Kapitel 02: Einführung in Formale Sprachen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

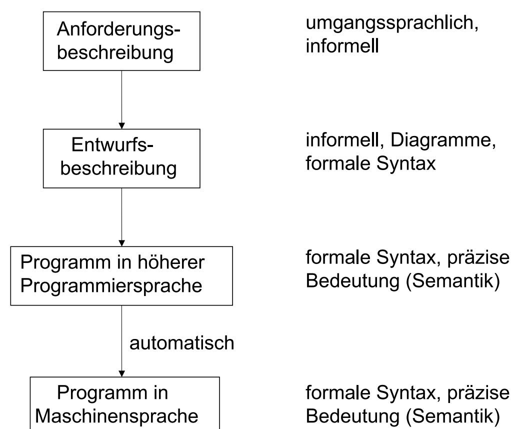
In diesem Abschnitt legen wir die Grundlage für die alle weiteren Vorlesungen. Wir werden sehen, wie die Syntax (d.h. die Form) und die Semantik (d.h. die Bedeutung) von Programmiersprachen definiert werden kann.

Lernziele dieses Kapitels:

- Zwischen informalen und formalen Sprachen zu unterscheiden.
- Die Syntax von Programmiersprachen und anderen formalen Sprachen mittels Syntaxdiagrammen und kontextfreien Grammatiken zu spezifizieren.
- Ableitungen für Worte einer formalen Sprache zu erstellen.
- Syntaktische und semantische Aspekte voneinander zu trennen.

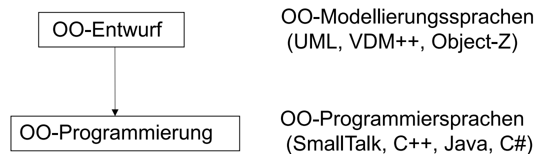
1 Was ist überhaupt eine Programmiersprache?

In der Software-Entwicklung gibt es verschiedene *Beschreibungsebenen*, um Programme und deren Bedeutung zu beschreiben.



Die Programmiersprache ist das Kommunikationsmedium, mit dem der Mensch der Maschine mitteilt, was sie tun soll. Letztendlich muss die Beschreibung eines Softwaresystems also so weit verfeinert werden, dass sie mit den Mitteln einer Programmiersprache ausgedrückt werden kann. Eine enge Beziehung zwischen Entwurf und Programmierung, beispielsweise durch ähnliche Konzepte und Modelle, erleichtern diesen Prozess.

Beispiel Wir werden im späteren Verlauf der Vorlesung mit Java eine objekt-orientierte Programmiersprache kennenlernen. Für objekt-orientierte Programmiersprachen gibt es entsprechende objekt-orientierte Modellierungssprachen.



In der Vorlesung SE1 stehen zunächst die Konzepte und Modelle der Programmierung im Vordergrund. Diese Konzepte sind *programmiersprachenunabhängig*. Wir erläutern sie anhand ihrer Ausprägungen in den Sprachen Java und C.
In der weiterführenden Vorlesung SE2 geht es dann verstärkt um Entwurfskonzepte und -modelle.

1.1 Charakterisierung von Programmiersprachen

Im Folgenden wollen wir Programmiersprachen genauer charakterisieren. Betrachten wir zunächst Beispiele von Programmen in verschiedenen Programmiersprachen. Diese Beispiele zeigen alle die Implementierung eines Suchalgorithmus'.

