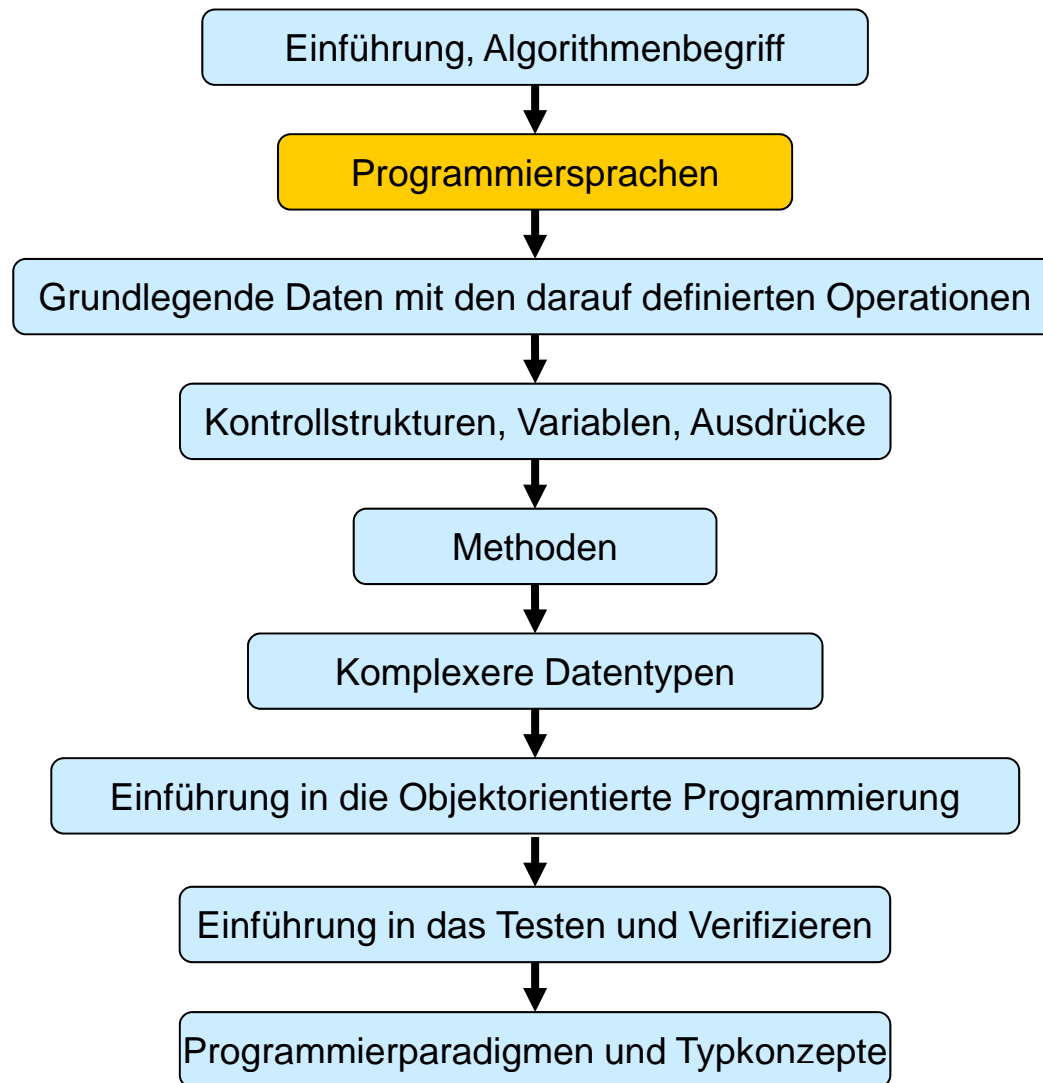
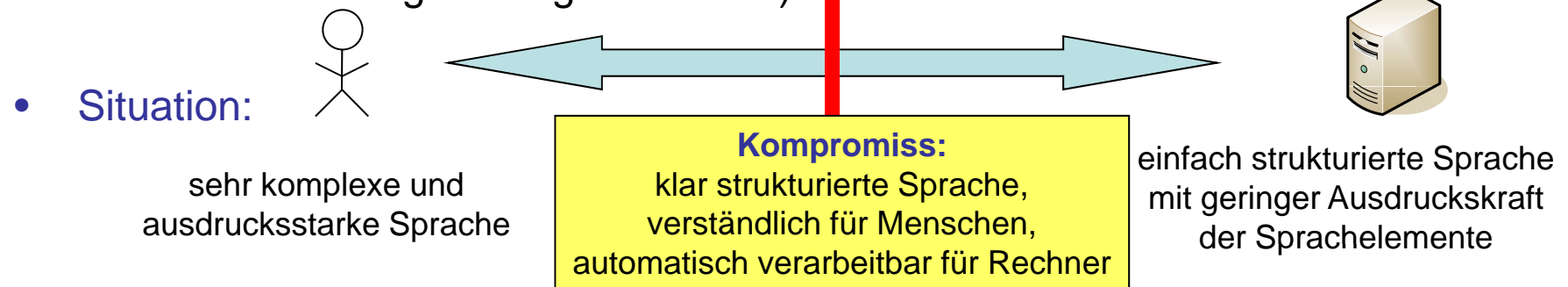


Inhalt dieser Veranstaltung



Wieso Programmiersprachen?

- Menschliche Sprachen haben **einige Nachteile** für unseren Zweck
 - Sie sind **mehrdeutig**
Beispiel: "Ein Junggeselle ist ein Mann, dem zum Glück noch eine Frau fehlt."
 - Sie sind **redundant**
 - Der Sprachumfang ist **sehr groß** (mehrere 100.000 Vokabeln)
 - Die Grammatik ist **sehr komplex**.
- **Maschinensprachen für Prozessoren in Rechnern werden in eindeutig codierten Bitfolgen ausgedrückt** und der Sprachumfang und die -komplexität ist sehr beschränkt (wenige hundert Befehle mit sehr einfachem und regelmäßigem Aufbau)



Aufgabe einer (höheren) Programmiersprache

- Steuerung der Abfolge
 - Für einen Algorithmus, wie wir ihn kennen gelernt haben, muss die exakte Reihenfolge des Programmablaufs festgelegt werden
 - Begriff in diesem Zusammenhang: **Anweisung**
 - Später dazu kommend: Parallelität mit mehreren parallel zueinander ausgeführten Abfolgen
- Angabe und Manipulation von Werten
 - Bereitstellung von **Basiswertemengen** mit darauf definierten Operationen (Zahlen, Buchstaben,...)
 - Möglichkeit der Definition und Manipulation **eigener aufgabenrelevanter Wertemengen** mit darauf definierten Operationen (MP3, Konto, ...)
 - Begriffe in diesem Zusammenhang: **primitiver Datentyp, Klasse, Ausdruck**

Klassen von Programmiersprachen

- **Maschinenorientierte Programmiersprachen** bieten im Wesentlichen die Instruktionen des jeweiligen Prozessors an.
Nachteil: Programme sind nicht portabel und schlecht lesbar
- **Problemorientierte Programmiersprachen** sind maßgeschneidert für eine bestimmte Problemklasse.
Nachteil: aber auch nur dafür
- **Universelle Programmiersprachen** bieten alle gängigen Bausteine zur Programmierung an

