Einführung in die Programmiertechnik

Programmiersprachen

Algorithmen und Programme

- Computer führt Berechnungen auf deterministische Weise aus
  - bei gleicher Eingabe und gleichem inneren Zustand erzeugt er gleiche Ausgabe
- Art und Weise der Berechnung wird durch ein Programm festgelegt
  - formale Sprachen: Notation des Programms folgt festen Regeln, jedes Programm hat mathematisch klare Bedeutung
- historisch: Formulierung mit Hilfe von Operationen der Hardware
  - Maschinensprache, Assembler
- heute: Abstraktion mithilfe höherer Programmiersprachen
- Algorithmus: genau definierte Handlungsvorschrift zur Lösung eines Problems
- imperative Programme: Anweisungen an den Computer
- funktionale Programme: Ziel der Berechnung wird durch mathematische Funktion beschrieben
  - Definition der Funktion mithilfe vorhandener Funktionen
- deklarative Programme: Eigenschaften der gesuchten Lösung werden definiert
  - konkreter Algorithmus zur Lösung wird nicht vorgegeben

Imperative Sprachen

- BASIC
- Pascal
- C
- C++
- Java
- Python
- ...
- Imperative Sprachen beinhalten oft funktionale oder deklarative Elemente
  - z.B. Funktionen höherer Ordnung

Funktionale Sprachen

- LISP
- Scheme
- ML (Meta Language)
- Miranda
- Haskell
- OPAL
- FORTRAN
- ...
- Funktionale Sprachen beinhalten oft imperative Elemente
  - z.B. Ein/Ausgabe

Deklarative Sprachen

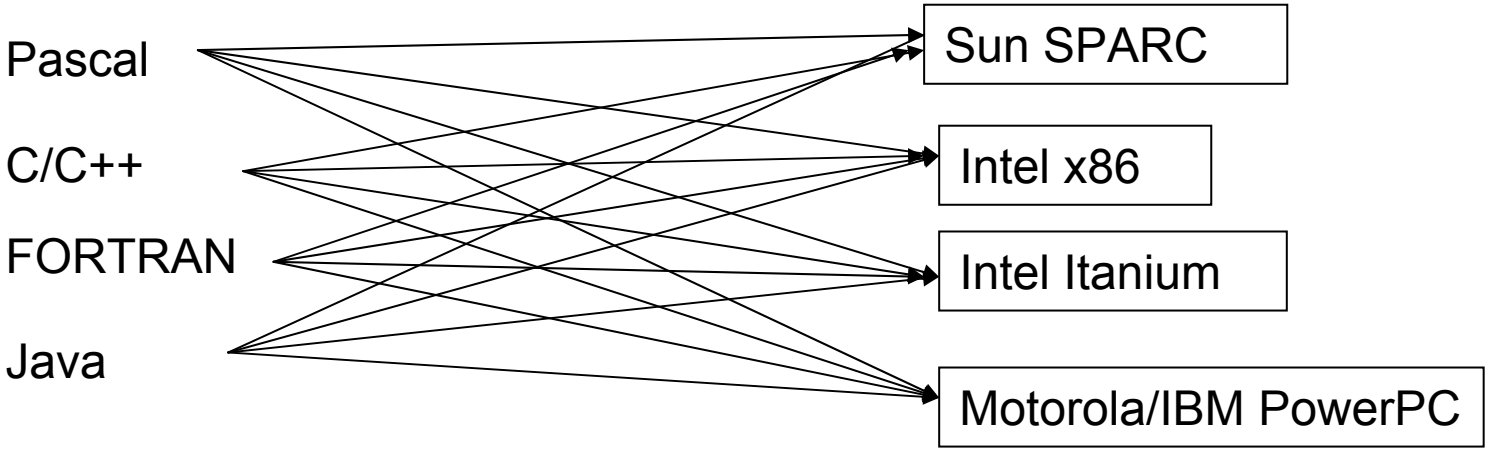
- Prolog
- SQL
- WSDL
- nicht alle deklarativen Sprachen sind „Programmiersprachen“
  - deklarativer Teil oft eingebettet in Gesamtprogramm (SQL, WSDL)
- deklarative Programmiersprachen enthalten oft imperative Teile
  - z.B. Ein/Ausgabe