Programmbeispiel: Haskell

```
qsort1 :: Ord a => [a] -> [a]
qsort1 []     = []
qsort1 (p:xs) = qsort1 lesser ++ [p] ++ qsort1 greater
where
lesser  = filter (< p) xs
greater = filter (>= p) xs
```

Programmbeispiel: Python

```
def qs(l):
if l==[]: return []
m = l[0]
return qs([x for x in l if x<m]) + \
[x for x in l if x==m] + \
qs([x for x in l if x>m])
```

Programmbeispiel: Java

```
void quicksort (int[] a, int lo, int hi) {
  int i=lo, j=hi, h;
  int x=a[(lo+hi)/2];
```

```
do {
  while (a[i]<x) i++;
  while (a[j]>x) j--;
  if (i<=j)
  {
    h=a[i]; a[i]=a[j]; a[j]=h;
    i++; j--;
  }
  } while (i<=j);
  if (lo<j) quicksort(a, lo, j);
  if (i<hi) quicksort(a, i, hi);
}
```

Programmbeispiel: Erlang

```
qsort([]) -> [];
qsort([Pivot|T]) ->
qsort([X || X <- T, X < Pivot])
++ [Pivot] ++
qsort([X || X <- T, X >= Pivot]).
```

Programmbeispiel: Lua

```
function quicksort(t, start, endi)
start, endi = start or 1, endi or #t
--partition w.r.t. first element
if(endi - start < 1) then return t end
local pivot = start
for i = start + 1, endi do
if t[i] <= t[pivot] then
if i == pivot + 1 then
t[pivot],t[pivot+1] = t[pivot+1],t[pivot]
else
t[pivot],t[pivot+1],t[i] = t[i],t[pivot],t[pivot+1]
end
pivot = pivot + 1
end
end
t = quicksort(t, start, pivot - 1)
return quicksort(t, pivot + 1, endi)
end
```

Programme sind zunächst einfache Texte, d.h. Folgen von Zeichen. Programmiersprachen nutzen zur besseren Lesbarkeit häufig Elemente natürlicher Sprache als Schlüsselwörter integrieren (z.B. `while`, `if`, `stop`, `begin`, `end`). Programme sind, je nach Sprache, außerdem durch weitere Elemente strukturiert (Kommas, Punkte, Semikolons, Klammern). Die Struktur von Programmen folgt bestimmten Regeln. Beispielsweise gibt es für jede Klammer `(` eine entsprechende schließende Klammer `)`; ebenso für andere Arten von Klammern etc. Auch die gültige Position von Schlüsselwörtern ist festgelegt. Zwischen Programmiersprachen und Maschinensprache gibt es oft weitere Sprachebenen:

- Assembler

- Abstrakte Maschinensprachen, Bytecode

Das folgende Programm ist in der Programmiersprache C geschrieben. [1]

```
char a[3], b[3];
int i;
char res;
int main() {
  i = 2;
  res = 1;
  while( -1 < i ) {
    if( res ) {
      res = (a[i]==b[i]);
      i = i-1;
      } else {
        i = i-1;
      }
    }
    return res;
  }
```

In Abbildung 1 findet sich das gleiche Programm in einer maschinennahen Sprache namens Assembler. Das Programm in C ist rechner-/plattformunabhängig (→ portabel). Die Sprachelemente des Assembler-Codes entsprechen den Befehlen (engl. *Opcodes* oder *operation codes*) und Speichermechanismen des zugehörigen Rechners. Maschinenprogramme sind Byte- bzw. Zahlenfolgen, diese sich aus einer einfachen Umwandlung von Assembler-Codes in eine maschinenlesbare Repräsentation ergeben. Das Programm ist daher von einem entsprechenden Rechner direkt ausführbar.

2 Formale Sprachen

Beschreibungstechniken spielen eine besondere Rolle in der Informatik. Sie sind *Hilfsmittel* und *Untersuchungsgegenstand* (vgl. Modul "Formale Grundlagen der Programmierung") der Informatik.
Formale Sprachen eignen sich zur (mathematisch) präzisen Beschreibung von Text bzw. Zeichenketten. Mit Hilfe von formalen Sprachen können wir die Syntax von Programmiersprachen exakt spezifizieren.
Die *Spezifikation* einer formalen Sprache S sollte es ermöglichen zu prüfen, ob eine Zeichenreihe w Element von S ist. Mit Hilfe der Spezifikation können wir also feststellen, ob ein Text ein syntaktisch gültiges Programm in einer Programmiersprache ist. Dazu führen wir zunächst einige grundlegende Begriffe ein.