Beispiel: $skalarprodukt(x, y) = \sum_{i=1}^n x_i \cdot y_i$

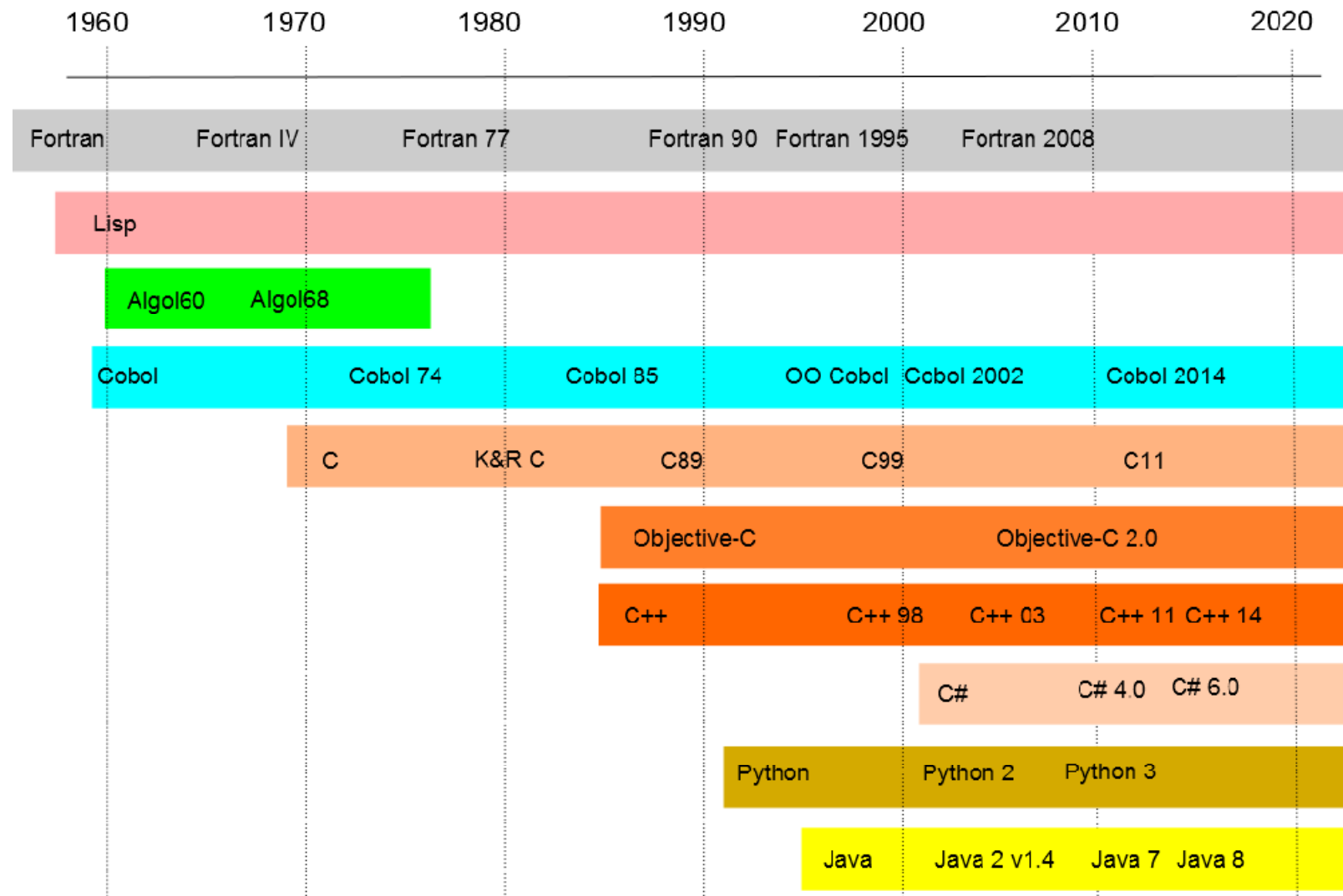
```
.text
    .p2align 4,,15
.globl skalarProdukt
    .type    skalarProdukt, @function
skalarProdukt:
.LFB2: testl    %edi, %edi
        xorpd   %xmm1, %xmm1
        jle     .L3
        xorpd   %xmm1, %xmm1
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L4:    movsd   (%rsi,%rax,8), %xmm0
        mulsd   (%rdx,%rax,8), %xmm0
        addq    $1, %rax
        cmpl    %eax, %edi
        addsd   %xmm0, %xmm1
        jg      .L4
.L3:    movapd  %xmm1, %xmm0
        ret
```

dot(x,y)

```
double skalarProdukt(double[] x, double[] y) {
    double summe = 0;
    for (int index = 0; index < x.length; index++) {
        summe += x[index] * y[index];
    }
    return summe;
}
```



Historie ausgesuchter Programmiersprachen



Kurzübersicht wichtiger Bestandteile

- **Formatierung**

Die meisten Programmiersprachen erlauben, dass man zwischen den einzelnen Elementen eines Programms beliebige Zwischenräumen lassen kann. Die Nutzung dieser Eigenschaft kann die Lesbarkeit für einen Menschen sehr erhöhen.

- **Kommentare**

Kommentare können im Programm zur Dokumentation angegeben werden und haben für die Programmausführung keine Bedeutung. Beispiel: `// Beschleunigung`

- **Bezeichner**

Variablen, Methoden, Klassen,... müssen einen Namen haben. Bezeichner beginnen mit einem Buchstaben, gefolgt von beliebig vielen Buchstaben oder Ziffern.

Beispiel: `x`, `getX`, `langerBezeichner5`

- **Schlüsselwörter**

Reservierte Bezeichner mit einer festgelegten Bedeutung. Beispiel: `while`

- **Variablen**

Variablen haben einen Wert, den man ändern und erfragen kann. Beispiel: `x=x+1`

- **Ausdruck**

Operationen / einfache Formeln lassen sich in einem Ausdruck angeben. Ein Ausdruck lässt sich ausrechnen und liefert dann genau einen Wert. Beispiel: `(x+1) * 5`

- **Anweisung**

Siehe Diskussion Programmiermuster



Beispiel

Kommentar

Variable

Ausdruck

Anweisung

```
/**
 * Dieses Programm wandelt eine Dollar-Angabe in Euro um
 */

public class DollarNachEuro {
    public static void main(String[] args) {

        double dollarBetrag = 37.48; // Dollar-Betrag
        double umrechnungskurs = 0.72; // Umrechnungskurs

        // berechne den Euro-Betrag
        double euroBetrag = dollarBetrag * umrechnungskurs;

        // gebe das Ergebnis auf dem Bildschirm aus
        System.out.println("Der Euro-Betrag ist " + euroBetrag);
    }
}
```



Zwischenstand

- Programmiersprachen sind ein Kompromiss zwischen Menschen und Rechnern
- In ihnen gibt es Möglichkeiten zur Steuerung der Abfolge von Anweisungen und zur Manipulation von Daten/Werten
- Es gibt verschiedene Klassen von Programmiersprachen
- Die meisten universellen Programmiersprachen haben ähnliche Grundbestandteile

Reflektion

- Überlegen Sie sich eine beliebige Aufgabenstellung und überlegen anschließend, mit welcher Art von Werten Sie es dort zu tun haben könnten.
- Wozu dienen Bezeichner in Programmiersprachen? Wo haben Sie es bereits außerhalb solcher Sprachen mit Bezeichnern zu tun gehabt?



