




Grundlagen der Programmierung 

Grundzüge der Informatik

2. Einführung in die Grundlagen der Programmierung


Prof. Dr. Mira Mezini
 AG Softwaretechnik
 TU Darmstadt



V2 - 2 Grundlagen der Programmierung 


Inhalt

- 2.2.1 Ziele der Programmierung
- 2.2.2 Algorithmen
- 2.2.3 Grundbegriffe der Programmierung
- 2.2.4 Compiler, Interpreter, Programmierumgebungen

V2 - 3 Grundlagen der Programmierung 

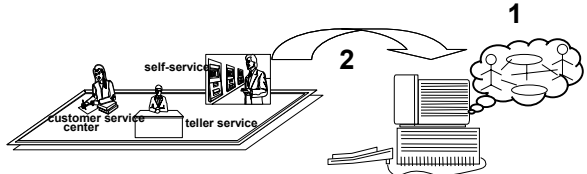
2.1 Ziele der Programmierung

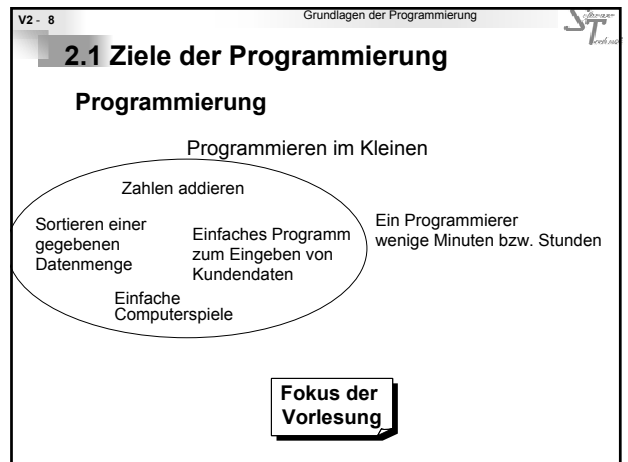
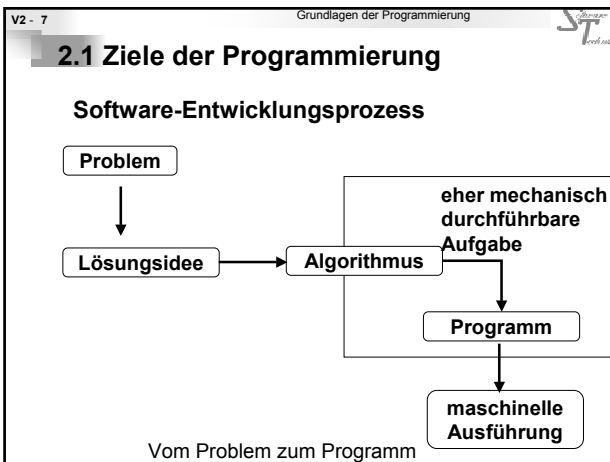
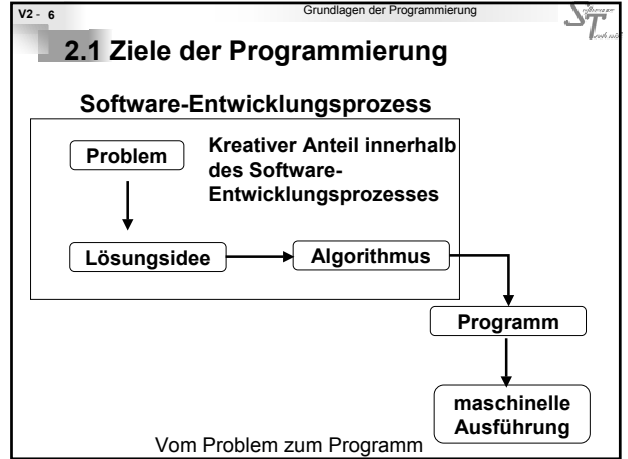
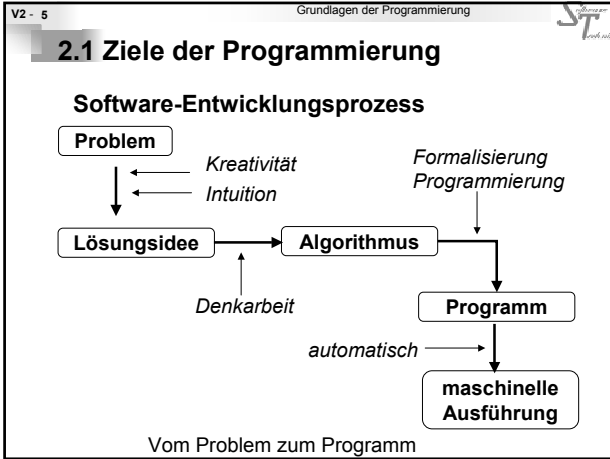
- **Ziel der Softwareentwicklung:** zu gegebenen Problemen auf dem Computer ausführbare Programme (Software) zu entwickeln, die
 - Probleme korrekt und vollständig lösen
 - möglichst effizient sind
 - wartbar, robust, flexibel
 - benutzerfreundlich (→ Mensch-Computer-Interaktion)

V2 - 4 Grundlagen der Programmierung 

2.1 Ziele der Programmierung

- **Programmierung** ist ein Teilgebiet der Informatik, der sich befasst:
 1. Methoden und Denkweisen bei der Problemlösung mit Computern
 2. mit dem Vorgang der Programmerstellung





2.1 Ziele der Programmierung

Programmierung

Programmieren im Grossen

Ordersystem für
Aktienhändler Komplexe
Computerspiele

Raketensteuerung

Teams
Monate, sogar Jahre

**Fokus der
SE Vorlesung im
Hauptstudium**

2.1 Ziele der Programmierung

Phasen der Softwareentwicklung (idealisiert)

- saubere, formale **Erfassung der Problemstellung**
 - Nebenbedingungen, Sonderfälle, etc. sind zu beachten
- **Analyse der Problemstellung**
 - Gibt es ähnliche, bereits gelöste Probleme?
 - Ist eine Aufteilung in Teilprobleme möglich?
- **Entwurf eines Lösungsverfahrens → Algorithmus**
- **Nachweis der Korrektheit**
- **Analyse des Aufwands bzgl. Rechenzeit und Speicherplatzverbrauch → Komplexitätsanalyse**
- **Implementierung**
- **Test und Fehlerbeseitigung**
- **Dokumentation**

2.1 Ziele der Programmierung

Phasen der Softwareentwicklung (praktisch)