Definition Ein ***Zeichen*** oder ***Symbol*** ist ein abstraktes Element, das eine *syntaktisch unmissverständliche Repräsentation* besitzt.

Typische Repräsentationen von Zeichen/Symbolen sind Ziffern, Buchstaben, das Euro-Zeichen, Klammern, Komma, Punkt, Plus-, Minus-Zeichen, Leerzeichen, Tabulatorzeichen, Zeilenumbruchzeichen, ... Die Repräsentation kann aber auch zusammengesetzt sein (z.B. `>=` für größer gleich).

[1] Das Programm testet, ob sich Array `a` und `b` elementweise gleichen.

```
main:
addi  $sp, $sp, -12       # make space for the variables
li    $t1, 2
sw    $t1, 0($sp)         # i = 2
li    $t1, 1
sb    $t1, 4($sp)         # set res at sp +4
l1:
lw    $t2, 0($sp)         # load i into $t2
li    $t3, -1             # load -1 into $t3
slt   $t0, $t3, $t2       # -1 < i ?
beq   $t0, $zero, l6      # if not -1 < i goto exit
lb    $t1, 4($sp)         # load res from stack into $t1
beq   $t1, $zero, l4      # if res == 0 goto else if
add   $t4, $sp, 5         # base address of array a
add   $t4, $t4, $t2       # add offset/ array index
lb    $t0, 0($t4)         # load a[i]
add   $t4, $sp, 8         # base address of array b
add   $t4, $t4, $t2       # add offset/ array index
lb    $t1, 0($t4)         # load b[i]
beq   $t0, $t1, l2        # if a[i] == b[i]
sb    $zero, 4($sp)       # set res to 0
j     l3
l2:
addi  $t3, $zero, 1       # $t3 = 1
sb    $t3, 4($sp)         # res = $t3
l3:
subi  $t2, $t2, 1         # i = i-1
sw    $t2, 0($sp)         # store i to $sp +4
j    l5                  # goto end of if statement
l4:
subi  $t2, $t2, 1         # i = i-1
sw    $t2, 0($sp)         # store i to $sp +4
l5:
j    l1                  # return to loop
l6:
lw    $a0, 4($sp)         # terminate with exit code res
addi  $sp, $sp, 12        # reset stack pointer
li    $v0, 17
syscall
```

Abbildung 1: Beispiel eines Assembler-Programms. Der Text nach dem #-Zeichen sind Kommentare.

Definition Eine ***Zeichenreihe*** (***Wort***, ***String***, ***Text***) ist eine endliche Folge bzw. Sequenz von Zeichen.

Die syntaktische Repräsentation einer Zeichenreihe ergibt sich durch Aneinanderreihen ihrer Zeichen. Dabei werden keine Zwischenräume hinzugefügt. Zeichenreihen werden oft eingeschlossen in hochgestellte Anführungszeichen, um sie von dem sie umgebenden Text abzutrennen.
Die leere Zeichenreihe wird als ε oder "" notiert.

> **Frage 1:** Was ist der Unterschied zwischen "" und " "?

Definition Ein ***Alphabet*** A ist eine endliche Menge von Zeichen. Die *Menge der Zeichenreihen über* A (mit Zeichen aus A) wird mit A^* bezeichnet.
Eine ***formale Sprache*** *über* A ist eine Menge von Zeichenreihen über A, also eine Teilmenge von A^*.

In manchen Situationen betrachtet man auch formale Sprachen mit unendlichen Zeichenfolgen.
Elemente formaler Sprachen haben zunächst einmal keine Bedeutung (vgl. Abschnitt 3). Wir betrachten hier also nur die Struktur.