Vom Programm zur Maschine

- Programme sind initial in Textdateien gespeichert
  - Quelltext
- Ausführung des Programms auf der Maschine muss Elementaroperationen verwenden
  - Daten aus dem Speicher lesen/in den Speicher schreiben
  - elementare arithmetische Operationen ausführen
  - Ausführung an anderer Stelle fortsetzen (Sprünge)
- Quelltext muss in Maschinencode übertragen werden

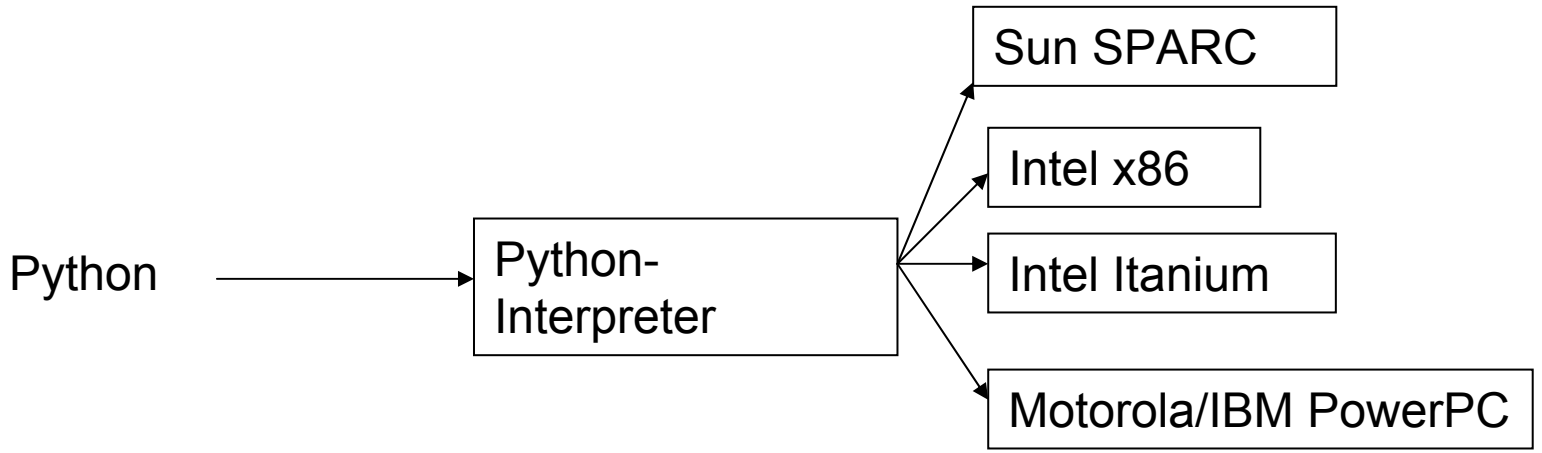
Compiler

- hier: Beschränkung auf imperative Sprachen
- einzelne Anweisungen aus dem Quelltext werden in Folgen von Maschinenanweisungen übersetzt
- Zur Ausführung des Programms werden „direkt“ die Maschinenanweisungen abgearbeitet