„The difference between theory and practice is that in theory, there is no difference between theory and practice, but in practice there is.“

- Oft muss zu bereits abgeschlossenen Phasen zurückgekehrt werden
 - **z. B. Fehlerkorrektur oder modifizierte Anforderungen**
- eine schnell erstellte Vorabversion (Prototyp) wird zunächst implementiert, die dann schrittweise verbessert, bzw. ergänzt wird

2.1 Ziele der Programmierung

Phasen der Softwareentwicklung (praktisch)

- die einzelnen Phasen sind nicht sauber getrennt
- organisatorische Probleme beeinträchtigen die Entwicklung
- Es gibt nicht immer eine ideale oder auch nur gute Lösung
- Korrektheitsnachweis wird in der Regel nicht durchgeführt
- Dokumentation wird mangelhaft, wenn überhaupt erstellt

2.1 Ziele der Programmierung

Phasen der Softwareentwicklung (idealisiert)

die Organisation des Software-Entwicklungsprozesses ist das Thema der Software-Engineering Vorlesung

2.2 Algorithmen

Intuitiver Algorithmusbegriff

Ein Algorithmus ist eine eindeutige, endliche **Beschreibung** eines allgemeinen, endlichen **Verfahrens** zur **schrittweisen Ermittlung** gesuchter Größen aus gegebenen Größen

- Beschreibung unabhängig von einer konkreten Programmiersprache und einem konkreten Computer-System
- Programmiersprachen und Computer-Systeme sind nur Mittel zum Zweck → automatische Ausführung von Algorithmen

2.2 Algorithmen

Eine erste Beschreibung

- die Verwendung von Algorithmen ist nicht auf die Informatik beschränkt
- **jedes schematische Lösungsverfahren** ist ein Algorithmus
 - Bastelanleitungen
 - Kochrezept
 - Spielregeln
 - Verfahren für die Konvertierung einer Dezimalzahl ins Dualsystem

2.2 Algorithmen

Eine erste Beschreibung

- jede Arbeitsanleitung besteht aus einzelnen **Anweisungen**
 - Kochrezept
 - Trauben waschen,
 - gewaschene Trauben halbieren und entkernen,
 - ...

2.2 Algorithmen

Eine erste Beschreibung

- gewisse Anweisungen sind nur unter bestimmten Bedingungen auszuführen (**bedingte Anweisungen**)
 - Anleitungen für einen Fußballschiedsrichter
“Wenn das Foul im Strafraum der eigenen Mannschaft erfolgt, dann pfeife Strafstoß, ansonsten pfeife Freistoß.”

2.2 Algorithmen

Eine erste Beschreibung

- gewisse Anweisungen werden mehrmals hintereinander in einer **Wiederholungsanweisung** ausgeführt, solange eine bestimmte Bedingung erfüllt ist
 - Anleitung beim Mensch-Ärgere-Dich-Nicht:
“Solange ein Spieler eine 6 würfelt, darf er nochmal würfeln.”

2.2 Algorithmen

Eine erste Beschreibung

- zum Ausführen der Anleitungen müssen gewisse Voraussetzungen erfüllt sein
 - zum Kochen werden Zutaten benötigt
 - Basteln ist ohne Materialien nicht möglich

2.2 Algorithmen

Historisches

- Bezeichnung nach dem arabischen Mathematiker Al-Khwarismi
 - um 825 n. Chr. im heutigen Usbekistan, das damals Khwarizm hieß
 - entwickelte Verfahren für das Regeln von Erbverhältnissen reicher Araber, die 4 Frauen hatten, mit Hilfe algebraischer Methoden
 - Aus dem Namen wurde Algorithmus und daraus Algorithmus

2.2 Algorithmen

Historisches

- Älteste bekannte Algorithmen wurden von dem griechischen Mathematiker EUKLID (3./4. Jahrhundert v. Chr.) entwickelt
 - Konstruktionsverfahren für Dreiecke aus gegebenen Stücken
 - EUKLID'scher Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen

2.2 Algorithmen

Beschreibung von Algorithmen

- Die Beschreibung eines Algorithmus
 - erfolgt in einem Formalismus
 - dabei können andere Algorithmen und, letztlich, elementare Algorithmen benutzt werden

2.2 Algorithmen

Beschreibung von Algorithmen Beschreibungstechniken

- mathematisch
- textuell
 - Umgangssprache
 - Programmiersprache
 - Pseudocode
- visuell
 - Programmabläufe
 - Struktogramme

2.2 Algorithmen

Beschreibung von Algorithmen

- In mathematischer Form
 $f: \mathbb{N} \rightarrow \mathbb{N}, f(n) =$

$$\sum_{i=1}^n i \quad \text{für } n \text{ in } \mathbb{N}$$

2.2 Algorithmen

Beschreibung von Algorithmen

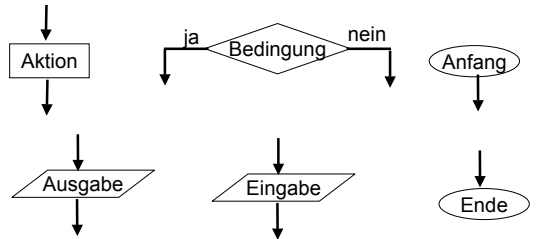
- Umgangssprachlich

Gegeben sei eine natürliche Zahl n .
 Addiere die natürlichen Zahlen von 1 bis n . Die Summe ist das Resultat.

2.2 Algorithmen

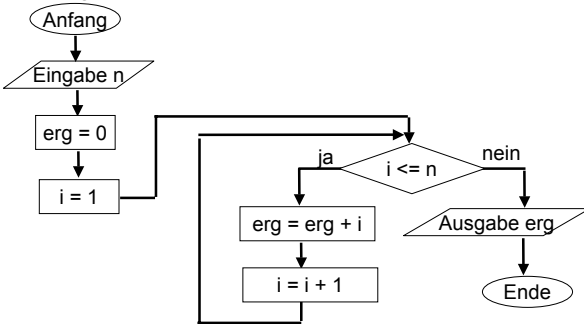
Beschreibung von Algorithmen

- Graphische Notationen: Programmabläufe (auch Flussdiagramme genannt)



2.2 Algorithmen

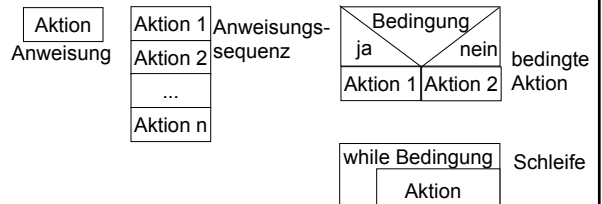
Beschreibung von Algorithmen Programmabläufe