Beispiele für Alphabete

- $\{\ a,\ b,\ c,\ d\ \}$
- $\{\ 0, 1, 2, 3, 4, 5, 6, 7, 8, +, ^*, (,), \pi, =\ \}$
- 8-Bit ASCII-Zeichensatz bestehend aus
  - Steuerzeichen
  - Groß- und Kleinbuchstaben
  - Ziffern
  - Satzzeichen, Klammern, arithmetische Operatoren
  - Leerzeichen, Prozentzeichen, ...
- { `if`, ⟨ Ausdruck ⟩, `then`, `else`, `end` }, wobei die Symbole wie `if` aus Zeichen eines anderen Alphabet zusammengesetzt sind.

Beispiel für Formale Sprachen

- {"aa", "abba", "babbab", "c"}
- Menge aller Zeichenreihen aus großen Buchstaben, die ein ***Palindrom*** bilden; z.B.: "RELIEFPFEILER"
- Menge aller Zeichenreichen bestehend aus Ziffern, die eine Primzahl repräsentieren; z.B.: "3", "007", "251", "2324567889944433379"
- Menge der Zeichenreihen bestehend aus Textzeichen des UTF-8 Zeichensatzes, welche Java-Programme repräsentieren

2.1 Spezifikation formaler Sprachen

Endliche formale Sprachen, wie {“aa“, “abba“, “babbab“, “c“} im obigen Beispiel, kann man durch Aufzählung/Aufschreiben der einzelnen Worte *spezifizieren*. Programmiersprachen erlauben es aber normalerweise unendlich viele, z.T. sehr komplexe Programme zu schreiben, die man nicht alle aufzählen kann. Von Interesse sind daher die Regeln, die festlegen, wie man Worte bzw. Texte konstruieren kann.
Zur Spezifikation solcher unendlicher formaler Sprachen betrachten wir im Folgenden zwei Herangehensweisen:

- Syntaxdiagramme und
- kontextfreie Grammatiken.

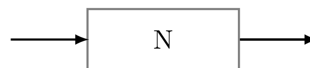
Mit kontextfreien Grammatiken lassen sich genau die Sprachen definieren, die man auch mit Syntaxdiagrammen definieren kann, d.h. beide Formalismen sind gleich mächtig.

2.1.1 Sprachdefinition mit Syntaxdiagrammen

Ein ***Syntaxdiagramm*** ist ein Flussgraph mit einem Eingang und einem Ausgang.
Die grundlegenden Syntaxdiagramme sind:

- ***Terminalknoten*** (Beschreibt die einelementige Wortmenge bestehend aus dem Wort, welches nur das genannte Terminalsymbol enthält):

- ***Nichtterminalknoten*** (Beschreibt die Wortmenge, welches durch das Syntaxdiagramm mit dem genannten Namen beschrieben wird):



- ***Diagramm für leeres Wort*** (Beschreibt die Wortmenge, die nur das leere Wort enthält):

Diese grundlegenden Syntax-Diagramme können zu größeren zusammengesetzt werden. Wir unterscheiden dabei folgende vier Arten der Komposition:

Sequenz	
Alternative	
Option	
Wiederholung	

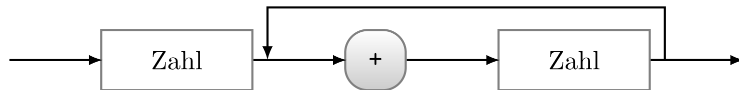
wobei jeweils ein kleineres Syntax-Diagramm repräsentiert.

Definition Seien T und N zwei disjunkte Alphabete. Die Elemente von T nennen wir ***Terminalsymbole***, die von N ***Nichtterminalsymbole***.
Sei Δ eine endliche Menge von Syntaxdiagrammen mit Namen aus N, in denen Terminalknoten mit Terminalsymbolen und Nichtterminalknoten mit Nichtterminalsymbolen markiert sind.
Sei $S \in N$; S heißt ***Startsymbol*** oder ***Axiom***.
Dann heißt $\Gamma = (N, T, \Delta, S)$ eine ***Sprachdefinition mit Syntaxdiagrammen***, kurz SDmSD.