Aufbau von Sprachen

- Aufbau der deutschen Sprache:
 - **Alphabet** {a,b,c,...,z,A,B,C,...,Z,0,...,9,!,?,... }
 - **Grundwortschatz** {..., Mensa, Mensaessen, Menschheit,...}
 - **Grammatikregeln**: R59 Groß schreibt man das erste Wort eines Ganzsatzes.
- **Syntax**: korrekter Aufbau eines Satzes nach den Regeln der Sprache
- **Semantik**: Bedeutung eines (syntaktisch korrekten) Satzes
- Wir bilden **Beispielsätze**:
 - Dieser Mensch rennt. syntaktisch und semantisch korrekt
 - Dieser Mensch pennt. syntaktisch und semantisch korrekt
 - Dieser Mensch krennt. syntaktisch inkorrekt
 - Dieser Baum rennt. syntaktisch korrekt, semantisch inkorrekt

Definitionen

- Ein **Alphabet** ist eine endliche nichtleere Menge $\Sigma = \{a_1, \dots, a_n\}$ von Symbolen
- Ein **Wort** w ist eine endliche Sequenz von Symbolen aus dem gewählten Alphabet Σ
- Die **Menge aller Wörter über einem Alphabet Σ** (Bezeichnung: Σ^*) ist induktiv definiert durch:
 1. $\varepsilon \in \Sigma^*$ (leeres Wort)
 2. $a \in \Sigma \Rightarrow a \in \Sigma^*$ (einzelne Symbole sind Wörter)
 3. $x, y \in \Sigma^* \Rightarrow xy \in \Sigma^*$ (zusammengesetzte Wörter)
- Die **Länge eines Wortes** $w \in \Sigma^*$ (Anzahl Symbole) bezeichnet man mit $|w|$
- Sei Σ ein Alphabet. $L \subseteq \Sigma^*$ heißt **formale Sprache über Σ** .

Beispiele

- Alphabet $\Sigma_{\text{bin}} = \{0,1\}$
- Alphabet $\Sigma_{\text{dez}} = \{0,1,2,3,4,5,6,7,8,9\}$
- Beispiele für Wörter über Σ_{bin} : ε , 0, 1, 00, 01, 10, 11, 000,...
- $|\varepsilon| = 0$
- $|011| = 3$
- $\Sigma_{\text{bin}}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, 0010, \dots\}$
- $L_1 = \{0,1, 000001\}$ ist formale Sprache über Σ_{bin}
- $L_2 = \Sigma_{\text{bin}}^*$ ist formale Sprache über Σ_{bin}



Grammatiken für Sprachdefinitionen

- Bis jetzt in alle Beispielen von Sprachdefinitionen: Aufzählung der möglichen Wörter
- Was bedeuten dann die drei Punkte in $\Sigma_{\text{bin}}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, 0010, \dots\}$?
- Intuitiv klar, aber nicht formal fundiert
- Beschreibung von Sprachen unendlicher Mächtigkeit durch allgemeine Bildungsregeln: **formale Grammatiken**



Chomsky-Grammatik

- Eine **Chomsky-Grammatik** hat die Form $G = \langle \Sigma, N, P, S \rangle$ mit
 - Σ endlich (Terminalsymbole, Alphabet)
 - N endlich, N und Σ disjunkt (Nichtterminalsymbole, Hilfssymbole)
 - $P \subseteq ((N \cup \Sigma)^* \setminus \Sigma^*) \times (N \cup \Sigma)^*$ (Produktionen, Grammatikregeln)
 - $S \in N$ (Startsymbol)
- Statt einer Produktion (x,y) schreibt man auch $x \rightarrow y$
- **Beispiel:** $G_{\text{bin}} = \langle \Sigma_{\text{bin}}, N, P, S \rangle$ mit $N = \{\text{BinZahl}, \text{BinRest}\}$, $S = \text{BinZahl}$, P enthält die Produktionen:
 1. $\text{BinZahl} \rightarrow 0$
 2. $\text{BinZahl} \rightarrow 1$
 3. $\text{BinZahl} \rightarrow 1 \text{ BinRest}$
 4. $\text{BinRest} \rightarrow 0 \text{ BinRest}$
 5. $\text{BinRest} \rightarrow 1 \text{ BinRest}$
 6. $\text{BinRest} \rightarrow 0$
 7. $\text{BinRest} \rightarrow 1$

Ableitung

- Sei $G = \langle \Sigma, N, P, S \rangle$ eine Grammatik. $u, v \in (N \cup \Sigma)^*$. Dann sind definiert:

- $u \vdash v \Leftrightarrow \exists x, y, y', z \in (N \cup \Sigma)^*, u = xyz, v = xy'z, y \rightarrow y' \in P.$
 u ist in einem Schritt (direkt) ableitbar nach v .

- $u \vdash^n v \Leftrightarrow \exists u_0, \dots, u_n \in (N \cup \Sigma)^*, u = u_0 \vdash \dots \vdash u_n = v$
Die Folge $u_0 \vdash \dots \vdash u_n$ heißt Ableitung (der Länge n).

- $u \vdash^* v \Leftrightarrow \exists n \geq 0: u \vdash^n v$

- $u \in (N \cup \Sigma)^*$ ist **ableitbar in $G = \langle \Sigma, N, P, S \rangle$** $\Leftrightarrow S \vdash^* u$
- $L(G) := \{w \in \Sigma^* \mid S \vdash^* w\}$ ist die **von G erzeugte Sprache**.

- Einige **mögliche Ableitungen** in G_{bin} :
 - $\text{BinZahl} \vdash 0$
 - $\text{BinZahl} \vdash 1 \text{ BinRest} \vdash 10 \text{ BinRest} \vdash 100$
- $L(G_{\text{bin}}) = \{0\} \cup$ Menge der 0-1-Wörter ohne führende 0



Beispiel 2

- $G_{\text{dez}} = \langle \Sigma_{\text{dez}}, N, P, S \rangle$ mit
 - $N = \{\text{DecimalNumeral}, \text{Digits}, \text{Digit}, \text{NonZeroDigit}\}$
 - $S = \text{DecimalNumeral}$
 - P enthält die Regeln:
 1. $\text{DecimalNumeral} \rightarrow 0$
 2. $\text{DecimalNumeral} \rightarrow \text{NonZeroDigit}$
 3. $\text{DecimalNumeral} \rightarrow \text{NonZeroDigit Digits}$
 4. $\text{Digits} \rightarrow \text{Digit}$
 5. $\text{Digits} \rightarrow \text{Digits Digit}$
 6. $\text{Digit} \rightarrow 0$
 7. $\text{Digit} \rightarrow \text{NonZeroDigit}$
 8. $\text{NonZeroDigit} \rightarrow 1$ (analog für 2,...,9)
- **Mögliche Ableitung:** $\text{DecimalNumeral} \mid\text{---} \text{NonZeroDigit Digits} \mid\text{---} 1 \text{ Digits} \mid\text{---} 1 \text{ Digits Digit} \mid\text{---} 1 \text{ Digit Digit} \mid\text{---} 10 \text{ Digit} \mid\text{---} 10 \text{ NonZeroDigit} \mid\text{---} 105$
- $L(G_{\text{dez}}) = \{0\} \cup \text{Menge der Dezimaldarstellungen natürlicher Zahlen ohne führende 0}$

Chomsky-Hierarchie

Aus Grammatikdefinition:
 $P \subseteq ((N \cup \Sigma)^* \setminus \Sigma^*) \times (N \cup \Sigma)^*$