2.2 Algorithmen

Beschreibung von Algorithmen

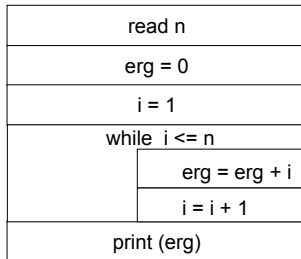
- Graphische Notationen: Struktogramme (Nassi-Shneiderman-Diagramme)
 - i.A. übersichtlicher und verständlicher als Programmablaufplänen



2.2 Algorithmen

Beschreibung von Algorithmen

- Graphische Notationen: Struktogramme



2.2 Algorithmen

Beschreibung von Algorithmen

- Programmiersprache / Pseudo-Code

```
int n = readInt();
int erg = 0;
int i = 1;
while ( i <= n) {
    erg = erg + i;
    i = i + 1;
}
printInt(erg);
```

2.2 Algorithmen

Beschreibung von Algorithmen

Verwendung anderer Algorithmen

- Ein Algorithmus verwendet in der Regel andere Algorithmen, um ein Verfahren zu beschreiben

Kochrezept

- Gemüse vorbereiten
- Fleisch ...
- ...

Gemüse vorbereiten

- Tomaten waschen,
- gewaschene Tomaten halbieren und entkernen
- ...

2.2 Algorithmen

Beschreibung von Algorithmen

Verwendung elementarer Algorithmen

- alle Algorithmen basieren letztendlich auf vordefinierten elementaren Algorithmen, die der ausführende Prozessor beherrscht
- in Programmiersprachen sind die elementaren Algorithmen definiert, z.B. durch die Festlegung von mathematischen und logischen Operationen

2.2 Algorithmen

Eigenschaften von Algorithmen

Eindeutige Beschreibung:

- aus der Formulierung des Algorithmus muss die Abfolge der einzelnen Verarbeitungsschritte eindeutig hervorgehen
- es dürfen keine widersprüchlichen Aussagen enthalten sein

2.2 Algorithmen

Eigenschaften von Algorithmen

Eindeutige Beschreibung:

- Wahlmöglichkeiten, wie Auswahl und Wiederholung, sind zugelassen
 - Es muss aber genau festliegen, wie die Auswahl einer der Möglichkeiten erfolgen soll

2.2 Algorithmen

Eigenschaften von Algorithmen

Eindeutige Beschreibung:

Beispiel eines mehrdeutigen Algorithmus:

1. Gegeben seien beliebige Zahlen M und N
2. Berechne $\text{ggT}(M, N)$
3. Das Ergebnis soll entweder auf dem Bildschirm ausgegeben oder in der Variable r gespeichert werden

Offene Fragen:

- was heißt „berechnen“?
- was soll mit dem Ergebnis nun geschehen?

2.2 Algorithmen

Eigenschaften von Algorithmen

- Endliche Beschreibung: statische Endlichkeit
 - Die Beschreibung des Algorithmus besitzt eine endliche Länge
 - Beispiel: Mathematik

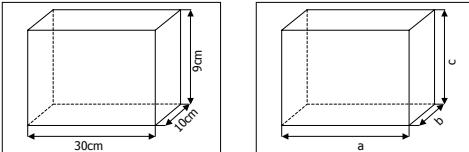
$$1 + 2 + 3 + \dots$$

2.2 Algorithmen

Eigenschaften von Algorithmen

■ Allgemeine Problemlösung

- Ein Algorithmus löst ein allgemeines Problem, bzw. Problemklasse und nicht nur ein spezielles Problem
- Die Wahl eines aktuell zu lösenden Problems aus dieser Klasse erfolgt durch Parameter
- Ziel: **Wiederverwendbarkeit**



2.2 Algorithmen

Eigenschaften von Algorithmen

■ Ein Algorithmus muss ausführbar sein

→ ein Prozessor, der den Formalismus kennt und die elementaren Algorithmen beherrscht, muss ihn abarbeiten können

→ Beispiel einer nicht ausführbaren Vorschrift:

»Wenn man in 14 Tagen Zahnschmerzen bekommt, dann ist bereits heute mit dem Zahnarzt ein Termin auszumachen, um die Wartezeit zu verkürzen.«

2.2 Algorithmen

Eigenschaften von Algorithmen

■ Endliches Verfahren: dynamische Endlichkeit

- die Ausführung eines Algorithmus muss in der Regel für jede Eingabe nach endlich vielen Abarbeitungsschritten ein Ergebnis liefern und anhalten, d.h. terminieren
- **Ausnahme**: die Endlichkeitsforderung wird fallengelassen
 - z.B. Steuerungsalgorithmen wie das Betriebssystem oder eine Ampelsteuerung

2.2 Algorithmen

Eigenschaften von Algorithmen

■ Endliches Verfahren: dynamische Endlichkeit

- Es gibt Aufgaben, die zwar lösbar (berechenbar) sind, für deren Lösung jedoch kein Algorithmus angegeben werden kann, der terminiert
- Es gibt keinen Algorithmus, der für alle Algorithmen feststellt, ob sie terminieren oder nicht
 - Halteproblem

2.2 Algorithmen

- Probleme durch Algorithmen lösen ist eine zentrale Aufgabe der Informatik
- ist ein Lösungsalgorithmus entdeckt und formuliert, dann ist das Problem aus Sicht der Theorie erledigt
- Für die Praktische Informatik stellen sich aber noch zusätzliche Fragen ...

2.2 Algorithmen

Zusätzliche Fragestellungen ...

- Unter welchen Voraussetzungen arbeitet der Algorithmus?
- Welche Eingaben sind erlaubt und wie sehen die möglichen Ausgaben bei zulässigen Eingaben aus?
- Endet der Algorithmus für alle zulässigen Eingaben, und wie beweist man das?
- Löst der Algorithmus das gewünschte Problem richtig? Wie kann es sichergestellt werden, dass für alle Eingaben die gewünschten Ergebnisse erzielt werden?
- Wie hoch ist der Aufwand des Algorithmus: Es lässt sich beweisen, dass es zu jedem Algorithmus unendlich viele verschiedene, äquivalente Algorithmen gibt, die die gleiche Aufgabe lösen.

