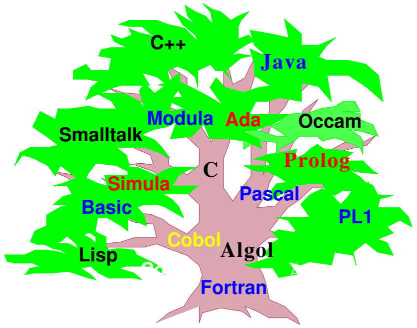
Prof. Dr. E. Rahm

Programmierung und Programmiersprachen

Wintersemester 1997/98

Universität Leipzig

Institut für Informatik



Zur Vorlesung allgemein

- **Vorlesungsskript**
  - im WWW: *http://www.informatik.uni-leipzig.de* Unterpunkt: Datenbanken; -> Lernmaterialien
  - ersetzt keinesfalls die Vorlesungsteilnahme !
  - ersetzt nicht zusätzliche Benutzung von Lehrbüchern
- **Übungen**
  - selbständige Lösung der Übungsaufgaben wesentlich für Lernerfolg
  - praktische Übungen am Rechner (HP-Pool URZ)
  - rechtzeitig abgegebene Lösungen der Übungsaufgaben werden korrigiert
  - Vergabe eines Übungsscheines erfolgt, wenn die meisten Aufgaben korrekt gelöst wurden
  - Informatiker: Übungsschein ist Voraussetzung für VD-Prüfung; schriftliche VD-Teilprüfung (Klausur)
  - Übungsblätter im WWW

Grundstudium Diplomstudiengang Informatik

Fachgebiet	1. Semester	2. Semester	3. Semester	4. Semester
Praktische Informatik (16 SWS)	Einführung in die Informatik (2 + 2)	Algorithmen und Datenstrukturen (2 + 2)	**Programmierung und Programmiersprachen (2 + 2)**	Einführung in die Software-Technologie (2) + Projektierungsseminar (2)
Hard- u. Software-Grundlagen (12 SWS)	Elektrotechnische Grundlagen (2 + 1)	Rechneraufbau (2 + 1)	Betriebssysteme und Compilertechnik (2 + 1)	Maschinennahe Programmierung (2 + 1)
Praktische Übungen (6 SWS)		C-Praktikum (2)	Hardware-Praktikum (2)	OOP-Praktikum (2)
Theoretische Informatik (14 SWS)	Mengenth./Algebr. Grundlagen (3 + 1)	Automaten und Sprachen (2 + 1)	Berechenbarkeits- und Komplexitätstheorie (2 + 1)	Logik (3 + 1)
Mathematik (26 SWS)	Lineare Algebra/ Analyt. Geometrie (4 + 2)	Analysis I (4 + 2)	Analysis II (4 + 2)	Wahrscheinlichkeitstheorie/ Stochastik (3 + 1) Numerik (3 + 1)
Nebenfach (12 SWS)	entsprechend Nebenfachangebot			

❑ Weitere Teilnehmergruppen
- Wirtschaftsmathematiker
- Magisterstudium mit 2. Hauptfach Informatik
- Magisterstudium mit NF Informatik
- Diplomstudiengang Mathematik mit NF Informatik
- sonstige Diplomstudiengänge mit NF Informatik

Vorläufiges Inhaltsverzeichnis

1. Einführung
- Programmiersprache
- Sprachübersetzung (Interpretation vs. Kompilierung)
- Programmierparadigmen

2. Überblick zu imperativen Programmiersprachen
- Historische Entwicklung (FORTRAN, ALGOL, COBOL, PL1, Pascal, C, ...)
- Daten- und Kontrollabstraktionen
- Typsystem

3. Grundkonzepte objektorientierter Programmierung

4. Objektorientierte Programmiersprachen

5. Ausnahmebehandlung

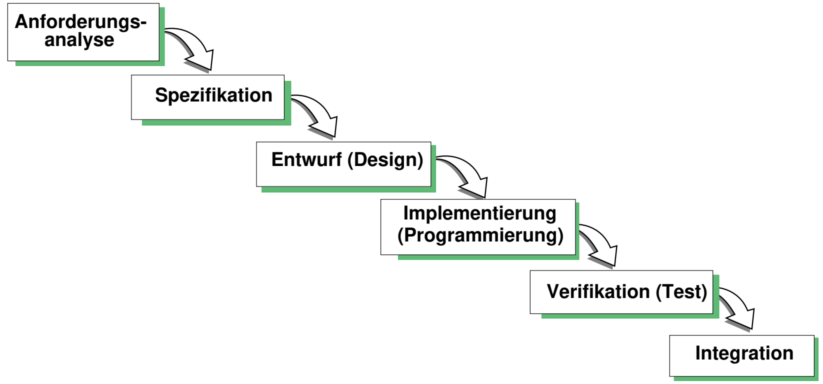
6. Funktionale und logische Programmierung