- Eine Grammatik $G = \langle \Sigma, N, P, S \rangle$ ist vom
 - Typ 0, falls $P \subseteq ((N \cup \Sigma)^* \setminus \Sigma^*) \times (N \cup \Sigma)^*$
 - Typ 1 oder kontextsensitiv, falls vom Typ 0 und für jede Regel $\alpha \rightarrow \beta$ zusätzlich gilt: $|\alpha| \leq |\beta|$
 - Typ 2 oder kontextfrei, falls $P \subseteq N \times (N \cup \Sigma)^*$
 - Typ 3 oder regulär, falls $P \subseteq N \times (N\Sigma \cup \Sigma \cup \{\varepsilon\})$ (linkslinär) oder $P \subseteq N \times (\Sigma N \cup \Sigma \cup \{\varepsilon\})$ (rechtslinär)
- Beispiele zu Regeln für Typ i Grammatiken:
 - Typ 3: $\text{Binzahl} \rightarrow 0$, $\text{Binzahl} \rightarrow 1$, $\text{Binzahl} \rightarrow 1A$, $A \rightarrow 0A$, $A \rightarrow 1A$, $A \rightarrow 0$, $A \rightarrow 1$
 - Typ 2: $\text{DecimalNumeral} \rightarrow 0$, $\text{DecimalNumeral} \rightarrow \text{NonZeroDigit}$, $\text{DecimalNumeral} \rightarrow \text{NonZeroDigits Digits}$, $\text{Digits} \rightarrow \text{Digit}$, ...
- Ab jetzt für unsere Zwecke Konzentration ausschließlich auf kontextfreie und reguläre Grammatiken



Zwischenstand

- Formale Sprachen sind eine Menge von Wörter (Folge von Symbolen).
- Regelwerke (Grammatiken) definieren Sprachen exakt.
- Eine Ableitung ist eine Folge von Regelanwendungen.
- Kann man vom Startsymbol eine Ableitungsfolge zu einem Wort nur aus Terminalsymbolen angeben, so ist dieses Wort in der Sprache erhalten.
- Nur die Chomsky-Klassen 2 und 3 sind für Programmiersprachen relevant.
- Solche Sprachen sind relativ einfach aufgebaut.

Reflektion

- Finden Sie zur Grammatik G_{dez} eine Ableitung für die Zahl 31415



Backus-Naur-Form

- Kontextfreie Grammatiken in der eingeführten Form haben **einige Nachteile**:
 - Viele (ähnliche) Produktionen machen die Produktionenmenge schnell **unübersichtlich**
 - Eine solche Grammatik ist **nicht maschinenlesbar**
- Deshalb **Backus-Naur-Form (BNF)**
- **Notation der Produktionen in der Form**:
 - Nichtterminalsymbol werden immer **mit <> geklammert**
 - Statt des Zeichens \rightarrow verwendet man **::=**
 - Haben mehrere Regeln das gleiche Nichtterminalsymbol auf der linken Seite der Regel, so können diese **Regeln zusammengefasst werden**, indem man die rechten Seiten der Regeln jeweils durch | trennt



Beispiel

- Produktionen von G_{bin} in BNF:
 - $\langle \text{Binzahl} \rangle ::= 0 \mid 1 \mid 1\langle \text{BinRest} \rangle$
 - $\langle \text{BinRest} \rangle ::= 0 \mid 1 \mid 0\langle \text{BinRest} \rangle \mid 1\langle \text{BinRest} \rangle$
- Produktionen von G_{dez} in BNF:
 - $\langle \text{DecimalNumeral} \rangle ::=$
 - 0
 - $\mid \langle \text{NonZeroDigit} \rangle$
 - $\mid \langle \text{NonZeroDigit} \rangle \langle \text{Digits} \rangle$
 - $\langle \text{Digits} \rangle ::=$
 - $\langle \text{Digit} \rangle$
 - $\mid \langle \text{Digits} \rangle \langle \text{Digit} \rangle$
 - $\langle \text{Digit} \rangle ::=$
 - 0
 - $\mid \langle \text{NonZeroDigit} \rangle$
 - $\langle \text{NonZeroDigit} \rangle ::=$
 - 1
 - $\mid 2$
 - $\mid \dots$
 - $\mid 9$

Extended Backus-Naur Form

- Die Regel $\langle \text{BinRest} \rangle ::= 0 \mid 1 \mid 0\langle \text{BinRest} \rangle \mid 1\langle \text{BinRest} \rangle$ besagt eigentlich, dass $\langle \text{BinRest} \rangle$ abgeleitet werden kann zu einer beliebig langen nichtleeren Folge von 0 und 1
- Die **Extended Backus Naur Form (EBNF)** erweitert die BNF dahingehend, dass solche, oft vorkommende Fälle in Regeln einfacher notiert werden können
- **Erweiterungen der EBNF zur BNF:**
 - In den rechten Seiten von Regeln kann man [] und { } verwenden.
 - [x] bedeutet, dass x 0 oder 1 mal vorkommen kann (optional)
 - { x } bedeutet, dass x beliebig oft (inkl. 0 mal) vorkommen kann
 - Terminalsymbole werden **typografisch hervorgehoben**
 - **Runde Klammern ()** kann man zur Gruppierung verwenden

Beispiel

- Produktionen von G_{bin} in EBNF:
 - $\langle \text{Binzahl} \rangle ::= 0 \mid 1 \mid 1 \langle \text{BinRest} \rangle$
 - $\langle \text{BinRest} \rangle ::= (0 \mid 1) \{ 0 \mid 1 \}$
- Produktionen von G_{dez} in EBNF:
 - $\langle \text{DecimalNumeral} \rangle ::= 0 \mid \langle \text{NonZeroDigit} \rangle [\langle \text{Digits} \rangle]$
 - $\langle \text{Digits} \rangle ::= [\langle \text{Digits} \rangle] \langle \text{Digit} \rangle$
 - $\langle \text{Digit} \rangle ::= 0 \mid \langle \text{NonZeroDigit} \rangle$
 - $\langle \text{NonZeroDigit} \rangle ::= 1 \mid 2 \mid \dots \mid 9$

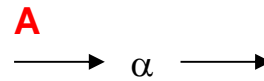
Syntaxdiagramme

- BNF und EBNF sind entwickelt worden zur einfachen maschinellen Verarbeitung (und nicht in erster Linie für einen Menschen)
- Syntaxdiagramme sind entwickelt worden, um einem Menschen zum Beispiel in einem Buch zu einer Programmiersprache einfach die Syntax zugänglich zu machen
- Syntaxdiagramme sind aufgebaut:
 - Zu jedem Nichtterminalsymbol auf der linken Seite einer Produktion gibt es ein eigenes Teildiagramm
 - Terminalsymbole werden in Kreise/Ovale eingeschlossen
 - Nichtterminalsymbole werden in Rechtecke eingeschlossen
 - Pfeile geben einen möglichen Abarbeitungsweg an
- Nachfolgend werden die Übersetzungsregeln definiert, wie beliebige Regeln gegeben in EBNF in ein Syntaxdiagramm übersetzt werden

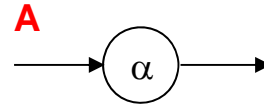


Übersetzungsregeln EBNF \rightarrow Syntaxdiagramm

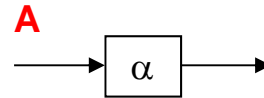
Für eine Regel $\langle A \rangle ::= \alpha$
 $A \in N, \alpha \in (N \cup \Sigma)^*$



