Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

# Programmierung 1

## Für die Studiengänge BI und BWI

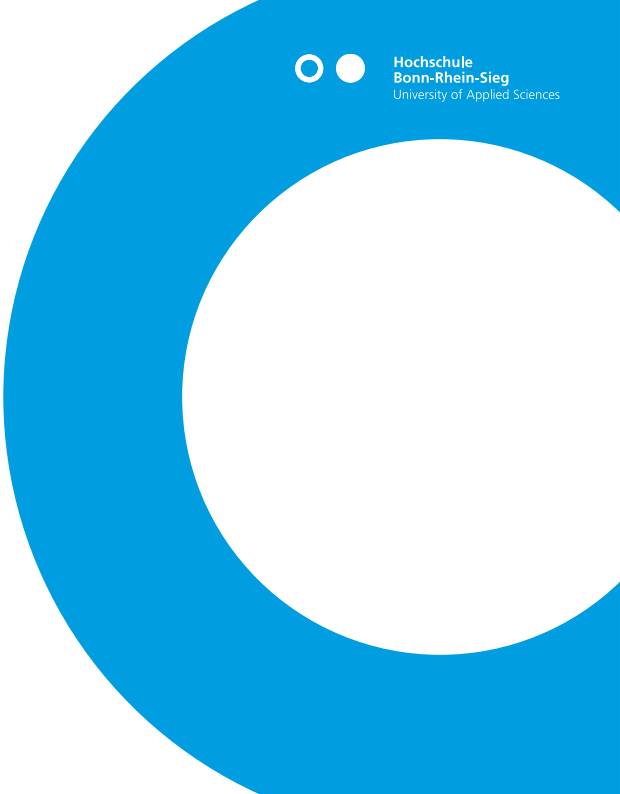
WiSe 25/26

M. Sc. Moritz Balg

# Inhaltsverzeichnis

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

# Programmiersprachen



Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

# 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)

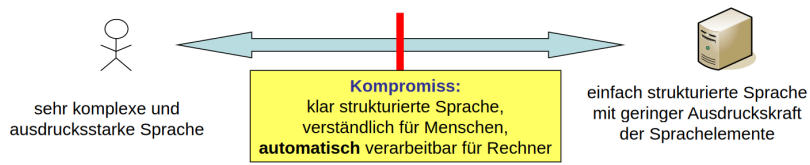



Abbildung 1: Programmiersprachen sind ein Kompromiss

# 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
- 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: Ausdruck, primitiver Datentyp, Klasse

# Klassen von Programmiersprachen



- Es existieren verschiedene Klassen von Programmiersprachen:
  - ⚙️ Maschinenorientierte Programmiersprachen
  - 🧮 Problemorientierte Programmiersprachen
  - 🌍 Universelle Programmiersprachen
- Im weiteren Verlauf betrachten wir die Bildung des Skalarprodukts in diesen Klassen aussehen kann.

$$\text{Skalarprodukt}(x, y) = \Sigma_{i=1}^{n} x_i * y_i$$

# Maschinenorientierte Programmiersprachen



- Maschinenorientierte Programmiersprachen bieten im Wesentlichen die Instruktionen des jeweiligen Prozessors an
- Nachteil: Programme sind nicht portabel und schlecht lesbar
- Bsp.: Assemblycode

```
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
```

Listing 1: Berechnung des Skalarprodukts in x86-64 Assemblycode

# Problemorientierte Programmiersprachen



- Problemorientierte Programmiersprachen sind maßgeschneidert für eine bestimmte Problemklasse.
- Nachteil: Aber auch nur dafür
- Bsp.: Matlab, SQL, R, ...

```
dot(A,B)
```

Listing 2: Berechnung des Skalarprodukts in Matlab

# Universelle Programmiersprachen



- Universelle Programmiersprachen bieten alle gängigen Bausteine zur Programmierung an
- Bsp.: Java, C, Python, JS, ...