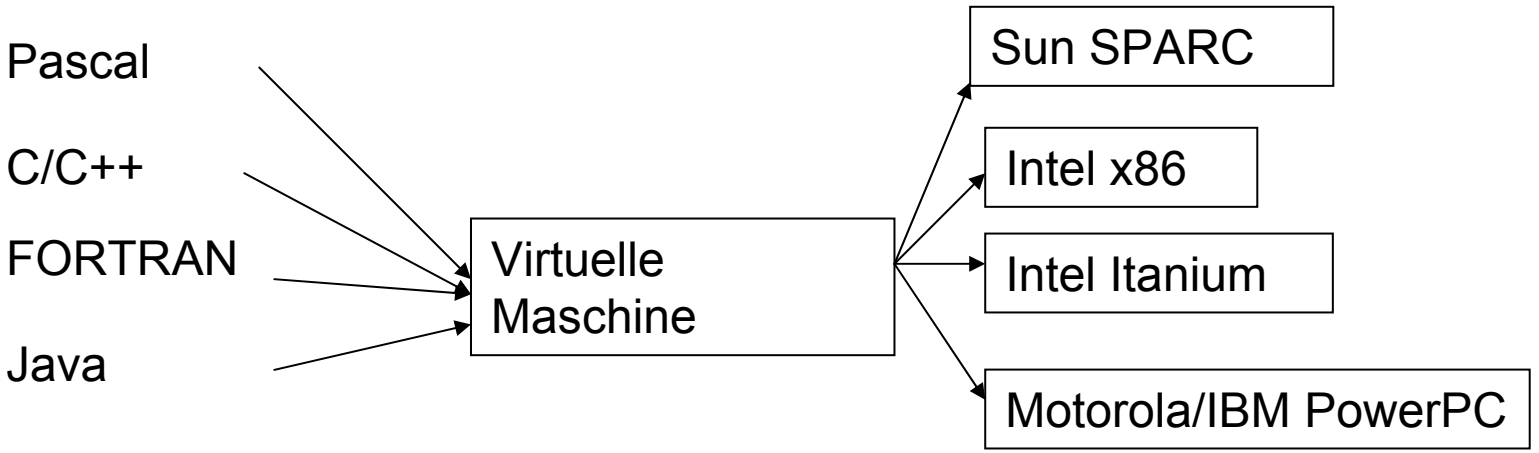
Interpreter

- Interpreter führt Anweisungen des Programms „direkt“ aus (ohne Übersetzung des Programms in Maschinensprache)
  - „virtuelle Maschine“ (VM): Elementaranweisungen werden unabhängig von konkreter Hardware definiert; Interpreter setzt VM-Anweisungen in Maschinenanweisungen um
    - VM-Anweisungen werden oft „Byte-Code“ genannt



Ideale Virtuelle Maschine

- Virtuelle Maschine könnte unabhängig von Programmiersprache sein
  - Teilweise erreicht in Microsoft .NET
- Implementierung durch Interpreter oder Compiler
  - „just-in-time compiler“: VM-Anweisungen werden im Moment der Ausführung in Maschinencode übertragen



Interpreter vs. interpretierte Sprachen

- Für jede Sprache ist jede Umsetzungsstrategie denkbar
- für viele Sprachen ist aber „praktisch“ festgelegt, ob sie interpretiert oder kompiliert werden
  - Pascal, Modula, COBOL, Fortran, C, C++, Ada: Compiler
  - Smalltalk, Python: Interpreter
  - Java, C#: Interpreter oder Just-in-time-Compiler, mit separatem Bytecodecompiler

Portabilität von Programmen

- Ziel: Das gleiche Programm soll auf verschiedener Hardware, verschiedenen Betriebssystemen gleichartig funktionieren
- Compiler-Lösung: Compiler für wohldefinierte (normierte) Sprachen
  - Zur Portierung auf ein neues System muss der Quelltext verfügbar sein
- VM-Lösung: Interpreter/Just-In-Time-Compiler für Zielplattform
  - Zur Portierung genügt u.U. der Bytecode
    - Java: „write once, run anywhere“

Programmierfehler

- Eingabe des Programms in Texteditor
- Übergabe des Programms an Compiler/Interpreter
  - Analyse durch Compiler zeigt Verletzung der Sprachsyntax an
    - Syntaxfehler
      - z.B. Schreibfehler in Schlüsselwörtern
  - Analyse zeigt u.U. auch Verletzung von Benutzungsregeln an
    - Semantikfehler
    - Beispiel: Verletzung des Typsystems (Typfehler)
      - z.B. Verknüpfung von Strings und Zahlen in einer Additionsoperation
- Bei Ausführung wird das Programm u.U. aufgrund von Fehlern abgebrochen
  - Laufzeifehler
- Selbst bei Beendigung ohne Fehler ist u.U. das Ergebnis nicht das erwartete
  - Denkfehler

Programmierfehler (2)

- Fehler im Programm heißen auf Englisch *bugs*
- Entfernen von Fehlern: *Debugging*
- Entwicklungszyklus: Editieren, Kompilieren, Testen
- E. Dijkstra (1970): Program testing can be used to show the presence of bugs, but never to show their absence!