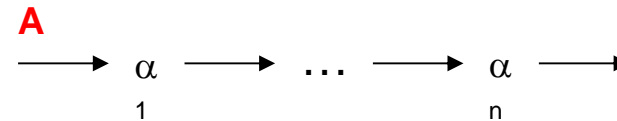
Falls α Terminalsymbol



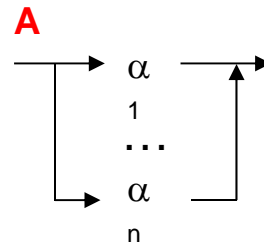
Falls α Nichtterminalsymbol



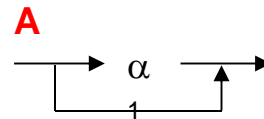
Falls $\alpha = \alpha_1 \dots \alpha_n$



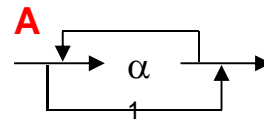
Falls $\alpha = \alpha_1 \mid \dots \mid \alpha_n$



Falls $\alpha = [\alpha_1]$



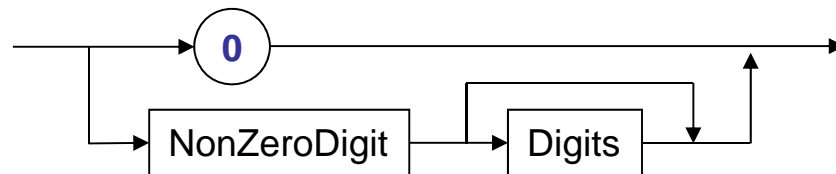
Falls $\alpha = \{ \alpha_1 \}$



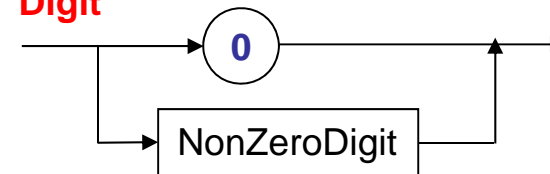
Beispiel

- Zur Erinnerung: Produktionen von G_{dez} in BNF:
 - $\langle \text{DecimalNumeral} \rangle ::= 0 \mid \langle \text{NonZeroDigit} \rangle [\langle \text{Digits} \rangle]$
 - $\langle \text{Digits} \rangle ::= [\langle \text{Digits} \rangle] \langle \text{Digit} \rangle$
 - $\langle \text{Digit} \rangle ::= 0 \mid \langle \text{NonZeroDigit} \rangle$
 - $\langle \text{NonZeroDigit} \rangle ::= 1 \mid 2 \mid \dots \mid 9$

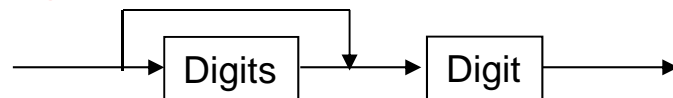
DecimalNumeral



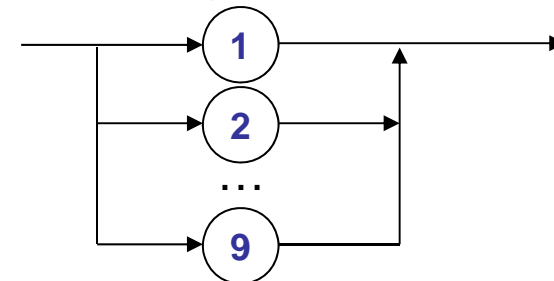
Digit



Digits



NonZeroDigit



Zwischenstand

- Grammatikregeln lassen sich auf verschiedene Weise angeben (Chomsky, BNF, EBNF, Syntaxdiagramm)

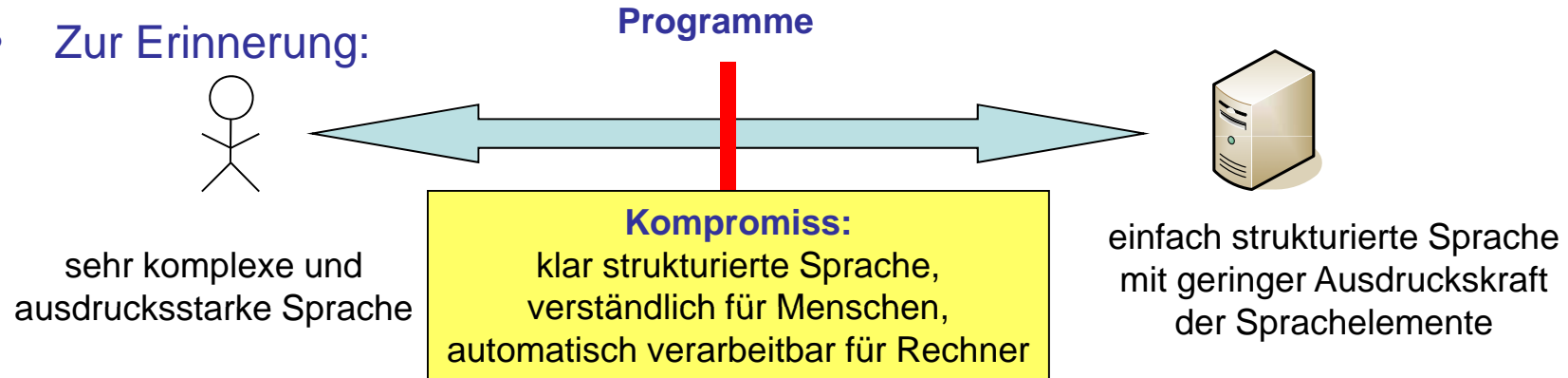
Reflektion

- Erläutern Sie ihrem Nachbarn die praktischen Vorteile der BNF und EBNF.
- Welche Vor- und Nachteile haben Syntaxdiagramme?



Ausführen von Programmen

- Zur Erinnerung:



- Frage: Wie kann ein Rechner nun Programme ausführen?
 - Antwort 1: Interpretierer
 - Antwort 2: Übersetzer / Compiler
 - Antwort 3: Mischung aus beiden

Beispielgrammatik

- Regeln einer Beispielgrammatik (kein Java):

<Programm>	::= { <Anweisung> }
<Anweisung>	::= <Leseanweisung> <Schreibanweisung> <Zuweisung> <Schleife>
<Leseanweisung>	::= read <Bezeichner> ;
<Schreibanweisung>	::= write <Bezeichner> ;
<Zuweisung>	::= <Bezeichner> = <Ausdruck> ;
<Schleife>	::= while <relAusdruck> do { <Anweisung> } endwhile
<Ausdruck>	::= <Zahl> <Bezeichner> <Ausdruck> + <Ausdruck> <Ausdruck> * <Ausdruck>
<relAusdruck>	::= <Ausdruck> < <Ausdruck>
<Bezeichner>	::= ...
<Zahl>	::= ...