2.3 Basis Begriffe der Programmierung

Das Konzept des gespeicherten Programms

- In den ersten Jahren der Informatik war das auszuführende Programm meist ein fester Bestandteil des Steuerwerks
- **UNFLEXIBILITÄT !**
- Analog zu einem Musikkasten, der immer die gleiche Melodie spielt
 - Was man brauchte, war die Flexibilität eines CD-Wechslers

2.3 Basis Begriffe der Programmierung

Das Konzept des gespeicherten Programms

- ein erster Ansatz:
 - Leicht wieder-verdrahtbare Steuereinheiten
 - ähnlich wie eine alte Telefonzentrale
- Durchbruch kam mit der Idee, dass *ein Programm, genau so wie Daten im Speicher codiert und gespeichert werden kann*



2.3 Basis Begriffe der Programmierung

Das Konzept des gespeicherten Programms Anweisungen als Bit-Mustern

- CPU ist in der Lage: (a) Anweisungen des Programms aus dem Speicher zu extrahieren, (b) sie zu decodieren und auszuführen
- Decodieren → CPU ist in der Lage, bestimmte Bit-Muster als eine Darstellung von bestimmten Anweisungen zu erkennen



2.3 Basis Begriffe der Programmierung

Das Konzept des gespeicherten Programms

- Die Menge solcher Anweisungen zusammen mit dem Kodierungssystem bilden die **Maschinensprache**
- Legt die Mittel fest, die uns zur Verfügung stehen, um der Maschine Algorithmen zu kommunizieren



2.3 Basis Begriffe der Programmierung

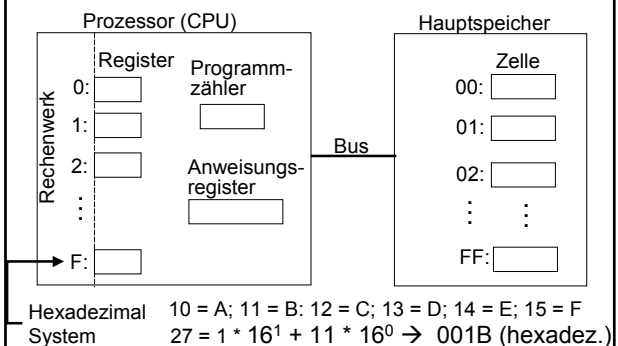
Maschinensprachen

- Sprachen der **ersten Generation**
- Direkte Programmierung in der Hardware → **Maschinenprogramm**
- Formulierung eines Programms durch eine Folge elementarer, in Binärcode dargestellten Befehlen aus dem Befehlsatz eines bestimmten Mikroprozessors
- direkt auf der Hardware ausführbar
- Maschinensprachen werden auch



2.3 Basis Begriffe der Programmierung

Eine typische Maschinensprache



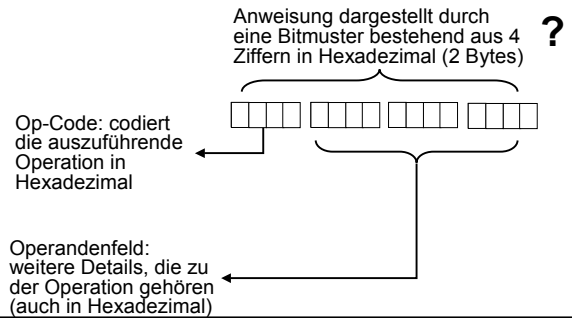
2.3 Basis Begriffe der Programmierung

Eine typische Maschinensprache Das Hexadezimal-System

- Zahlensystem mit 16 als Basis
- Für die Zahlendarstellung braucht man 15 Ziffern (0 – 15)
- Um alle Ziffern mit einem einzigen Zeichen darstellen zu können, werden die Ziffern 10 – 15 jeweils durch Buchstaben A – F codiert
- Für die Umrechnung einer dezimalen Zahl nach Hexadezimal kann man das gleiche Verfahren anwenden, das man für die Umrechnung nach dem Dualsystem benutzt.

2.3 Basis Begriffe der Programmierung

Eine typische Maschinensprache Das Format einer Maschinenanweisung



2.3 Basis Begriffe der Programmierung

Eine typische Maschinensprache Das Format einer Maschinenanweisung

„Lade den Inhalt der Zelle No. 108 im Register No. 5“

0001 0101 0110 1100

Lade

6 12 = C

Register No. 5

$$5 = 4 + 1$$

$$= 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

Zelle No. 6C

$$6C =$$

$$6 * 16 + 12 = 108$$

2.3 Basis Begriffe der Programmierung

Eine typische Maschinensprache

Algorithmus umgangssprachlich

1. Lade Register No. 5 mit dem Inhalt der Zelle No. 108
2. Lade Register No. 6 mit dem Inhalt der Zelle No. 109
3. Wenn der zweite Wert 0 ist, springe zu 6
4. Dividiere den Inhalt des ersten Registers mit dem des zweiten Registers und speichere das Resultat im Register 0
5. Speichere den Inhalt des Registers 0 in der Zelle 110
6. HALT



1. 156C
2. 166D
3. 5056
4. 306E
5. C000

Algorithmus in Maschinensprache

2.3 Basis Begriffe der Programmierung

Assemblersprachen

- Sprachen der zweiten Generation
- Niedere, maschinenorientierte Sprachen
- Programmierung der Hardware unter Verwendung *symbolischer Namen* für Maschinenbefehle und Adressen
- **Assembler**
 - Programm, das ein Assembler-Programm in ein Maschinenprogramm umsetzt

2.3 Basis Begriffe der Programmierung

Assemblersprachen

Mnemonics für die Darstellung der Anweisungen

```
1. 156C
2. 166D
3. 5056
4. 306E
5. C000
```

Algorithmus in
Maschinsprache

Benennung der Registern
und Speicherzellen
→ Identifier

```
1. LD R5 PRICE
2. LD R6 TAX
3. ADDI R0, R5 R6
4. ST R0 TOTAL
5.HLT
```

Algorithmus in
Assemblersprache

Für die damalige Zeit ein gigantischer Schritt !

2.3 Basis Begriffe der Programmierung

Maschinsprachen und Assemblersprachen

- **Vorteil:**
 - ermöglichen die Erstellung sehr effizienter Programme

2.3 Basis Begriffe der Programmierung

Maschinsprachen und Assemblersprachen

Nachteile

- Hohe Abhängigkeit des Programms von dem speziellen Maschinentyp
 - Beim Portieren auf eine andere Maschine muss das Programm neugeschrieben werden, konform zu der Registerkonfiguration und Anweisungsmenge der neuen Maschine
- Programme sind für den Anwender, aber auch für den Entwickler, nur sehr schwer verständlich

2.3 Basis Begriffe der Programmierung

Maschinensprachen und Assemblersprachen

Nachteile

- Der Entwickler wird gezwungen im Sinne von kleinen Schritten der Maschinensprache zu denken
- Analogie:** Entwerfen eines Hauses im Sinne von Nagel, Ziegeln, usw. !