7. Parallele Programmierung

Literatur

- Allgemeine Lehrbücher
  - Louden, K.C.: *Programmiersprachen*, Int. Thomson Publishing, 1994
  - Sebesta, R.W.: *Concepts of Programming Languages,* 2nd edition, Benjamin Cummings, 1993
  - Wilson, L.B., Clark, R.G.: *Comparative Programming Languages,* 2nd edition, Addison-Wesley, 1993
  - Meyer, B.: *Objektorientierte Softwareentwicklung.* Hanser-Verlag, 1990
  - Witt, K.-U-: *Einführung in die objektorientierte Programmierung.* Oldenbourg 1992
  - Pomberger, G., Blaschek, G.: *Software Engineering.* Hanser-Verlag, 2. Auflage, 1996
- Zahlreiche Lehrbücher zu einzelnen Programmiersprachen (Java, C++, ...)
- MeDoc: die elektronische Informatik-Bibliothek

Der Software-Entwicklungsprozeß



- abschließender Schritt: Nutzung und Wartung (Maintenance) der erstellten Software
- Wahl der Programmiersprache beeinflußt auch frühe Phasen

Programmiersprache

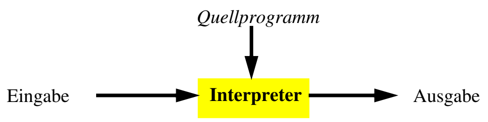
- Sprache zur Formulierung von Rechenvorschriften, d.h. von Datenstrukturen und Algorithmen, die von einem Computer ausgeführt werden können
- Hauptzweck einer Programmiersprache: Erstellung zuverlässiger Software
- Anforderungen an Programmiersprache
  - hohe Ausdrucksmächtigkeit (expressive power)
  - Einfachheit und Orthogonalität
  - gute Implementierbarkeit
  - frühzeitige Fehlererkennung
  - Unterstützung von Verfikation bzw. Tests
  - standardisierte Sprachbeschreibung (-> Portabilität)
- Syntax und Semantik einer Programmiersprache müssen eindeutig definiert sein
  => Festlegung in einer Sprachdefinition
- Programmiersprachen sind i.a. kontextsensitive Sprachen
  - Syntax wird meist formal durch kontextfreie Grammatik beschrieben
  - umgangssprachliche Beschreibung von Kontextabhängigkeiten und der Semantik

Programmiersprache (2)

- Historische Unterscheidung zwischen
  - Maschinensprachen
  - niederen bzw. maschinenorientierten Sprachen (Assembler)
  - höheren bzw. problemorientierten Programmiersprachen
- Maschinenorientierte Sprachen (Assembler)
  - direkte Entsprechung zu Maschinenbefehlen
  - Verwendung von Symbolen und mnemonischen Ausdrücken
  - niedriger Abstraktionsgrad; stark maschinenabhängig
- über 1000 höhere Programmiersprachen
- Sprachen der 4. und 5. Generation: deklarative Ansätze

Sprachübersetzung

- Quellprogramm in höherer PS muß in Maschinenprogramm transformiert werden (Translation)
- 2 Varianten: Interpretation oder Kompilierung
- Interpretation: Translation und Ausführung in einem Schritt

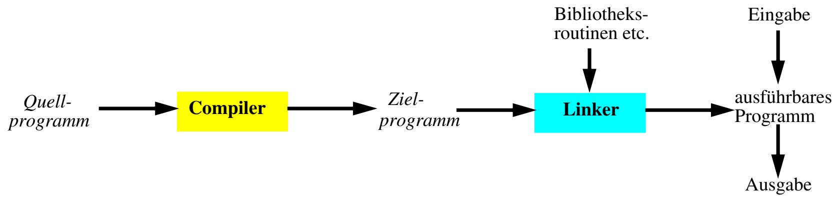


- interaktive Programmänderungen möglich
- Abfragen von Variablenwerten etc.
- schnelle Programmentwicklung
- Einsatz v.a. bei stark dialogorientierten Sprachen, z.B. BASIC, LISP, PROLOG, APL