- Beispielprogramm:

```
read x;
i = 1;
while i < 3 do
    x = x * x;
    i = i + 1;
endwhile
write x;
```



Interpretierer

- Aufgaben eines Interpretierers
 - Zerlegung des Programms in Grundsymbole (**Token**)
 - Analyse des Programms (**Erkennen der Struktur**) anhand von Grammatikregeln
 - **Ausführung von Programmcode** entsprechend erkannter Strukturelemente
- Token sind **relativ einfach aufgebaut** (in der Praxis üblicherweise Typ-3-Grammatik) und können effizient erkannt werden
- Token werden dabei unterschieden in verschiedene **Token-Klassen**:
 - **Konstanten / Literale** (Beispiel: 1, 3)
 - **Bezeichner** (Beispiel: x, i)
 - **Schlüsselwörter** (Beispiel: read, while)
 - **Sonderzeichen** (Beispiel: =, <, *)
- Nach Zerlegung:

read x ; i = 1 ; while i < 3 do x = x * x ; i = i + 1 ; endwhile write x ;

Ablauf der Interpretierung des Beispiels

```
read x ; i = 1 ; while i < 3 do x = x * x ; i = i + 1 ; endwhile write x ;
```



1. Abarbeitung der Token-Sequenz an der aktuellen Arbeitsposition

- nach `read` muss ein Bezeichner kommen gefolgt von einem Semikolon
- Bezeichner muss evtl. in einer Name-Wert-Tabelle neu angelegt werden
- Einlesen eines Wertes von der Tastatur und Speichern dieses Wertes unter dem Namen in der Tabelle

```
read x ; i = 1 ; while i < 3 do x = x * x ; i = i + 1 ; endwhile write x ;
```




2. Aufgrund fehlenden Schlüsselworts muss eine **Zuweisung** vorliegen

- Erkennen des Bezeichners (evtl. neu anlegen in Tabelle) und =
- Abarbeitung des nachfolgenden Ausdrucks bis zum schließenden Semikolon
- Berechnung des Ausdruckswertes und Speichern in Name-Wert-Tabelle

Ablauf der Interpretierung des Beispiels

```
read x ; i = 1 ; while i < 3 do x = x * x ; i = i + 1 ; endwhile write x ;
```



3. Schleife bearbeiten

- Token-Position merken
- Nachfolgenden Vergleichsausdruck ($i < 3$) bis zum `do` erkennen und Wert berechnen
- Falls der Ergebniswert `false` ist, `endwhile` suchen und dahinter weitermachen
- Falls der Ergebniswert `true` ist, den Schleifenrumpf abarbeiten bis zum `endwhile`. Anschließend an gespeicherter alter Token-Position wieder aufsetzen
- (Frage: was passiert, wenn ein Schleifenrumpf eine weitere Schleife enthält)

• ...

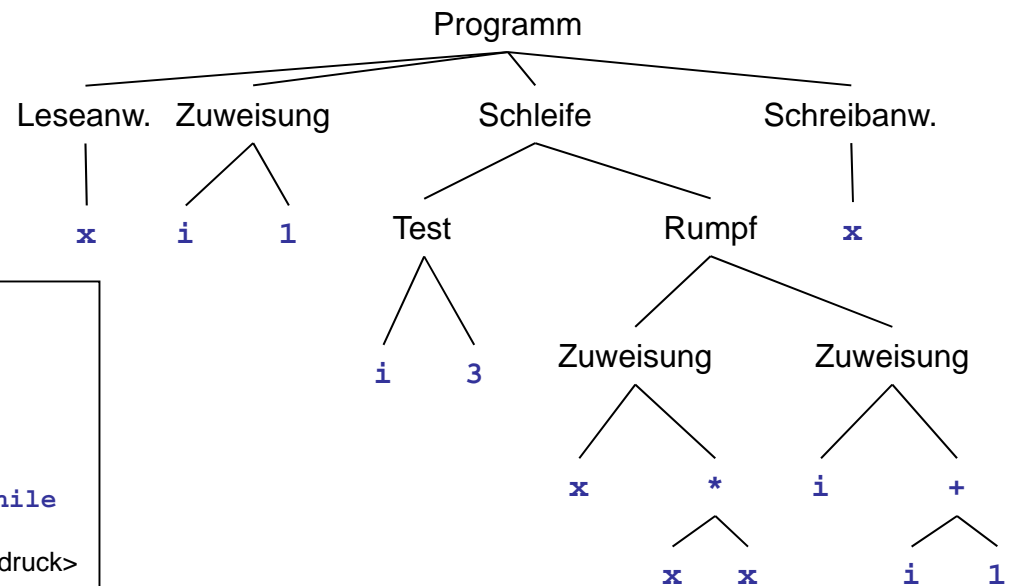


Compiler

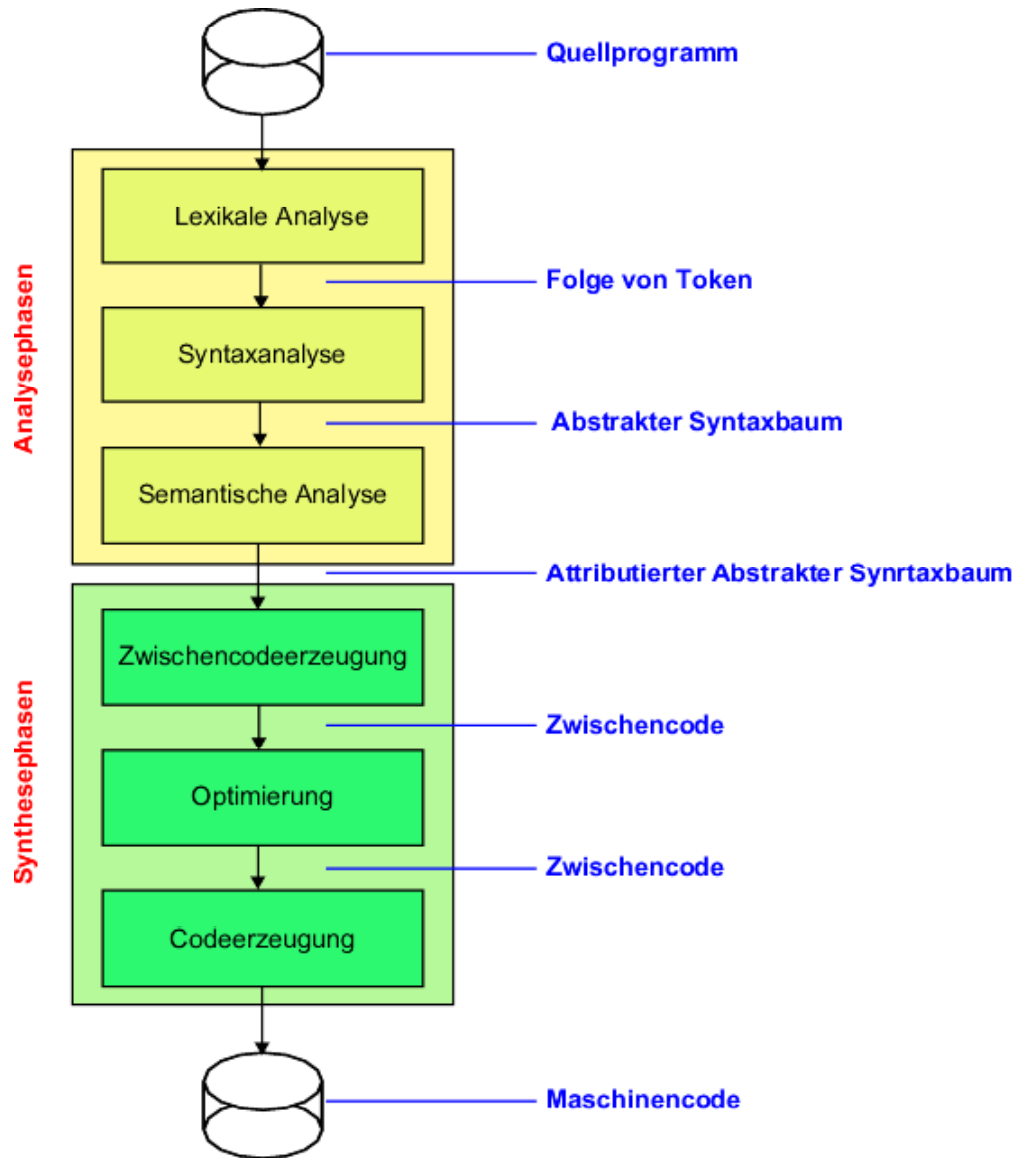
- Ein Compiler analysiert (genau einmal) das **Quellprogramm (Source)** und generiert daraus ein **Objektprogramm** für den Zielrechner
- **Analysephase:** Finden einer Ableitung vom Startsymbol der Grammatik zum vorliegenden Programm (mehrere Strategien dazu möglich; sehr komplex)
- Kennt man aufgrund der Analyse die Struktur des Programms, so kann man daraus entsprechend den Strukturelementen Code generieren (**Synthese**)
- **Beispiel zur Programmstruktur:**

```
read x;
i = 1;
while i < 3 do
  x = x * x;
  i = i + 1;
endwhile
write x;
```