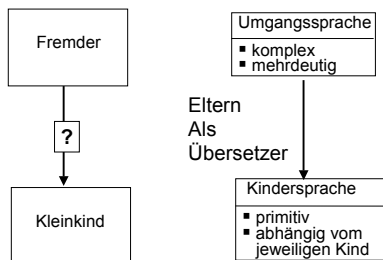
2.3 Basis Begriffe der Programmierung

Höhere Programmiersprachen

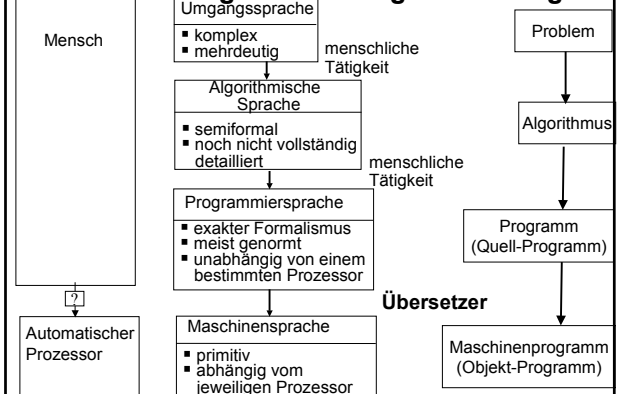
- Sprachen der dritten Generation
- Hardware-Unabhängigkeit**
- Orientierung an den zu bearbeitenden Problemfeldern
- für Menschen verständlicher und einfacher zu handhaben
- Abbildung in eine Maschinensprache
 - Verwendung eines **Übersetzters (Compiler)**
 - direkte Ausführung unter Anwendung eines **Interpreters**

2.3 Basis Begriffe der Programmierung

Kommunikation Mensch - Maschine über eine höhere Programmiersprache



2.3 Basis Begriffe der Programmierung



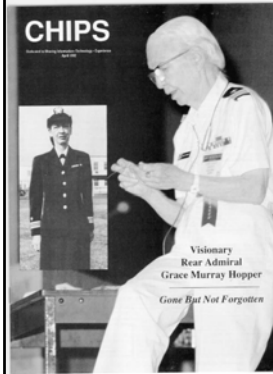
2.4 Compiler, Interpreter, Umgebungen



Admiral Grace Murray Hopper
(1906 – 1992)

She did this - the invention of the compiler -, she said, because she was lazy and hoped that "the programmer may return to being a mathematician."

2.4 Compiler, Interpreter, Umgebungen



Admiral Grace Murray Hopper
(1906 – 1992)

Flow-Matic

COBOL

„It really came about because I couldn't balance my checkbook“

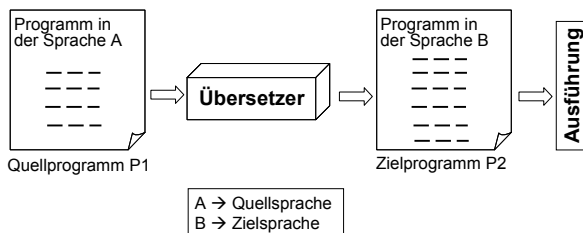
BUG, Debug

„It's easier to ask forgiveness than it is to get permission“

2.4 Compiler, Interpreter, Umgebungen

Definition eines Übersetzters

- Übersetzer (Compiler) ist ein **Programm**, das Programme aus einer **Programmiersprache A** in eine **Programmiersprache B** übersetzt



2.4 Compiler, Interpreter, Umgebungen

Definition eines Übersetzters

- **Semantische Korrektheit**

- Für die Abbildung

Quellsprache A → Zielsprache B

muss folgende Bedingung erfüllt sein:

- **Jedem Quellprogramm P1 in A ist genau ein Zielprogramm P2 in B zugeordnet**
- **Das dem Quellprogramm P1 zugeordnete Zielprogramm P2 muss die gleiche Bedeutung (Semantik) wie P1 besitzen**

2.4 Compiler, Interpreter, Umgebungen

Elementen einer Programmiersprache

- **eindeutige Lexikalik** → Festlegung der gültigen Zeichen bzw. Wörter, aus denen Programme der Programmiersprache zusammengesetzt sein dürfen
- **eindeutige Syntax** → legt fest, welche Zeichenreihen bzw. Folgen von Wörtern korrekt formulierte („syntaktisch korrekte“) Programme der Sprache darstellen und welche nicht

2.4 Compiler, Interpreter, Umgebungen

Elementen einer Programmiersprache

- **eindeutige Semantik** → Festlegung, welche Auswirkung die Ausführung des Programms auf dem Rechner hat
- **Pragmatik** → definiert den Einsatzbereich einer Sprache

2.4 Compiler, Interpreter, Umgebungen

Beschreibung einer Programmiersprache

- Syntax (beinhaltet häufig auch die Lexikalik)
 - Beschreibungsformalisten
 - kurz hier erwähnt, mehr dazu später
 - viel detaillierter jedoch in INF 4
- Semantik
 - (mathematische) Beschreibungsformalisten
 - nicht Thema in dieser Vorlesung → INF 4
 - wird jedoch i.d.R. umgangssprachlich beschrieben
- Pragmatik: umgangssprachlich

2.4 Compiler, Interpreter, Umgebungen

Übersetzungsphasen

Lexikalische Analyse

Quellprogramm wird in eine Folge von Terminalsymbolen (Token, Worte) zerlegt

Syntaktische Analyse

Testet, ob das Quellprogramm den Syntaxregeln der Quellsprache entspricht. Strukturiert Terminalsymbole in gültige Sätze.

Semantische Analyse

Testet, ob alle in Quellprogramm benutzten Namen deklariert wurden, und ihrem Typ entsprechend verwendet werden, usw.