- jede Anweisung wird vor ihrer Ausführung transformiert
- hoher Aufwand bei wiederholter Ausführung einer Anweisung
- Umsetzungsprozeß verlängert Programmlaufzeit

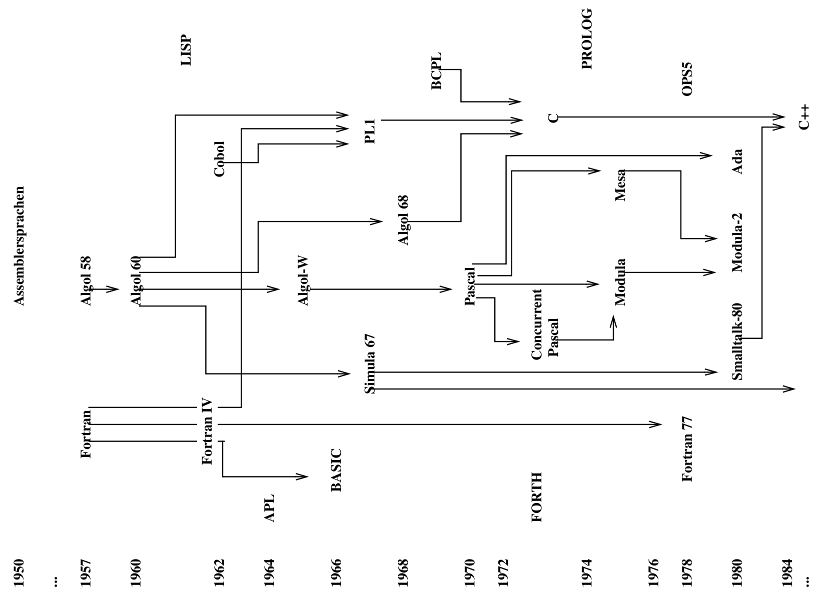
Sprachübersetzung (2)

- Kompilierung: getrennte Translation und Ausführung



- Teilschritte der Übersetzung
  - lexikalische Analyse (Scanner)
  - syntaktische Analyse (Parser)
  - semantische Analyse und Code-Generierung
  - Code-Optimierung
- Arten von Programmfehlern
  - Syntaxfehler
  - sonstige (semantische) Übersetzungsfehler
  - logische Fehler (Laufzeitfehler)

Entwicklung von Programmiersprachen



Sprachübersetzung (3)

- Compiler realisiert bedeutungswahrende Abbildung
  Quellsprache A -> Zielsprache B
- Quellsprache meist mächtiger als Zielsprache (z.B. bei Erzeugung von Maschinenprogrammen)
- 1-1-Übersetzer: Übersetzung zwischen gleichmächtigen Programmiersprachen
- Prä-Compiler: Vorübersetzung für erweiterte Programmiersprache im Rahmen eines mehrstufigen Übersetzungsprozesses
- De-Compiler (Umkehr-Compiler): Zielsprache ist mächtiger als Quellsprache
- Assemblierer: Übersetzer einer maschinenorientierten Sprache in eine Maschinensprache
- Compiler-Compiler: Generierung von Compilern aus Sprachbeschreibung
- Inkrementelle Übersetzung
  - bei Änderungen im Quellprogramm wird Zielprogramm unmittelbar angepaßt
  - kombiniert Vorteile der Interpretation und Kompilierung
  - Einsatz v.a. im Rahmen von Programmierumgebungen

Programmierparadigmen (Denkmodelle)

- Imperative (prozedurale) Programmiersprachen
- Objektorientierte Programmiersprachen
- Funktionale (applikative) Programmiersprachen
- Logik-Programmiersprachen (Prädikative PS)
- Weitere Ansätze
  - Regelbasierte Programmiersprachen
  - Spezialsprachen
  - Parallele Programmierung
  - Visuelle Programmierung

