des Programmiersystems**
- **Datei (file), Verzeichnis (directory)**
- **externe, interne Dateien, Namen für beides**
- **Dateiformen, Eingabe-/Ausgabedatei**
- **Dateioperationen, Öffnen, Schließen, Lesen, Schreiben, Positionierung, Abfrage Dateiende**
- **Einschränkung der Operationen bei bestimmten Dateiformen, Ein- oder Ausgabedateien**

Zusammenfassung OO-Konzepte (1)

- **Klasse**
  - **Implementierter ADT**
  - **Zentrales Konstruktionselement (Statik)**
- **Abstrakte Klasse**
  - **Partiell implementierter ADT**
  - **Dient zur Konstruktion von Klassenhierarchien**
  - **Spezifiziert Objektverhalten**

Zusammenfassung OO-Konzepte (2)

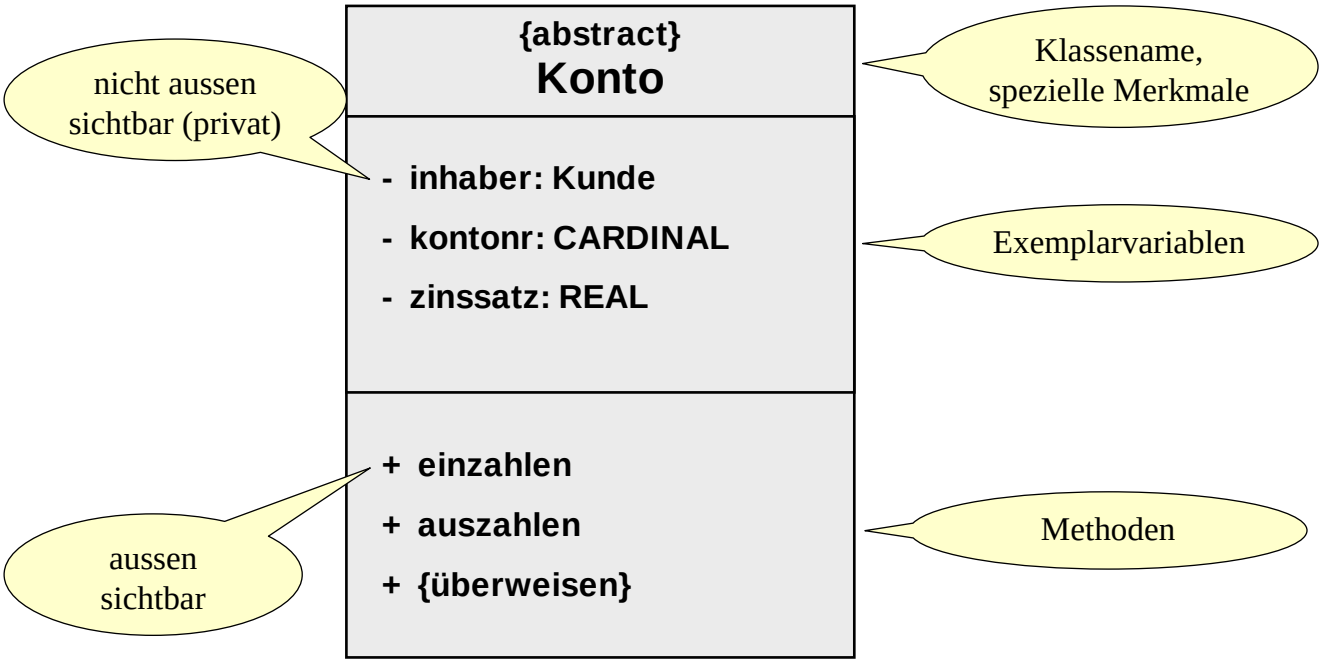
- **Objekt**
  - **Exemplar einer Klasse**
  - **Existiert zur Laufzeit**
  - **Klassenzugehörigkeit kann nicht wechseln**
- **Vererbung**
  - **Programmtechnische Umsetzung der Spezialisierung**
  - **Etabliert einer Oberklasse-Unterklasse-Beziehung**
  - **Dies entspricht einer Subtyp-Beziehung**

Notation für OO-Programme

- **OO-Programme bestehen aus einer Vielzahl von Klassen → UML zur Visualisierung von OO-Systemen**
- **Statische Sicht auf das System**
  - **Ein System besteht aus Klassen**
  - **Beziehung zwischen Klassen müssen modelliert werden**

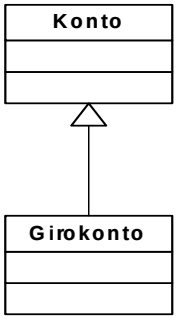
  → Klassendiagramme

UML-Klassendiagramme (1)

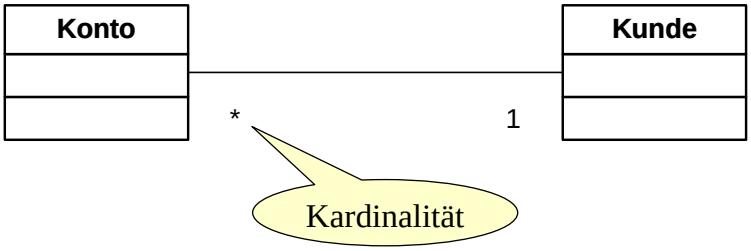


UML-Klassendiagramm (2)

- **Generalisierung (Vererbung)**
  - **Klasse Girokonto erbt von der Klasse Konto**



- **Allgemeine Beziehungen**



PS-Grundlagen

- **Eigenschaften eines Algorithmus**
- **Syntax**
- **Semantik**
- **Grammatik**
- **EBNF**
- **Syntaxdiagramme**

Funktionale Programmierung

- **Funktionen**
- **Parameter**
- **Datentypen (`INTEGER, CARDINAL, REAL, BOOLEAN, CHAR, TEXT`)**
- **Rekursion**

Imperative Programmierung

- **Variablen (Wertzuweisung)**
- **Konstanten**
- **Prozeduren**
- **Call-by-value**
- **Call-by-reference**
- **Gültigkeitsbereich und Lebensdauer**

Kontrollstrukturen

- **Fallunterscheidungen (`IF-THEN-ELSE`, `CASE`)**
- **Schleifen**
  - `FOR`
  - `WHILE`
  - `REPEAT-UNTIL`

Statische Datentypen

- **Unterbereichstyp**
- **Aufzählungstyp**
- **Feld (**ARRAY**)**
- **Verbund (**RECORD**)**
- **Menge (**SET OF**)**

Dynamische Datentypen

- **Zeigertyp**
- **Lineare Liste**
- **Sortierte lineare Liste**
- **Prozedurtyp**

Testen

- Black-Box-Test
  - Äquivalenzklasssenbildung
  - Grenzwertüberprüfung
- White-Box-Test
  - Ablaufgraph
  - Anweisungsüberdeckung (C0)
  - Zweigüberdeckung (C1)
  - Pfadüberdeckung (C2)

Ausblick

- Modulkonzept
- Datenabstraktion
  - Objektmodul
  - Abstrakter Datentyp (ADT)
- Vetragsmodell
- Objektorientierung
- LISP
- PROLOG