Programmierumgebungen

- Ziel: Verkürzung des Entwicklungszyklus
- Strategien:
  - Interpreter: interaktiver Modus (z.B. Python)
  - Compiler: Integration von Quelltexteditor, Compiler (*build process*), Debugger, evtl. Quelltextverwaltungssystem, ...
    - engl. IDE (Integrated Development Environment)
  - Editor: Vereinfachung der Eingabe von Programmen
    - automatische Vervollständigung
    - Syntax-Hervorhebung (*highlighting*)
    - Generierung von Codeblöcken (*wizards*)
- Beispiele:
  - Java: Eclipse, netbeans, JBuilder, BlueJ
  - C, C++: Visual Studio, C++Builder
  - Python: IDLE, PythonWin

Fortran

- (auch: FORTRAN)
  - **For**mula **Tran**slator
- in den 1950ern entwickelt, unter Leitung von John W. Backus
  - für IBM 704 (36-bit-Worte, erster Rechner mit Gleitkommahardware)
  - erste Hochsprache (neben LISP)
- ursprünglich: Orientierung auf Lochkartenprogrammierung
  - Jede Lochkarte enthält eine Zeile
  - Spalten 1..5 enthalten die Zeilennummer, Spalten 7..72 Programmtext
  - Kontrollflußanweisung: GOTO
- neuere Versionen bereinigen und vereinfachen die Syntax
  - GOD is REAL (unless declared INTEGER)
- Sprachversionen: FORTRAN II (1958), FORTRAN IV (1961), FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003

Pascal

- zwischen 1968 und 1974 von N. Wirth entwickelt
  - für Unterrichtszwecke
- strukturierte Programmierung, Rekursionen, Unterprogramme
- später um weitere Konzepte erweitert
  - Turbo-Pascal: Modularisierung (units)
  - Object-Pascal, Delphi: Objekt-Orientierung
- ISO/IEC 7185:1983, ISO/IEC 10206:1990 (Extended Pascal)

C

- ursprünglich zwischen 1969 und 1973 an den Bell Labs entwickelt
  - Nachfolger von B
  - erstes Ziel: Entwicklung von Software für PDP-7
- erstes Standardwerk: The C Programming Language
  - Brian Kernighan, Dennis Ritchie (K&R)
- Standardisierung begann 1983; erster Standard 1989 (ANSI)
  - 1990 von ISO übernommen: ISO/IEC 9899:1990
- Überarbeitung des Standards: ISO/IEC 9899:1999
  - Sprachversionen werden C89 und C99 genannt

C++

- Entwickelt in den 1980ern von Bjarne Stroustrup
  - ursprünglich „C with Classes“
- Ziel: Objektorientierte Programmierung in C
  - Klassen, Methoden, Mehrfachvererbung
  - später auch Templates, Ausnahmebehandlung, Namespaces
- Standardisierung: ISO/IEC 14882:1998

Java

- 1991 entwickelt von James Gosling
  - ursprünglich „oak“ genannt
  - Weiterentwicklung von C++
- entwickelt als „reine“ objekt-orientierte Sprache
  - Klassen sind das Hauptstrukturierungskonzept
- Sprachversionen werden von Sun definiert
  - aktuelle Version: Java 5

LISP

- **List P**rocessing Language
- entwickelt 1958 von John McCarthy
  - erstmalig implementiert von Steve Russel für IBM 704
  - erster LISP-Compiler 1962 (gemischter Interpreter/Compiler-Betrieb)
- funktionale Sprache: Ziel der Berechnung wird als Funktionsausdruck formuliert
- primäre Datentypen: Listen, Zahlen, „Atome“
- von Anfang an für KI (Künstliche Intelligenz) verwendet
- ursprünglich viele Sprachversionen (InterLisp, Franz Lisp, ...)
  - ANSI-Standard 1994: „Common Lisp“

Prolog

- um 1972 von Alain Colmerauer und Robert Kowalski entwickelt
  - „programming in logic“
- logische Programmiersprache
  - basierend auf dem Prädikatenkalkül (PK1)
- verwendet für KI und Computerlinguistik
- ISO/IEC 13211:1995