Imperative Programmierung

- ausgerichtet auf von-Neumann-Rechnerarchitektur
  (1 CPU, welche sequentiell Instruktionen auf im Speicher gelagerten Werten ausführt)
- Programm besteht aus Folge von *Befehlen* (lat. imperare = befehlen)
- Kennzeichnende Merkmale imperativer Sprachen:
  - sequentielle Ausführung von Befehlen
  - *Variablenkonzept* : Eingabewerte werden in Variablen gespeichert und weiterverarbeitet
  - *Zuweisungsoperation* (assignment statement) zur Belegung von Variablen
- wichtigstes Programmierparadigma in der Praxis (FORTRAN. COBOL, Pascal, C, Modula, ...)
- Beispiel in Modula-2:

```
PROCEDURE GGT (m, n: INTEGER, VAR x: INTEGER);
VAR    r: INTEGER;              (* es sei n > 0 *)
BEGIN
     r := m MOD n;
     WHILE r <> 0 DO
          m:= n; n := r;
          r:= m MOD n
     END;
     x := n;
END GGT;
```


Funktionale Programmierung

- Programme entsprechen mathematischen Funktionen
  - Eingabewerte werden als Parameter beim Funktionsaufruf übergeben
  - Ergebniswerte entsprechen Rückgabewerten der Funktionen
- Berechnungen erfolgen durch Auswertung von Funktionen oder Anwendung von Funktionen auf bekannten Werten (=> applikative Sprachen)
  - Funktionen können als Parameter auftreten
  - bestimmte Grundfunktionen sind vorgegeben
- Unterschiede zu imperativer Programmierung:
  - keine Variablen und Zuweisungen !
  - iterative Ausführung erfolgt nicht durch Schleifen, sondern nur durch Rekursion !
  - größere Unabhängigkeit von Maschinenarchitektur
- Hauptvertreter: LISP (LISt Processor) und Derivate (Common Lisp, Scheme)
  - LISP wurde bereits in den späten 50er Jahren am MIT entwickelt (J. McCarthy)
  - Verwendung von Listenstrukturen (S-Ausdrücke) und Funktionen
  - dynamische Speicherverwaltung (garbage collection)
  - Verwendung v.a. im Bereich der Künstlichen Intelligenz
- weitere funktionale Sprachen: ML, Miranda

Funktionale Programmierung (2)

- funktionales Beispiel in Modula:

```
PROCEDURE GGT (m, n: INTEGER): INTEGER;
BEGIN
      IF n = 0 THEN RETURN m;
      ELSE RETURN GGT (n, m MOD n);
END GGT;
```

  - keine lokalen Variablen, keine Zuweisungen, keine Schleife
- Realisierung in LISP-Variante SCHEME

```
(define (ggt m n)
     (if (= n 0) m
      (ggt n (remainder m n))))
```

  - LISP-Programme sind Listenausdrücke, die von Klammern umschlossen sind
  - erstes Element jedes Ausdrucks ist eine Funktion, die auf die folgenden Argumente angewendet wird
  - auch if-Funktion gibt Wert zurück: (if x y z) bedeutet if x then y else z

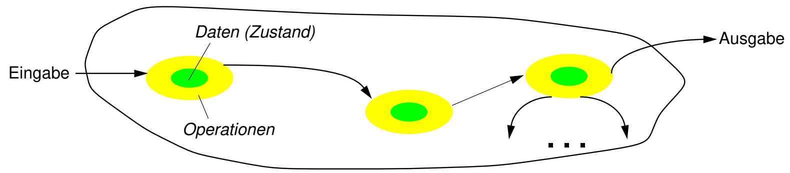
Logische Programmierung

- andere Bezeichnungen: logikbasierte Programmierung, prädikative Programmierung
- Programmausführung entspricht Beweisen in einem System von Tatsachen und Schlußfolgerungen
- Festlegung von *Fakten* (gültige Prädikate) und *Regeln* (Aussagen, um aus Fakten neue Fakten zu erzeugen)
- zu einer gegebenen *Abfrage*, wird Wahrheitswert NEIN oder JA ermittelt
- keine Festlegung durch Kontrollstrukturen wie Berechnung zu erfolgen hat !
  => deklarative Programmierung
- Beispiel einer logischen Programmiersprache: PROLOG
  - Programm besteht aus Klauseln der Form a :- b, c, d
    (a ist wahr, wenn b, c und d wahr sind)
  - Beispiel:

```
ggt (M, N, M)      :-    N = 0.
ggt (M, N, X)      :-    N > 0,
                         Y is M mod N,
                         ggt (N, Y, X).
```


Objektorientierte Programmierung

- Daten und Anweisungen werden zu *Objekten* zusammengefaßt
- Grundkonzepte:
  - Geheimnisprinzip (Kapselung) und Datenabstraktion (Abstrakte Datentypen, Objektklassen)
  - Klassenhierarchien und Vererbung
  - Polymorphismus (dynamisches Binden)
- Objekte kommunizieren über Nachrichten, welche durch ADT-Operationen (Methoden) bearbeitet werden