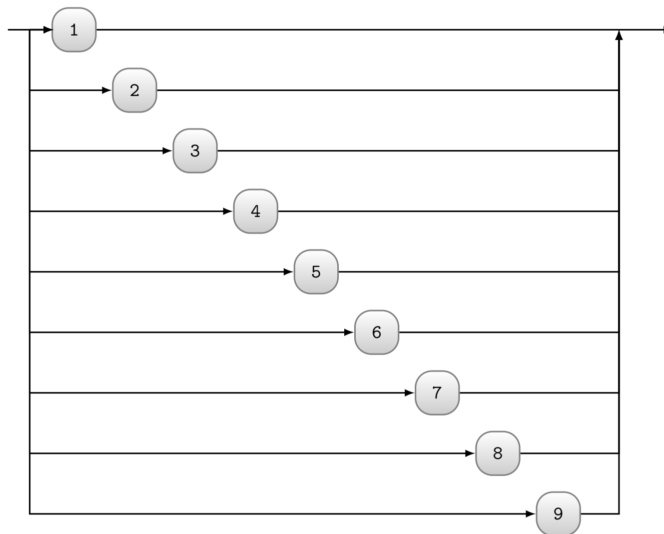
Die von einer ***Sprachdefinition mit Syntaxdiagrammen definierte Sprache*** ist die Menge der Zeichenreihen über dem Alphabet T, die man durch "Ablaufen" der Syntaxdiagramme ausgehend vom Syntaxdiagramm mit Namen S erhält.

Beispiel: Arithmetischer Ausdrücke über natürlichen Zahlen

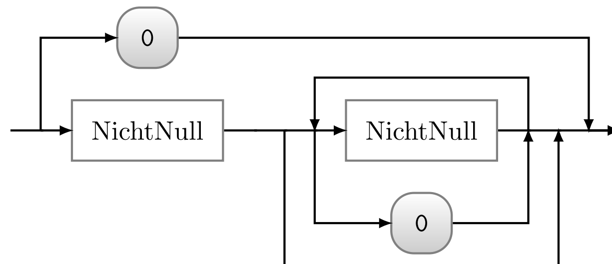
- Das Syntaxdiagramm namens *Summe* (analog Differenz, Produkt, etc.) beschreibt die (beliebig lange) Aneinanderreihung von Zahlen, getrennt durch das Plus-Symbol:



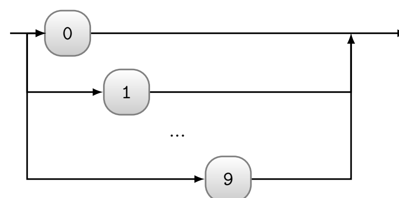
- Das Syntaxdiagramm *NichtNull* beschreibt die Alternativen für Ziffern, die ungleich Null sind:



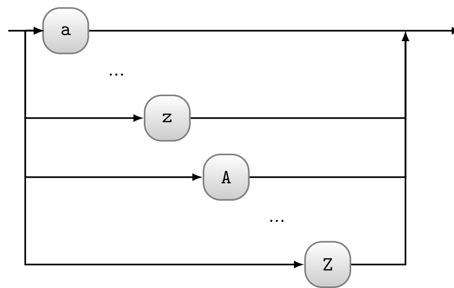
- Das Syntaxdiagramm *Zahl* beschreibt dann die Struktur natürlicher Zahlen $\mathbb{N}_0$:



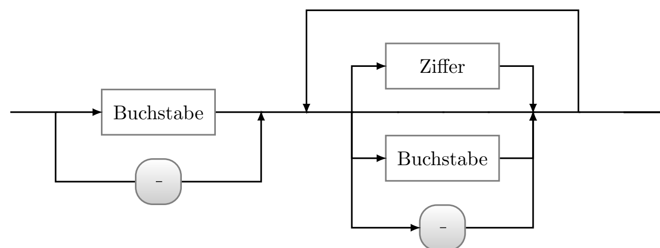
Beispiel: Bezeichner Sei $T = \{a, b, ..., z, A, ..., Z, 0, 1, ..., 9, _\}$ eine Menge von Terminalensymbolen sowie $N = \{\text{Bezeichner, Buchstabe, Ziffer}\}$ eine Menge von Nichtterminalsymbolen. Sei weiterhin Δ die Menge der folgenden drei Syntaxdiagramme *Ziffer, Buchstabe, Bezeichner.*
Ziffer:



Buchstabe:



Bezeichner:



Dann definiert (N, T, Δ, Bezeichner) eine Menge von Worten, die in Programmiersprachen häufig als Menge der zulässigen Namen/Bezeichner verwendet wird. "i36_x" und "_n" sind Elemente dieser Sprache, nicht jedoch "2gtbt" oder "+34" oder "Hallo Welt".

Beispiel: Programmiersprache "Femto" Als realistischeres Beispiel für eine Sprachdefinition betrachteten wir Vereinbarungslisten mit nachfolgender Ausgabevereinbarung. Dies beschreibt die Programme der sehr kleinen Programmiersprache Femto. Hier ein Beispiel für ein Femto-Programm:

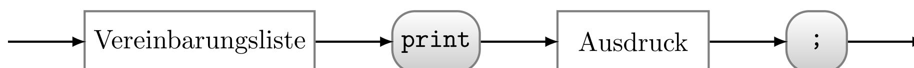
```
int a = 73;
int b = (8 * a);
print(a + b);
```

Wir wollen nun die Syntax von Femto formal spezifizieren. Dazu verwenden wir folgende Definition mit Syntaxdiagrammen:

$$T = \{\texttt{int}, \texttt{print}, \texttt{Bezeichner}, \texttt{Zahl}, (,), *, +, =, ;\}$$

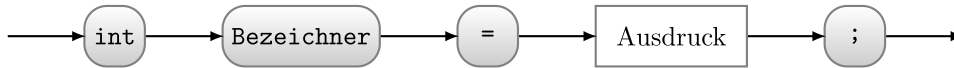
$$N = \{\mathit{Programm}, \mathit{Vereinbarungsliste}, \mathit{Wertvereinbarung}, \mathit{Ausdruck}\}$$

sowie Δ als die Menge der folgenden Syntaxdiagramme.
Programm:

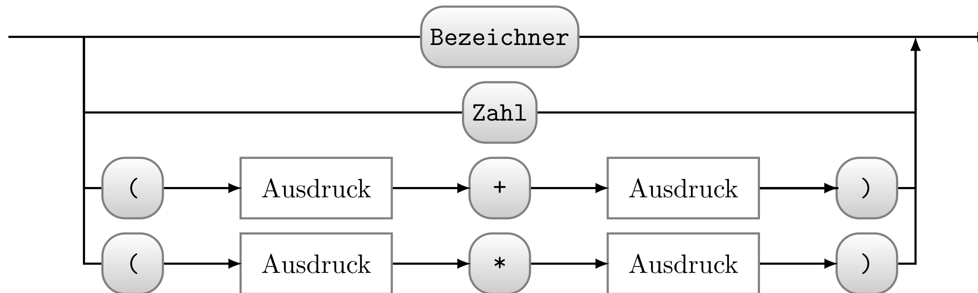


Vereinbarungsliste:

Wertvereinbarung:



Ausdruck:

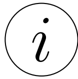


Dann beschreibt $(N, T, \Delta,$ *Programm*$)$ die Syntax der Femto-Programme.