Code Generierung

Zielprogramm wird erzeugt

2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen

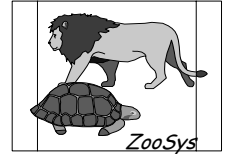
- Zur Überprüfung der syntaktischen Korrektheit eines Programms, muss zuvor die Syntax der Programmiersprache formal beschrieben werden.
- Zwei gängige Beschreibungsformalismen für eine Syntax
 - Syntaxdiagramme
 - BNF

2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: Syntaxdiagramme

Am Zoo Beispiel

- Im Zoo gibt es eine Menge von Gehegen mit verschiedenen Tieren
- Besucher können Gehege auf Wegen erreichen
- Es gibt Kreuzungen, an denen mehrere Wege eingeschlagen werden können

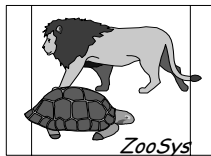


2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: Syntaxdiagramme

Am Zoo Beispiel

- Ein Fotograf möchte im Zoo eine Photoserie erstellen
- Er schießt jeweils ein Photo, immer wenn er an einem Gehege vorbei kommt

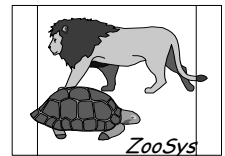


2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: Syntaxdiagramme

Am Zoo Beispiel

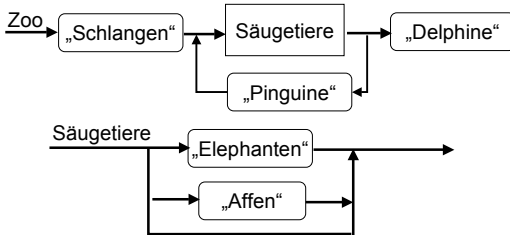
- Gegeben beliebige Photo-Sequenzen, soll später entschieden werden, ob sie gültig sind
- Eine Sequenz ist gültig, wenn sie der Regel entspricht: „*Photograph läuft durch Wege und schießt ein Photo bei jedem Gehege*“



2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: Syntaxdiagramme Am Zoo Beispiel

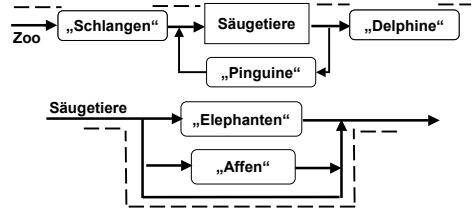
- Mögliche Wege durch den Zoo werden mittels folgende Diagramme veranschaulicht



2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: Syntaxdiagramme

- anhand der Diagramme Zoo Beispiel kann man entscheiden:
 - „Schlangen“ → „Delphine“ ist gültig
 - „Schlangen“ → „Pinguine“ ist ungültig



2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: Syntaxdiagramme

- Graphische Notation der gültigen Sätze in einer Sprache
- Diagramme, die aus runden und eckigen Kästchen sowie Pfeile bestehen
- Worte (Terminalsymbol, Token) werden in runden Kästchen dargestellt
- Untergeordnete Diagramme werden durch eckige Kästchen dargestellt
- Aus jedem Kästchen führt genau ein Pfeil hinaus und genau einer hinein
- Pfeile dürfen sich aufsplitten und zusammengezogen werden
- Jeder Diagramm hat einen Namen

2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: BNF

- Backus-Naur-Form (BNF):** textuelle Beschreibungsform
- Produktion (BNF Regel):** das Äquivalent zu einem einzelnen Syntaxdiagramm
 - Linke Seite:** einzelne Nicht-Terminalsymbole (in spitze Klammern)
 - Rechte Seite:** Folgen von
 - Terminalsymbolen, ϵ (epsilon) leerer Weg in einem Syntaxdiagramm
 - Nicht-Terminalsymbolen in Hochkommata
 - dem Meta-Symbol $|$ → bedeutet „oder“
- Getrennt durch $::=$ Aufsplittender Pfeil in einem Syntaxdiagramm

2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: BNF

Beispiele

- Nicht-terminales Symbol

- BNF

<A>

- Syntaxdiagramm



- Terminales Symbol

- BNF

class

- Syntaxdiagramm



2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: BNF

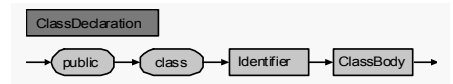
Definition einer Syntaxregel

- BNF

<ClassDeclaration> ::=

public class <Identifier> <ClassBody>

- Syntaxdiagramm



2.4 Compiler, Interpreter, Umgebungen

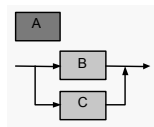
Syntaxdarstellungen: BNF

Darstellung einer Alternative

- BNF

<A> ::= | <C>

- Syntaxdiagramm



2.4 Compiler, Interpreter, Umgebungen

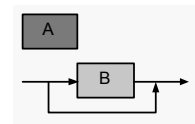
Syntaxdarstellungen: BNF

Darstellung einer Option

- BNF

<A> ::= [B]

- Syntaxdiagramm



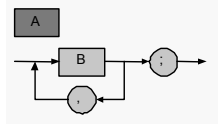
2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: BNF

Darstellung einer Wiederholung

- BNF $\langle A \rangle ::= \langle B \rangle \{ , \langle B \rangle \}$

- Syntaxdiagramm

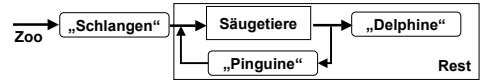


2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: BNF

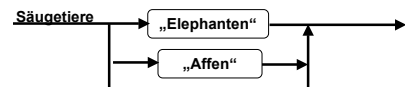
Zoo Beispiel

$\langle \text{Zoo} \rangle ::= \text{„Schlangen“ } \langle \text{Rest} \rangle$



$\langle \text{Rest} \rangle ::= \langle \text{Säugetiere} \rangle \text{ „Pinguine“ } \langle \text{Rest} \rangle \mid \langle \text{Säugetiere} \rangle \text{ „Delphine“}$

$\langle \text{Säugetiere} \rangle ::= \text{„Elephanten“ } \mid \text{„Affen“ } \mid \epsilon$



2.4 Compiler, Interpreter, Umgebungen

Syntaxdarstellungen: BNF

- BNF ist in Vergleich zu Syntaxdiagrammen
 - kompakter
 - schwieriger zu lesen.

