1 Einführung

In diesem einführenden Kapitel wollen wir einige grundlegende Begriffe im Zusammenhang mit dieser Vorlesung erläutern. In dieser Vorlesung geht es um Programmiersprachen, daher fangen wir damit an.

1.1 Programmiersprachen

Eine **Programmiersprache** ist eine Notation für Programme, d. h. eine (formale) Beschreibung für Berechnungen. In diesem Sinn ist ein **Programm** eine Spezifikation einer Berechnung und die **Programmierung** ist die Formalisierung von Algorithmen und informellen Beschreibungen.

1.2 Berechnungsmodelle

Jede Programmiersprache basiert auf einem bestimmten **Berechnungsmodell**, welches beschreibt, wie Programme abgearbeitet werden. Daher können wir Programmiersprachen auch nach diesen Modellen klassifizieren:

Maschinensprachen Die basieren direkt auf dem klassischen von Neumann-Rechner d.h. dort sind alle Details wie Register, Speicher, Adressen u.ä. direkt sichtbar.

Assembler Vom Berechnungsmodell sind diese sehr ähnlich zu Maschinensprachen, aber sie stellen folgende Abstraktionen bereit:

- Abstraktion von Befehlscode (symbolische Namen)
- Abstraktion von Speicheradressen (symbolische Marken)

Höhere Programmiersprachen Diese abtrahieren von den zu Grunde liegenden Maschinen und sind stärker am Problem orientiert. Übliche Abstraktionen sind:

- Ausdrücke (mathematische Notation)
- Berechnungseinheiten (Prozeduren, Objekte, ...)

Weil höhere Programmiersprachen das Ziel haben, von der reinen Maschine zu abstrahieren, kann man damit auch sehr unterschiedliche Berechnungsmodelle realisieren. Aus diesem Grund kann man hierbei z.B. noch die folgende Klassifikation vornehmen:

Imperative Programmiersprachen

- Berechnung $\approx$ Folge von Zustandsänderungen.

- Zustandsänderung ≈ Zuweisung, Veränderung einer Speicherzelle (immer noch von-Neumann-Rechner!)

Funktionale Programmiersprachen

- Programm ≈ Menge von Funktionsdefinitionen
- Berechnung ≈ Ausrechnen eines Ausdrucks

Logische Programmiersprachen

- Programm ≈ Menge von Relationen
- Berechnung ≈ Beweisen einer Formel

Nebenläufige Programmiersprachen

- Programm ≈ Menge von Prozessen, die miteinander kommunizieren
- Berechnung ≈ Kommunikation, Veränderung der Prozessstruktur

1.3 Programmierparadigmen

Ein **Programmierparadigma** bestimmt die Art der Programmierung bzw. der Programmstrukturierung. Häufig wird das Programmierparadigma bereits durch die verwendete Programmiersprache bestimmt (z. B. imperatives, funktionales oder logisches Programmierparadigma). Aber ein Programmierparadigma ist nicht zwangsläufig an eine bestimmte Programmiersprache gebunden, es ist auch möglich

- funktional in einer imperativen Programmiersprache zu programmieren, oder
- imperativ in einer funktionalen/logischen Programmiersprache zu programmieren (und Zustände als Parameter durchzureichen).

Neben diesen drei genannten existieren noch einige weitere bekannte Programmierparadigmen:

Strukturierte Programmierung (Pascal)

- Zusammenfassen von Zustandsübergangsfolgen zu Prozeduren
- keine Sprünge („gotos considered harmful“ (Dijkstra, 1968)), sondern nur: Zuweisung, Prozeduraufrufe, bedingte Anweisung, Schleifen, teilweise Ausnahmebehandlung

Modulare Programmierung (Modula-x, Ada)

- Programm ≈ Menge von Modulen
- Modul ≈ Schnittstelle und Implementierung
- Benutzung eines Moduls ≈ Benutzung der Schnittstellenelemente

Objektorientierte Programmierung (Simula, Smalltalk, C++, Eiffel, Java)

- Programm: Menge von Klassen ≈ Schema für Objekte
- Objekte: Zustand und Prozeduren (≈ Modul)
- Berechnung: Nachrichtenaustausch zwischen Objekten (≈ Prozeduraufruf)

Üblicherweise werden bei der Programmierung in einer Programmiersprache mehrere Paradigmen kombiniert, beispielsweise die imperative und objektorientierte Programmierung bei Java, oder die imperative, objektorientierte und funktionale Programmierung bei Scala.

1.4 Eigenschaften von Programmiersprachen

Wenn man Programmiersprachen verstehen oder beurteilen will, muss man sich deren Details genauer anschauen.

1.4.1 Elemente von Programmiersprachen

Jede Programmiersprache beinhaltet drei wichtige Elemente, die man beherrschen sollte, um diese zu verwenden.

Syntax: Dies ist eine Beschreibung (einer Obermenge) der zulässigen Zeichenfolgen, die ein Programm bilden. Die Syntax wird meistens durch reguläre Ausdrücke und kontextfreie Grammatiken beschrieben.

Semantik: Dies ist eine Beschreibung der Bedeutung der durch die Syntax beschriebenen Zeichenfolgen. Häufig unterscheidet man weiter:

statische Semantik: Dies beinhaltet die Eigenschaften von Programmen, die unabhängig von der Ausführung sind, z.B. Einschränkungen der Syntax:

- „Bezeichner müssen deklariert werden."
- „Bei einer Zuweisung müssen die linke und rechte Seite den gleichen Typ haben."

dynamische Semantik: Was passiert eigentlich bei der Programmausführung?