2.1.2 Behandeln von komplexen Bezeichnern

So wie in natürlichen Sprachen Sätze aus Wörtern zusammengesetzt sind und die Wörter wiederum aus Buchstaben, sind auch die Programmtexte von Programmiersprachen aus ***Tokens*** zusammengesetzt, die wiederum aus einzelnen Symbolen (häufig Character genannt) bestehen. Für beide Ebenen der Strukturierung kann man jeweils Sprachdefinitionen angeben. Wir haben die Sprache Femto daher mit Terminalsymbolen wie *Zahl* und *Bezeichner* beschrieben, die selbst aus Text-Zeichen bestehen können. Für eine komplette Beschreibung der Syntax müssen wir noch angeben, wie wir von einem Eingabetext zu den Terminalsymbolen der Grammatik (Tokens) kommen. Dies behandeln wir hier nur am Rande.
Für Femto gelten die folgenden Vereinbarungen:

1. Leerzeichen und Zeilenumbrüche werden nicht berücksichtigt
2. Die Terminalsymbole `Bezeichner` und `Zahl` sind so aufgebaut, wie zuvor schon mit Syntaxdiagrammen definiert.
3. Die Schlüsselwörter `int` und `print` ergeben sich aus den jeweiligen Zeichenfolgen und sind keine zulässigen Bezeichner.
4. Die restlichen Terminalsymbole bestehen nur aus einem einzelnen Textzeichen.



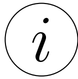
In der Praxis wird das Aufteilen des Eingabetexts in passende Terminalsymbole Lexen oder Scannen genannt. Dazu werden die einzelnen Terminalsymbole und ignorierte Textteile wie Leerzeichen oder Kommentare meist durch *reguläre Ausdrücke* beschrieben. Dieses Thema wird in der Vorlesung "Übersetzer und sprachverarbeitende Werkzeuge" genauer behandelt. Bei Interesse finden Sie weitere Informationen auch in A. Appel: Modern Compiler Implementation in Java, Cambridge University Press, 2002 (2nd edition), INF 466/120.

Beispiel: Das folgende Beispiel zeigt den Programmtext für ein Femto-Programm gefolgt von den daraus resultierenden Terminalsymbolen.

```
int a = 73;
int b = (8 * a);
print(a + b);
```

2.1.3 Sprachdefinition mit kontextfreien Grammatiken

Definition Seien T und N zwei disjunkte Alphabete. Die Elemente von T heißen ***Terminalsymbole***, die von N ***Nichtterminalsymbole***.
Sei Π eine endliche Teilmenge von $N \times (N \cup T)^*$. Die Elemente von Π heißen ***Produktionen*** (***Regeln***). Produktionen werden in der Form $A \to \alpha$ notiert, wobei A ein Nichtterminalsymbol ist und α eine möglicherweise leere Folge von Terminal- und Nichtterminalsymbolen. Wir interpretieren Produktionen als Ersetzungsregeln, wobei wir die linke durch die rechte Seite ersetzen (siehe Abschnitt 2.1.4). Abkürzend steht $A \to \alpha \mid \beta \mid \gamma \mid \dots$ für die Regeln $A \to \alpha$, $A \to \beta$, $A \to \gamma$, $A \to \dots$
Sei $S \in N$; S heißt ***Startsymbol*** oder *Axiom*.
Dann heißt $\Gamma = (N, T, \Pi, S)$ eine ***kontextfreie Grammatik***, kurz ***kfG***.



- Die Definition erlaubt es auch “problematische” Grammatiken zu formulieren. Sie verlangt z.B. nicht, dass für jedes Nichtterminalsymbol eine Produktion definiert werden muss. Grammatiken, die weitere Einschränkungen verlangen, werden in späteren Vorlesungen (FGdP, Compilerbau, etc.) behandelt.
- Die Abbildung L : Grammatik $\rightarrow$ Sprache ist im Allgemeinen nicht injektiv; d.h. zu einer Sprache gibt es im Allgemeinen mehrere erzeugende Grammatiken.