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

Listing 3: Berechnung des Skalarprodukts in Java

# Historie ausgesuchter Programmiersprachen



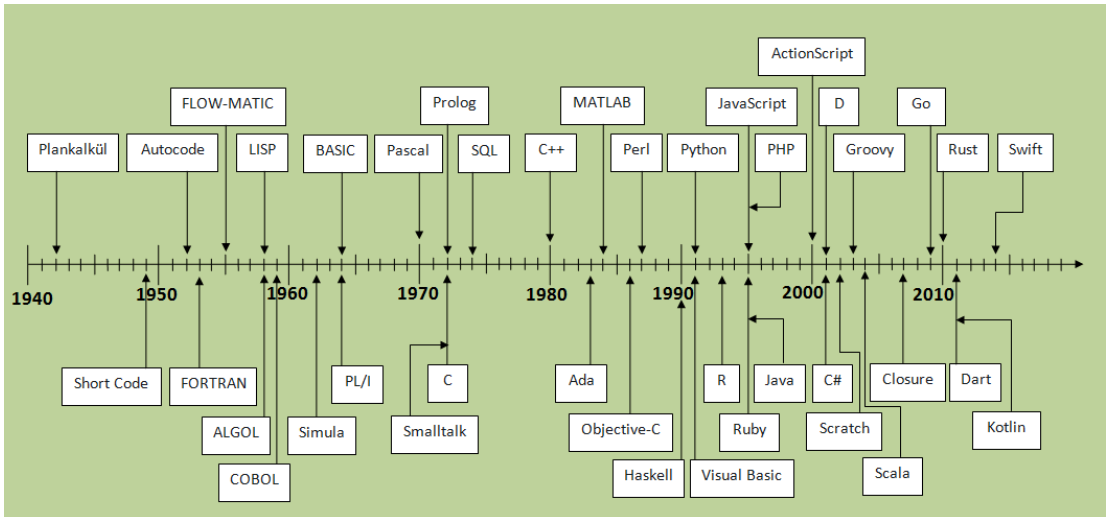



Abbildung 3: Übersicht einiger Programmiersprachen und deren Erscheinungsdatum (*History of programming languages*, 2019)

# 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

# Kurzübersicht wichtiger Bestandteile (ii)



- 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

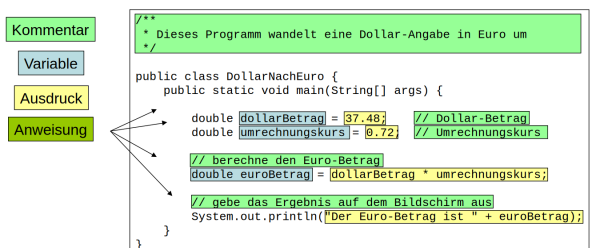



Abbildung 4: Bestandteile von Programmiersprachen

# 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

## Reflexion

- 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?

# Wie kann ein Computer „Sprache“ verarbeiten?



- In den folgenden Abschnitten werden wir uns mit sehr einfachen Sprachen beschäftigen. Sie sollen die theoretischen Grundlagen zugänglicher machen.
- Wie kann ein Browser feststellen, ob eine E-Mail Adresse gültig ist?

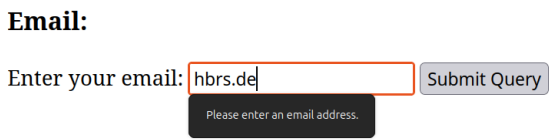



Abbildung 5: Verifikation einer E-Mail Adresse in einem Webbrowser

- Es müssen Regeln für den Aufbau einer E-Mail Adresse definiert sein
- Frage: Wie könnten diese aussehen?