Pragmatik: Dies beinhaltet Anwendungsaspekte der Sprache, die außerhalb der Semantikbeschreibung liegen, z.B.:

- Wie werden Gleitpunktzahlen addiert?
- Wie soll man bestimmte Sprachelemente anwenden?
- Wie werden Programme analysiert (z.B. Debugging-Methoden)?

1.4.2 Aspekte von Programmiersprachen

Neben der Beschreibung einer Programmiersprache kann man diese auch unter verschiedenen Gesichtspunkten anschauen, die dann eventuell für eine beabsichtigte Anwendung unterschiedlich gewichtet werden können.

Sprachbeschreibung Die Beschreibung einer Programmiersprache sollte formal sein, damit diese dann eindeutig angewendet werden kann. Meistens ist aber nur die Syntax formal beschrieben, und die Semantik nur informell (umgangssprachlich), was zu Gefahren bei der Verwendung der Sprache führen kann.

Implementierbarkeit Die Programmiersprache sollte einfach zu implementieren sein, dies bedeutet:

- preiswerte Übersetzer (vgl. Ada!)
- schnelle Übersetzer
- geringere Fehleranfälligkeit von Übersetzern
- schnelle Zielprogramme

Anwendbarkeit

- keine ungewohnten Schreib- und Denkweisen
- problemorientierte Programmierung, geringe Rücksicht auf Implementierungsanforderungen (nicht Hardware-orientiert)
- leichte Integration mit „externer Welt“ (Betriebssystem, Grafik, Bibliotheken, andere Programme, ...)

Robustheit

- Inwieweit ist ein sicherer Programmablauf garantiert oder welche Arten von Laufzeitfehlern können auftreten?
  (Typprüfung, Speicherverwaltung, ...)
- Können Programme robust gemacht werden, damit ein Anwendungsprogramm nicht einfach abstürzt?
- Welche möglichen Fehler können behandelt werden?

Portabilität

- Laufen die Programme auf vielen Rechnern/Rechnertypen/Betriebssystemen?
- Vermeidung von Rechner/Betriebssystem-spezifischen Konstrukten (I/O, Parallelität)
- besser: Bibliotheken (implementiert auf verschiedenen Rechnern / Betriebssystemen)

Orthogonalität Dies bezeichnet hier die Unabhängigkeit verschiedener Sprachkonstrukte. Wünschenswert ist hierbei:

- wenig Konstrukte
- universell kombinierbar
  Beispiel: Wenn die Konzept von Typen und Funktionen orthogonal sind, dann sollte es keine Restriktionen bei den Typen von Parametern und Ergebnissen von Funktionen geben

- einfache (verständliche) Sprache
- einfache Implementierung
- häufig: spezielle Syntax für häufige Kombinationen (dies wird auch als „syntaktischer Zucker“ bezeichnet)

1.4.3 Bereitgestellte Konzepte

Im Vergleich zur „nackten“ Maschine, also der jeweiligen Maschinensprache, stellen Programmiersprachen in der Regel die folgenden Konzepte bereit:

Berechnungsmodell: Dies kann wie die zu Grunde liegende Maschine sein, aber auch sehr verschieden davon, wie funktionale, logische oder nebenläufige Sprachen zeigen.

Datentypen und Operationen: Problemorientierte Datentypen, z. B. Zahlen in unterschiedlichen Ausprägungen, Zeichen(-ketten), Felder, Verbünde, ...

Abstraktionsmöglichkeiten: Nicht nur bloße Aneinanderreihung von Grundoperationen (wie z. B. in Assembler), sondern Strukturierung dieser:

- Funktionen: Abstraktion komplexer Ausdrücke
- Prozeduren: Abstraktion von Algorithmen
- Datentypen: Abstraktion komplexer Strukturen
- Objekte: Abstraktion von Strukturen, Zuständen und Operationen

Programmprüfung: Möglichkeiten, bestimmte Eigenschaften des Programms zur Übersetzungszeit zu prüfen und zu garantieren, wie z. B. Initialisierung von Variablen, Typprüfung, Deadlockfreiheit, ...

Eine Anmerkung zur *Effizienz*: Dieser Aspekt steht nicht im Fokus von Hochsprachen, denn es ist klar, dass die Verwendung von Hochsprachen Kosten verursacht (Übersetzungszeit, Laufzeit, Speicherplatz etc.). Diese Kosten werden aber aufgewogen durch die schnellere Programmentwicklung, bessere Wartbarkeit, Zuverlässigkeit und Portabilität von Programmen, die in einer Hochsprache geschrieben werden. Hierzu ein Zitat von Dennis Ritchie, einem der maßgeblichen Entwickler von C und Unix:

> „Auch wenn Assembler schneller ist und Speicher spart, wir würden ihn nie wieder benutzen.“

1.5 Ziele und Inhalte der Vorlesung

- Vorstellung wichtiger Sprachkonzepte und Programmierparadigmen:
  - imperative Programmiersprachen
  - Konzepte zur Programmierung im Großen
  - funktionale Programmiersprachen

  - logische Programmiersprachen
  - nebenläufige und verteilte Programmierung
- multilinguales Programmieren
- einfaches Erlernen neuer Programmiersprachen
- kritische Beurteilung von Programmiersprachen hinsichtlich ihrer Eignung für bestimmte Anwendungen
- eigener Sprachentwurf („special purpose language“/„domain-specific languages“)