2.4 Compiler, Interpreter, Umgebungen

Klassifikation von Übersetzern

- Klassifikation anhand der Ausdrucksmächtigkeit (**Anzahl der Konstrukte**) der Quell- und Zielsprache
 - Übersetzer (Compiler)
 - Umkehrübersetzer (Decompiler)
 - 1-1-Übersetzer (Präprozessor)

2.4 Compiler, Interpreter, Umgebungen

Klassifikation von Übersetzern

- **Übersetzer (Compiler): Quellsprache ausdrucksstärker als Zielsprache**
 - Beispiel
 - die Quellsprache ist eine höhere Programmiersprache
 - die Zielsprache ist eine Assembler-ähnliche Sprache
 - Schleifen und bedingte Ausdrücke müssen unter Verwendung von Sprungbefehlen übersetzt werden

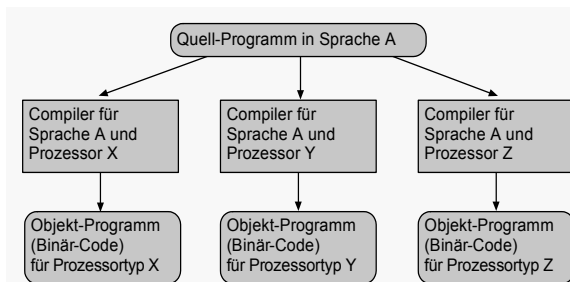
2.4 Compiler, Interpreter, Umgebungen

Klassifikation von Übersetzern

- **Umkehrübersetzer (Decompiler): Zielsprache ausdrucksstärker als Quellsprache**
 - **Beispiel:** Übersetzung eines Maschinenprogramms in ein Assemblerprogramm
- **1-1-Übersetzer (Präprozessor): Quellsprache und Zielsprache ungefähr gleichausdrucksstark**
 - **Beispiel:** textuelle Ersetzung von Macros

2.4 Compiler, Interpreter, Umgebungen

Traditionelle Übersetzung



2.4 Compiler, Interpreter, Umgebungen

Traditionelle Übersetzung

Vorteile

- Optimale Ausnutzung der jeweiligen Prozesseigenschaften
- **hohe Abarbeitungsgeschwindigkeit** der übersetzten Programme

2.4 Compiler, Interpreter, Umgebungen

Traditionelle Übersetzung

Vorteile

- Ein Programm, das in einer höheren Programmiersprache geschrieben ist, kann **theoretisch** nach der Anwendung der entsprechenden Übersetzer auf jeder Maschine laufen.
- ... Wirklich?

„The difference between theory and practice is that in theory, there is no difference between theory and practice, but in practice there is.“

2.4 Compiler, Interpreter, Umgebungen

Traditionelle Übersetzung

Nachteile

- Das übersetzte Programm läuft nur auf dem jeweiligen Prozessortyp
 - Für jeden Prozessortyp muss das Programm mit einem anderen Compiler neu übersetzt werden

2.4 Compiler, Interpreter, Umgebungen

Traditionelle Übersetzung

Nachteile

- Programmiersprachen sind oft **plattformabhängig** definiert
 - Bestimmte Eigenschaften der Maschine beeinträchtigen den Compiler-Entwurf
 - Größe der Register oder Speicherzellen beeinträchtigt die maximale Länge der Zahlen, die manipuliert werden können
 - Das Resultat: es gibt unterschiedliche **Dialekte einer Sprache**

2.4 Compiler, Interpreter, Umgebungen

Klassifikation der höheren Sprachen

- Es existieren mehr als 200 problemorientierte Sprachen für die verschiedensten Anwendungsgebiete
- Keine ideale Sprache, die für alle Gebiete optimale Sprachkonstrukte besitzt

2.4 Compiler, Interpreter, Umgebungen

Klassifikation der höheren Sprachen

▪ Fortran	1954-1957
▪ Algol 60	1958-1960
▪ Cobol	1959-1960
▪ Lisp	1959-1962
▪ Basic	1963-1965
▪ PL/1	1964-1967
▪ Simula 67	1965-1967
▪ Pascal	1971
▪ C	1974
▪ Modula-2	1976

Überblick über Sprachen

2.4 Compiler, Interpreter, Umgebungen

Klassifikation der höheren Sprachen

▪ Prolog	1977
▪ Ada	1979
▪ SQL	1970-1980
▪ Smalltalk-80	1970-1980
▪ C++	1980-1983
▪ Eiffel	1986-1988
▪ Oberon	1988
▪ Beta	1993
▪ Java	1990-1995.

Überblick über Sprachen

2.4 Compiler, Interpreter, Umgebungen

Klassifikation der höheren Sprachen

- Sprachunterschiede
 - Anwendungsgebiete
 - Mathematisch-technische Probleme
 - Kaufmännische Probleme
 - Linguistische Probleme
 - Steuerung zeitabhängiger Vorgänge, z.B. bei Ampeln
 - Speziell vs. allgemein (*general purpose*)
 - Je mehr Möglichkeiten eine Sprache dem Benutzer bietet, umso aufwendiger wird der Übersetzer.

2.4 Compiler, Interpreter, Umgebungen

Klassifikation der höheren Sprachen

- Sprachunterschiede
 - Konzepte
 - Prozedurale (imperative) Sprachen
 - Objektorientierte Sprachen
 - Funktionale und applikative Sprachen
 - Prädikative, logik-orientierte Sprachen
 - Deklarative Sprachen bzw. Sprachen der 4. Generation
 - Mehr darüber später !

2.4 Compiler, Interpreter, Umgebungen

Interpreter

- ein Interpreter ist ein **Programm, das ein Programm** einer bestimmten Programmiersprache **direkt ausführt**
- **Arbeitsweise:**
 - syntaktische Analyse der (nächsten) Anweisung des Quellprogramms
 - Übersetzung dieser Anweisung in eine Befehlsfolge der Maschinensprache
 - Ausführung der Befehlsfolge

2.4 Compiler, Interpreter, Umgebungen

Interpreter: Vorteile

- geänderte Anweisungen / Deklarationen des Quellprogramms sind sofort ausführbar
 - Neuübersetzung des kompletten Quellprogramms nicht notwendig
- Unterstützung der Testphase des Programms
 - schnelle Änderbarkeit
 - es lassen sich relativ schnell lauffähige Programmversionen erstellen → **Prototyping**

2.4 Compiler, Interpreter, Umgebungen

Interpreter: Nachteile

- Die Ausführungszeit eines Programms ist im Vergleich zur übersetzten Variante deutlich länger

2.4 Compiler, Interpreter, Umgebungen

Interpreter: Nachteile

Gründe für die längere Ausführungszeit

- Werden Anweisungen des Quellprogramms k-mal verwendet (z.B. bei Schleifen), werden sie k-mal analysiert und übersetzt

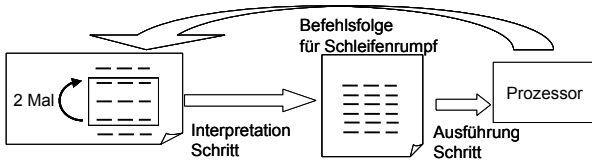
→ Verarbeitung des Quellprogramms folgt dem Ablauf der Ausführung des Quellprogramms