# Verarbeitung von Sprachen



- Ist eine gegebene Eingabe gültig (Erfüllt sie die Regeln einer Grammatik)?
- Erfüllt beispielsweise die eingegebene E-Mail Adresse alle Anforderungen?
- Wir möchten nur Wörter der Form $ab^na \mid n \in \mathbb{N}_0$ zulassen
  - Bsp.: aa, aba, abba, ab...ba
- Für diese Sprache kann man einen deterministischer endlicher Automat (DEA) konstruieren.
  *Die Automatentheorie wird in dieser Veranstaltung nicht weiter vertieft und dient nur der Veranschaulichung*

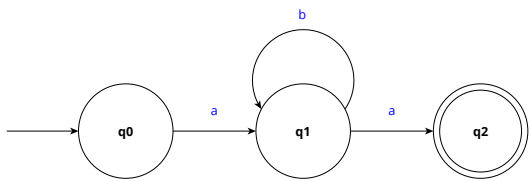



Abbildung 6: Deterministischer endlicher Automat für die Sprache $L = \{ab^na \mid n \in \mathbb{N}_0\}$

# 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. $\rightarrow$ syntaktisch und semantisch korrekt
  - Dieser Mensch pennt. $\rightarrow$ syntaktisch und semantisch korrekt
  - Dieser Mensch krennt. $\rightarrow$ syntaktisch inkorrekt
  - Dieser Baum rennt. $\rightarrow$ syntaktisch korrekt, semantisch inkorrekt

# Alphabet und Wörter



## Definition: Alphabet

Ein Alphabet ist eine endliche nichtleere Menge $\Sigma = \{a_1, ..., a_n\}$ von Symbolen

## Definition: Wort

Ein Wort $w$ ist eine endliche Sequenz von Symbolen aus dem gewählten Alphabet $\Sigma$

## Definition: Menge aller Wörter

Die Menge aller Wörter über einem Alphabet $\Sigma$ (Bezeichnung: $\Sigma^*$) ist induktiv definiert durch:

1. $\varepsilon \in \Sigma^*$ (leeres Wort)
2. $a \in \Sigma \Rightarrow a \in \Sigma^*$ (einzelne Symbole sind Wörter)
3. $x, y \in \Sigma^* \Rightarrow \mathrm{yx} \in \Sigma^*$ (zusammengesetzte Wörter)

## Definition: Länge eines Wortes

Die Länge eines Wortes $w \in \Sigma^*$ (Anzahl Symbole) bezeichnet man mit $|w|$

# Formale Sprachen



## Definition: Formale Sprache

Sei $\Sigma$ ein Alphabet. $L \subseteq \Sigma^*$ heißt formale Sprache über $\Sigma$.

## 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 $\Sigma_{\text{bin}}$ : $\varepsilon, 0, 1, 00, 01, 10, 11, 000, ...$
- $|\varepsilon| = 0$; $|011| = 3$
- $\Sigma^*_{\text{bin}} = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, 0010, ...\}$
- $L_1 = \{0, 1, 000001\}$ ist formale Sprache über $\Sigma_{\text{bin}}$
- $L_2 = \Sigma^*_{\text{bin}}$ ist formale Sprache über $\Sigma_{\text{bin}}$

# Grammatiken für Sprachdefinitionen



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

# Chomsky-Grammatik



## Definition: Chomsky-Grammatik

Eine Chomsky-Grammatik hat die Form $G =< \Sigma, N, P, S >$ mit:

- $\Sigma$ endlich (Terminalsymbole, Alphabet)
- $N$ endlich, $N$ und $\Sigma$ 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 $x \to y$

# Chomsky-Grammatik (ii)



## Beispiel

$G_{\text{bin}} =< \Sigma_{\text{bin}}, N, P, S >$ mit:

- $N = \{\text{BinZahl}, \text{BinRest}\}$
- $S = \text{BinZahl}$
- $P$ enthält die Produktionsregeln:

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 (= Anwendung der Regeln)