<Programm>	::= { <Anweisung> }
<Anweisung>	::= <Leseanweisung> <Schreibanweisung> <Zuweisung> <Schleife>
<Leseanweisung>	::= read <Bezeichner> ;
<Schreibanweisung>	::= write <Bezeichner> ;
<Zuweisung>	::= <Bezeichner> = <Ausdruck> ;
<Schleife>	::= while <relAusdruck> do { <Anweisung> } endwhile
<Ausdruck>	::= <Zahl> <Bezeichner> <Ausdruck> + <Ausdruck> <Ausdruck> * <Ausdruck>
<relAusdruck>	::= <Ausdruck> < <Ausdruck>
...	



Phasen eines Compilers



1. Zerlegung der Eingabedaten in Folge von Token (Grundsymbole der Sprache wie Bezeichner, Konstanten, Sonderzeichen, Trennzeichen)
2. Erkennung der Programmstruktur durch Ableitung des Startsymbols der Grammatik zur Tokenfolge
3. Semantische Überprüfung, beispielsweise, dass eine Variable vor ihrer Nutzung deklariert sein muss
4. Aus der Struktur des Programms wird ein Code erzeugt, der ähnlich einer Maschinensprache ist
5. Optimierung des Zwischencodes hinsichtlich Zeit und/oder Speicherbedarf
6. Generierung des Codes für einen Zielrechner



Gegenüberstellung Interpretierer / Compiler

- **Interpreter**
 - analysieren **Schritt für Schritt** nur den Programmteil, der als nächstes zur Abarbeitung ansteht
 - Analyse ist schneller als in einem Compiler, wiederholt sich aber mit jeder Interpretierung dieses Programmstücks
 - gut geeignet für Programme, die **selten / einmalig laufen**
 - schlecht geeignet für Programme, die wiederholt / effizient ablaufen sollen
 - **Einsatzgebiete:** Kontrollsprachen (Shell), interaktive Umgebungen
- **Compiler**
 - analysieren und übersetzen das **komplette (Teil-) Programm**
 - umfassende Fehleranalyse und Programmoptimierung möglich
 - Übersetzungsvorgang ist nur ein mal nötig **für beliebig viele Programmausführungen**
 - primär geeignet für **oft laufende Programme** oder wo es auf Effizienz ankommt
 - **Einsatzgebiete:** größere Programme, die oft ausgeführt werden (das sind die meisten Programme)



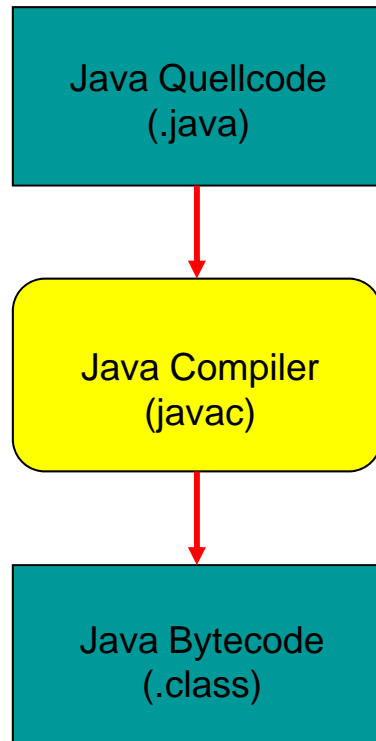
Virtuelle Maschinen

- Generierter Code eines Compilers ist maschinenspezifisch
- **Problem:** ich möchte ein Programm im Netz zum Download anbieten, das auf allen Rechnern der Welt laufen soll (→ geht nicht)
- **Lösung:** Übersetzung für einen hypothetischen Prozessor
- Beim **Start des Programms** wird der Code für diesen hypothetischen Prozessor dann auf dem Zielrechner
 - **interpretiert** (wenig Aufwand, langsame Ausführung)
 - **übersetzt** (größerer Aufwand, schnelle Ausführung)
- Dazu benötigt man allerdings **Laufzeitunterstützung auf jedem Rechner**, auf dem solch ein Programm gestartet werden soll
- Java hat dazu die **JVM (Java Virtual Machine)** als Teil der **JRE (Java Runtime Environment)**
- Microsoft hat dafür die **.NET Umgebung**
 - Common Intermediate Language (CIL): Prozessorsprache
 - Common Language Runtime (CLR): Laufzeitunterstützung