2.4 Compiler, Interpreter, Umgebungen

Interpreter: Nachteile

Gründe für die längere Ausführungszeit

Anweisungen des Quellprogramms, die k-mal verwendet werden, werden k-mal analysiert und übersetzt



2.4 Compiler, Interpreter, Umgebungen

Interpreter: Nachteile

Gründe für die längere Ausführungszeit

- Bei Variablen-Zugriffen müssen die zugeordneten Adressen immer wieder bestimmt werden
 - Während der Verarbeitung einer Deklaration erweitert das Interpreter-Programm eine von ihm angelegte Abbildung
Variablenname → Speicheradresse
 - Diese Abbildung muss bei jedem Variablenzugriff mit dem Variablennamen als Suchbegriff durchsucht werden

2.4 Compiler, Interpreter, Umgebungen

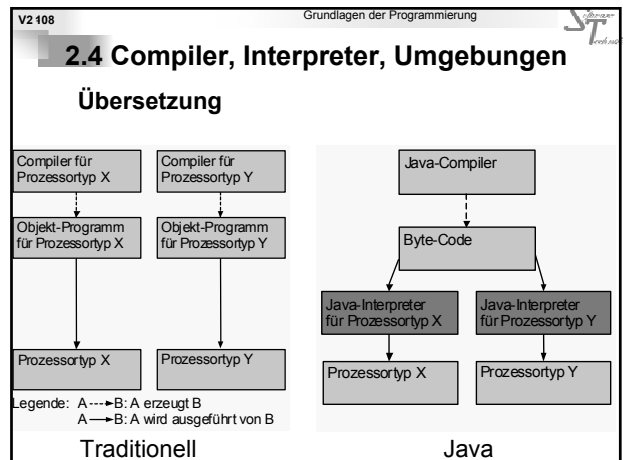
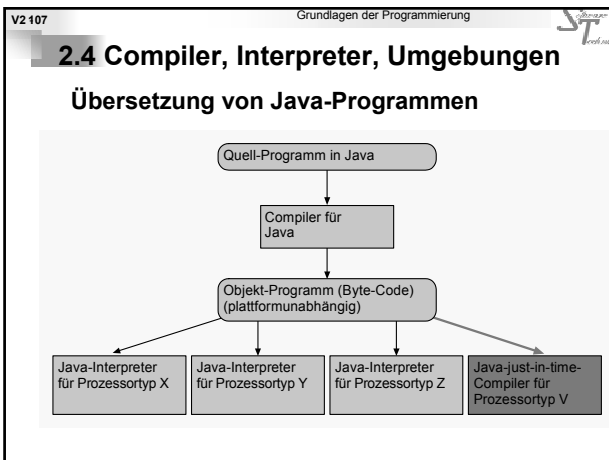
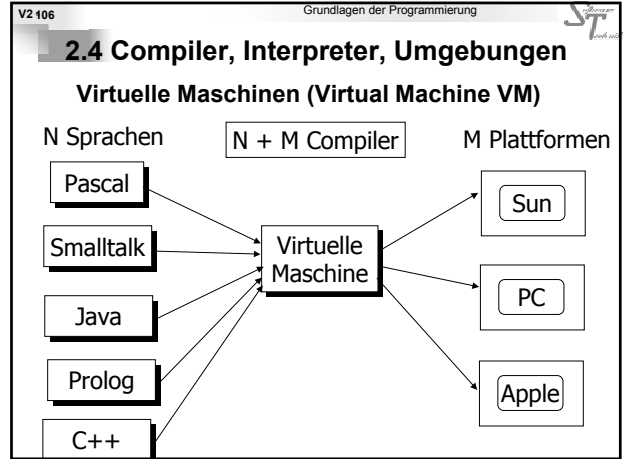
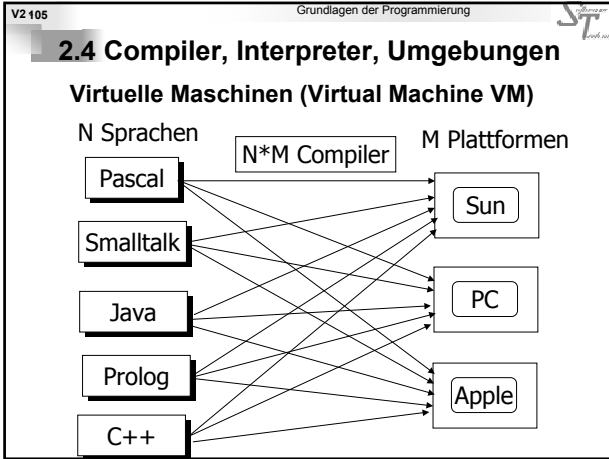
Virtuelle Maschinen (Virtual Machine VM)

- Eine Virtuelle Maschine ist in Programm, das die Arbeit eines Prozessors in Software simuliert
- Ein VM verdeckt die speziellen Eigenschaften des jeweiligen Prozessortyps
 - **Abstraktionsschicht !**

2.4 Compiler, Interpreter, Umgebungen

Virtuelle Maschinen (Virtual Machine VM)

- Programme einer höheren Sprache werden in eine Assembler-ähnlichen Zwischensprache übersetzt
- Die Anweisungen der Zwischensprache nennt man auch **Byte-Code**
- Die Zwischensprache wird von der VM interpretiert





2.4 Compiler, Interpreter, Umgebungen

Vorteile der VM Technologie

- Für alle Prozessortypen wird nur ein Java-Compiler benötigt
- Java ist unabhängig von allen Plattformen exakt definiert (Praxis?)
- Übersetzte Java-Programme laufen auf allen Prozessortypen, für die es einen Java-Interpreter gibt



2.4 Compiler, Interpreter, Umgebungen

Nachteile der VM Technologie

- Byte-Code Programme sind langsamer als Maschinenprogramme
 - *just-in-time-compiler*
 - Übersetzen den *Byte-Code* in ein Objekt-Programm für einen speziellen Prozessortyp
- Für jeden Prozessortyp wird ein Java-Interpreter benötigt.



2.4 Compiler, Interpreter, Umgebungen

Programmierumgebungen

- Erstellen von Java-Programmen
 - Texteditor / Java-Compiler / Java-Interpreter / Debugger ...
- Programmierumgebungen
 - Komfortabler als Einzelkomponenten
 - Stellen zusätzliche Funktionen zur Verfügung
 - Teilweise für mehrere Sprachen vom gleichen Hersteller erhältlich