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

- $u \vdash v :\Leftrightarrow \exists x, y, y', z \in (N \cup \Sigma)^*, u = xyz, v = xy'z, y \to y' \in P$
  $u$ ist in einem Schritt (direkt) ableitbar nach $v$.
- $u \overset{n}{\vdash} v :\Leftrightarrow \exists u_0, ..., u_n \in (N \cup \Sigma)^*, u = u_0 \vdash ... \vdash u_n = v$
  Die Folge $u_0 \vdash ... \vdash u_n$ heißt Ableitung (der Länge n)
- $u \overset{*}{\vdash} v \Leftrightarrow \exists n \geq 0 : u \overset{n}{\vdash} v$
- $u \in (N \cup \Sigma)^*$ ist ableitbar in $G =< \Sigma, N, P, S >:\Leftrightarrow S \overset{*}{\vdash} u$
- $L(G) := \left\{ w \in \Sigma^* \mid S \overset{*}{\vdash} w \right\}$ ist die von $G$ erzeugte Sprache
- Einige mögliche Ableitungen in $G_{\text{bin}}$: BinZahl $\vdash$ 1 BinRest $\vdash$ 10 BinRest $\vdash$ 100
- $L(G_{\text{bin}})$ = $\{0\}$ $\cup$ Menge der 0-1-Wörter ohne führende 0

# Beispiel 2



$G_{\text{dez}} =< \Sigma_{\text{des}}, N, P, S >$ mit

- $N$ = {DecimalNumeral, Digits, Digit, NonZeroDigit}
- $S$ = DecimalNumeral
- $P$ enthält die Regeln:

1. DecimalNumeral $\rightarrow$ 0
2. DecimalNumeral $\rightarrow$ NonZeroDigit
3. DecimalNumeral $\rightarrow$ NonZeroDigit Digits
4. Digits $\rightarrow$ Digit
5. Digits $\rightarrow$ Digits Digit
6. Digit $\rightarrow$ 0
7. Digit $\rightarrow$ NonZeroDigit
8. NonZeroDigit $\rightarrow$ 1 (analog für 2,...,9)

- Mögliche Ableitung: DecimalNumeral $\vdash$ NonZeroDigit Digits $\vdash$ 1 Digits $\vdash$ 1 Digits Digit $\vdash$ 1 Digit Digit $\vdash$ 10 Digit $\vdash$ 10 NonZeroDigit $\vdash$ 105
- $L(G_{\text{dez}}) = \{0\} \cup$ Menge der Dezimaldarstellungen natürlicher Zahlen ohne führende 0

# Chomsky-Hierarchie



Eine Grammatik $G =< \Sigma, N, P, S >$ 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 $\alpha \to \beta$ zusätzlich gilt: $|\alpha| \leq |\beta|$
- Typ 2 oder kontextfrei, falls $P \subseteq N \times (N \cup \Sigma)^*$
- Typ 3 oder regulär, falls
  - $P \subseteq N \times (N\Sigma \cup \Sigma \cup \{\varepsilon\})$ (linkslinear)
  - $P \subseteq N \times (\Sigma N \cup \Sigma \cup \{\varepsilon\})$ (rechtslinear)
- Beispiele zu Regeln für Typ i Grammatiken:
  - Typ 3: Binzahl $\to$ 0, Binzahl $\to$ 1, Binzahl $\to$ 1A, A $\to$ 0A, A $\to$ 1A, A $\to$ 0, A $\to$ 1
  - Typ 2: DecimalNumeral $\to$ 0, DecimalNumeral $\to$ NonZeroDigit, ...
- Ab jetzt für unsere Zwecke Konzentration **ausschließlich** auf kontextfreie und reguläre Grammatiken

# Zwischenstand



- Formale Sprachen sind eine Menge von Wörtern (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.

## Reflexion

- Finden Sie zur Grammatik $G_{\text{dez}}$ 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 in dieser Darstellung 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 $G_{\text{bin}}$ in BNF:

$< \text{Binzahl} > ::= 0 \mid 1 \mid 1 < \text{BinRest} >$

$< \text{BinRest} > ::= 0 \mid 1 \mid 0 < \text{BinRest} > \mid 1 < \text{BinRest} >$

Produktionen von $G_{\text{dez}}$ in BNF:

$< \text{DecimalNumeral} > ::= 0 \mid < \text{NonZeroDigit} > \mid < \text{NonZeroDigit} >< \text{Digits} >$