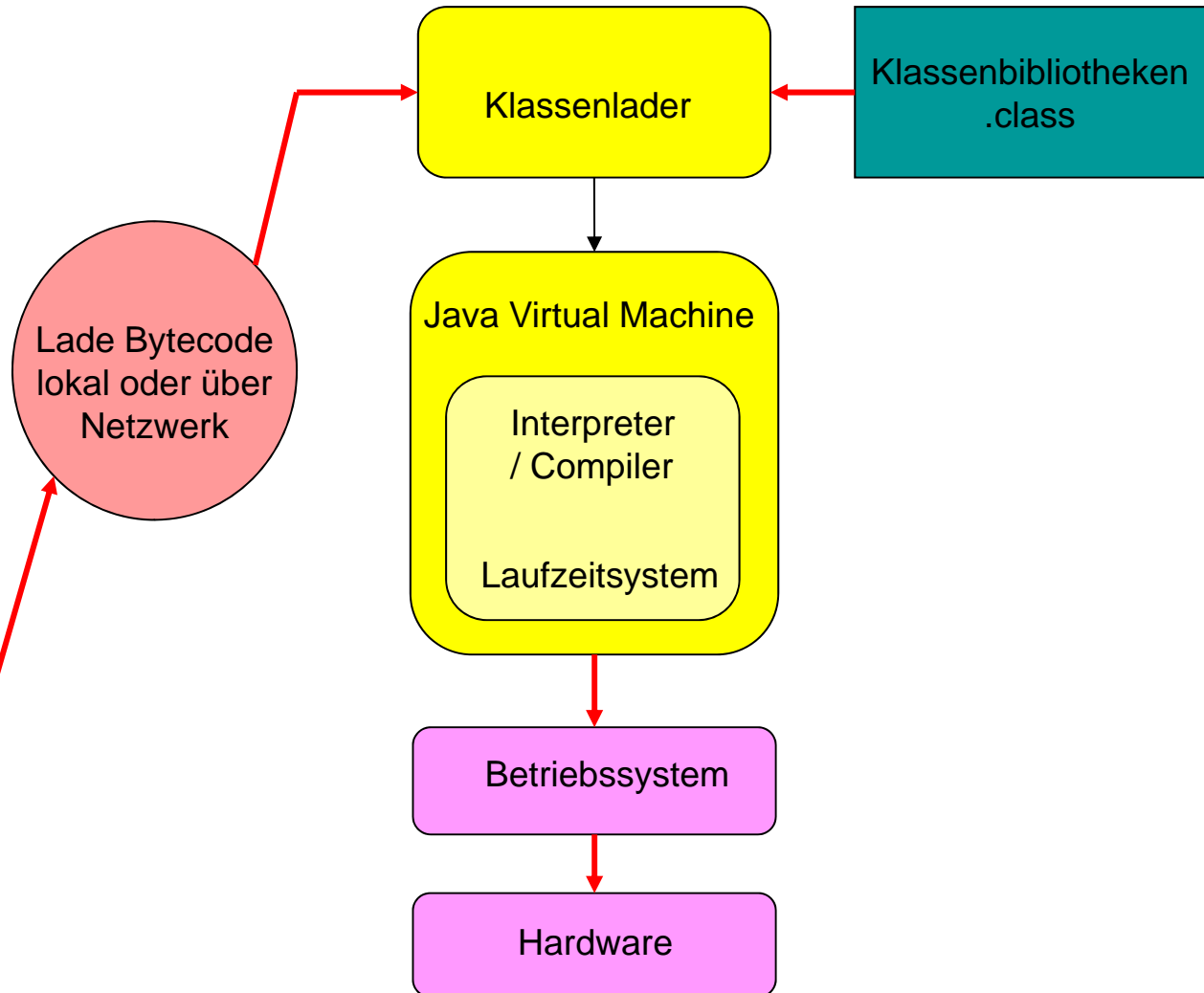


Java Umgebung

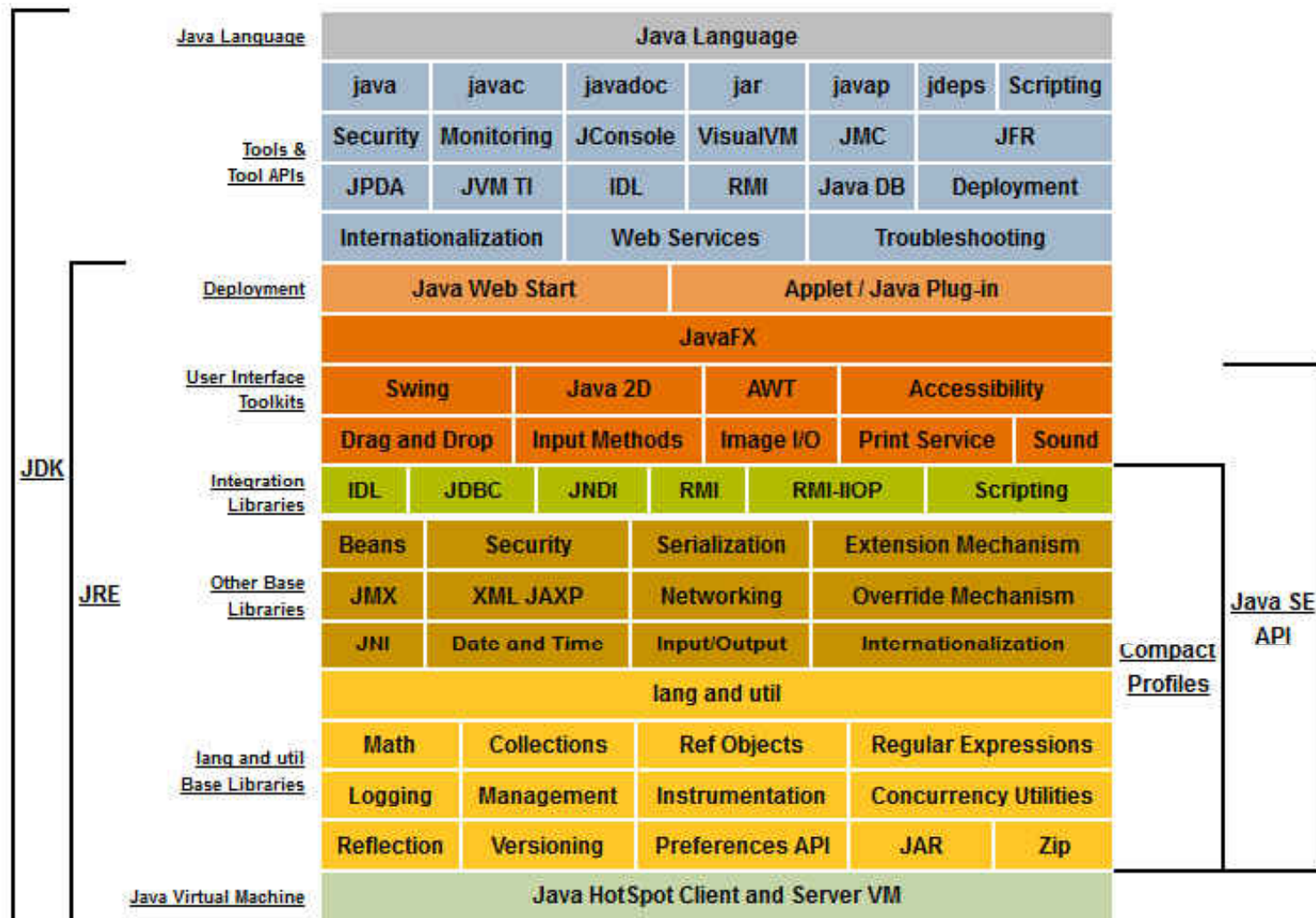
Entwicklungsumgebung



Laufzeitumgebung (JRE)



Java Software Development Kit (SDK 8)



Quelle: <http://docs.oracle.com/javase/8/docs/>



Zwischenstand

- Interpreter gehen schrittweise durch einen Programm durch und schauen jeweils, was an der aktuellen Betrachtungsstelle zu tun ist.
- Compiler übersetzen ein Programm in den Binärcode des Zielrechners, wo dieser beliebig oft ohne weiteren Aufwand ausgeführt werden kann.
- Compilern haben mit einer virtuellen Maschine nur eine Zielarchitektur. Auf einem realen Rechner muss dann aber der Code dieses virtuellen Rechners auf den vorliegenden realen Rechner abgebildet werden .
- Für die Sprache Java ist die Java Virtual Machine definiert.

Reflektion

- Erläutern Sie ihrem Nachbarn die Vor- und Nachteile einer virtuellen Maschine.
- Könnte man eine virtuelle Maschine wie die JVM auch als realen Rechner bauen?
- Könnte man einen realen Rechner auch als virtuelle Maschine angeben?



Zusammenfassung

- Programmiersprachen als **Kompromiss** zwischen Bedürfnissen von Menschen und Möglichkeiten von Prozessoren
- Verschiedene **Klassen von Programmiersprachen**
- Programmiersprachen werden **definiert durch Grammatiken**. Die Produktionen der Grammatik definieren die **Struktur der Sprache**.
- Angabe von Grammatikregeln durch **direkte Grammatikregeln (Theoretiker)**, **BNF / EBNF (maschinenverarbeitbar)** und **Syntaxdiagramme (Menschen)**
- **Compiler** zur Übersetzung eines Programms für einen Rechner nötig. Der erzeugte Maschinencode ist nur auf dieser Prozessorfamilie / diesem Betriebssystem lauffähig
- Statt Code für einen speziellen Prozessor zu erzeugen, kann Code für eine **virtuelle Maschine** erzeugt werden und dann auf einem beliebigem Rechner **bei** Start des Programms interpretiert / übersetzt werden. Dazu ist eine **Laufzeitunterstützung** nötig (Beispiele: JVM/JRE, .NET)