- wesentliche Vorteile: Erweiterbarkeit und Wiederverwendbarkeit (-> Klassenbibliotheken)
- Beispiele objektorientierter Sprachen: Simula, Smalltalk, C++, Eiffel, Oberon, Java

Objektorientierte Programmierung (2)

- GGT-Beispiel (C++)

```
class GGT
 { public:

     GGT (int a, int b)               // Konstruktor
     { int n, m, r;
       if (a > b) { G = a; K = b;} else { G = b; K = a;}
       n = K; m = G;
       r = m % n;         // die mod-Operation in C bzw. C++
       while (r != 0) { m = n; n = r; r = m % n; }
       ggtGK = n;
     }

     int GGTWert ()                   // Lese-Funktion (reader)
             { return ggtGK; }

   private: int G, K, ggtGK;
};
•••
GGT g (64, 28);
int i;
•••
i = g.GGTWert();
```


Prozedurorientiertes Denken

- Programme bestehen aus Prozeduren, welche Eingabedaten in Ausgabedaten transformieren

- *Beispiel:* x := Flaeche (f)
- Berechnung der Fläche für unterschiedliche Figuren verlangt jedoch verschiedene Prozeduren, z.B. `RechteckFlaeche`, `DreieckFlaeche`, `KreisFlaeche`, etc.
- umständliche Auswahl der “richtigen” Flächenprozedur

  IF *f ist ein Rechteck* THEN x := RechteckFlaeche (f)
  ELSIF *f ist ein Dreieck* THEN x := DreieckFlaeche (f)
  ELSIF *f ist ein Kreis* THEN x := KreisFlaeche (f) . . .

- umständliche Erweiterung, z.B. zusätzlicher Figurentyp Ellipse (Änderung aller Fallunterscheidungen)

Objektorientiertes Denken

- Daten stehen im Mittelpunkt

Auftrag → **Objekt** → Daten

- Programm besteht aus Objekten, welche bestimmte Aufträge ausführen können
- Objekte interpretieren Aufträge auf eigene Weise und führt selbst die richtige Operation aus
- unterschiedliche Objektarten (Rechtecke, Dreiecke, ...) können den gleichen Auftrag verarbeiten

  x := f.Flaeche

- bessere Erweiterbarkeit um weitere Objektarten (keine Änderung bestehender Aufrufe, weniger Fallunterscheidungen)

Weitere Ansätze

- Regelbasierte Programmiersprachen
  - Verwendung von “wenn-dann-Regeln” zur Beschreibung von Zustandsübergängen, z.B. der Form “wenn <Bedingung>, dann <Aktion> ausführen”
  - Anwendbarkeit mehrerer Regeln führt zu Nichtdeterminismus
  - Beispiel: OPS5
- Spezialsprachen
  - Spezifikationssprachen
  - Datenbankanfragesprachen (z.B. SQL)
  - Simulationssprachen (z.B. GPSS)
  - statistische Auswertungen (z.B. SPSS)
  - Stringverarbeitung (z.B. SNOBOL)
  - Systemprogrammierung (z.B. BCPL, BLISS, XPL)

    •••
- Parallele Programmierung
  - Ziele: parallele bzw. asynchrone Ausführung von Programm(teil)en auf Mehrprozessorsystem
  - Anweisungen zur Kommunikation und Synchronisation
  - Unterstützung in mehreren Programmiersprachen: Modula, Ada, Occam, ...

Zusammenfassung

- jede Programmiersprache basiert auf bestimmtem Denkmodell (Paradigma)
- wesentliche Programmierparadigmen

Stil	Beschreibung	Definitionsform	Aktivierungskonzept	Ausführungs-konzept	Beispiele
imperativ	prozedural	Prozedur	Aufruf	Zuordnung mit Variablenkonzept	Pascal, C
applikativ	funktional	Funktion	Anwendung mit Parametervermittlung	Wertebestimmung	LISP, Miranda
deklarativ	logisch	Regelsystem	Unifikation	Ableitung (Beweis)	PROLOG
direktiv	objektorientiert	Objektklasse	Instantiierung	Kommunikation zwischen Objekten	Smalltalk, Java

- keine generelle Überlegenheit einzelner Paradigmen
- Eignung bestimmt von Problemklasse, Einfachheit der Programmierung, Effizienz, ...