$< \text{Digits} > ::= < \text{Digit} > \mid < \text{Digits} >< \text{Digit} >$

$< \text{Digit} > ::= 0 \mid < \text{NonZeroDigit} >$

$< \text{NonZeroDigit} > ::= 1 \mid 2 \mid ... \mid 9$

# Extended Backus-Naur Form



- Die Regel $< \mathrm{BinRest} > ::= 0 \mid 1 \mid 0 < \mathrm{BinRest} > \mid 1 < \mathrm{BinRest} >$ besagt eigentlich, dass $< \mathrm{BinRest} >$ 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 $G_{\text{bin}}$ in EBNF:

$< \text{Binzahl} > ::= 0 \mid 1 \mid 1 < \text{BinRest} >$

$< \text{BinRest} > ::= (0 \mid 1)\{0 \mid 1\}$

Produktionen von $G_{\text{dez}}$ in EBNF:

$< \text{DecimalNumeral} > ::= 0 \mid < \text{NonZeroDigit} > [< \text{Digits} >]$

$< \text{Digits} > ::= [< \text{Digits} >] < \text{Digit} >$

$< \text{Digit} > ::= 0 \mid < \text{NonZeroDigit} >$

$< \text{NonZeroDigit} > ::= 1 \mid 2 \mid ... \mid 9$

# Syntaxdiagramme



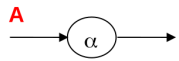
- BNF und EBNF sind entwickelt worden zur einfachen maschinellen Verarbeitung (und nicht in erster Linie für Menschen)
- Syntaxdiagramme sind entwickelt worden, um einem Menschen zum Beispiel in einem Buch zu einer Programmiersprache einfach die Syntax zugänglich zu machen
- Syntaxdiagramme sind aufgebaut:
  - 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 → Syntaxdiagramm

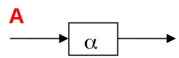
Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

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

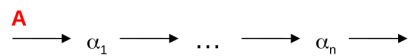

Falls $\alpha$ Terminalsymbol



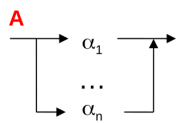

Falls $\alpha$ Nichtterminalsymbol



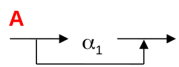

Falls $\alpha = \alpha_1 \ldots \alpha_n$



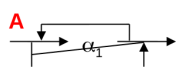

Falls $\alpha = \alpha_1 \mid \ldots \mid \alpha_n$




Falls $\alpha = [\ \alpha_1\ ]$



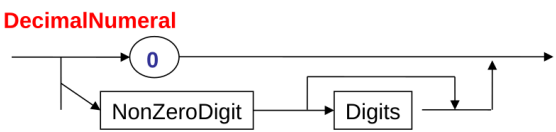

Falls $\alpha = \{\ \alpha_1\ \}$

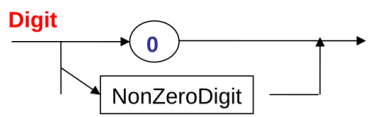


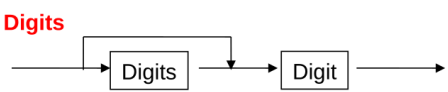
# Beispiel

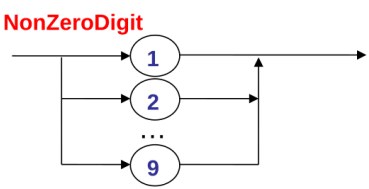


- Zur Erinnerung: Produktionen von Gdez in BNF:
  - $< \text{DecimalNumeral} > ::= 0 \mid < \text{NonZeroDigit} > [< \text{Digits} >]$
  - $< \text{Digits} > ::= [< \text{Digits} >] < \text{Digit} >$
  - $< \text{Digit} > ::= 0 \mid < \text{NonZeroDigit} >$
  - $< \text{NonZeroDigit} > ::= 1 \mid 2 \mid \ldots \mid 9$



# Zwischenstand



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

## Reflexion

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

# Literatur



, P. (2019, Juli 21). *History of programming languages*. https://javaconceptoftheday.com/history-of-programming-languages/