Beispiele für kontextfreie Grammatiken Analog zu der Definition mit Syntaxdiagrammen können wir auch eine kontextfreie Grammatik für Bezeichner in Programmiersprachen angeben:

$$
\begin{array}{lcll}
T & = & \{ & a, b, ..., z, A, ..., Z, 0, 1, ..., 9, \text{'_'}\} \\
N & = & \{ & \text{Bezeichner, Buchstabe, Ziffer, BezeichnerRest, BustaUnstri } \} \\
\Pi & = & \{ & \text{Ziffer} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \text{ ,} \\
 & & & \text{Buchstabe} \rightarrow a \mid b \mid \ \ldots \ \mid z \mid A \mid \ \ldots \ \mid Z \text{ ,} \\
 & & & \text{Bezeichner} \rightarrow \text{BustaUnstri BezeichnerRest ,} \\
 & & & \text{BustaUnstri} \rightarrow \text{Buchstabe} \mid \text{'_' ,} \\
 & & & \text{BezeichnerRest} \rightarrow \varepsilon \\
 & & & \qquad \mid \text{Ziffer BezeicherRest} \\
 & & & \qquad \mid \text{BustaUnstri BezeichnerRest} \\
 & & \} &
\end{array}
$$

Das Zeichen ϵ (der griechische Buchstabe “Epsilon”) ist kein Terminalsymbol, sondern steht für das leere Wort und verdeutlicht hier nur eine leere Produktion.
Wenn Terminalsymbole verwendet werden, die keine Buchstaben oder Ziffern sind, sollte das entsprechend kenntlich gemacht werden, z.B. mit Anführungszeichen um das Terminalsymbol.

Frage 2:

- Vergleichen Sie die Definition mit Syntaxdiagramm und kontextfreier Grammatik! In welchem Zusammenhang stehen die Produktionen und die Grundtypen der Syntaxdiagramme?

Wir verwenden im Folgenden zur Illustration von Begriffen oft die Grammatik für balancierte Klammerausdrücke, die aus den folgenden Symbolen und Regeln besteht:

$$
\begin{array}{lcl}
T & = & \{ \text{'(', ')'} \} \\
N & = & \{S\} \\
\Pi & = & \{S \rightarrow S\ S \mid \text{'('}\ S\ \text{')'} \mid \varepsilon\}
\end{array}
$$

Nichtterminal S ist Startsymbol.
Ein Wort dieser Sprache ist z.B. der Ausdruck $(()(()()))$. Aber wie können wir dies nachweisen?

2.1.4 Ableitungen

Für die folgenden Definitionen sei eine kontextfreie Grammatik $\Gamma = (N, T, \Pi, S)$ gegeben. Außerdem seien

- $A, B, C, \ \ldots$ aus N,
- $a, b, c, \ \ldots$ aus T,
- $x, y, z, \ \ldots$ aus T^* und
- $\alpha, \beta, \gamma, \psi, \varphi, \sigma, \tau, \ \ldots$ aus $(N \cup T)^*$.

- ψ ist mit Γ aus φ ***direkt ableitbar*** ($\varphi \Rightarrow \psi$), wenn es eine Zerlegungen $\sigma A \tau$ von φ und $\sigma \alpha \tau$ von ψ gibt und $A \rightarrow \alpha$ eine Produktion in Π ist.

  Beispiel: $\psi = ((SS)(S))$ ist aus $\varphi = ((SS)S)$ direkt ableitbar.

  Betrachte folgende Zerlegung: $\varphi = \sigma\ S\ \tau$ mit $\sigma =$ '(''('SS')' und $\tau =$ ')'.
  Da die Regel $S \rightarrow$ '(' S ')' eine Produktion in Π ist, können wir das S zwischen σ und τ durch die rechte Seite der Produktion, also durch '(' S ')', ersetzen und erhalten ψ.

- ψ ist mit Γ aus φ ***ableitbar***, ($\varphi \Rightarrow^* \psi$), wenn es $\varphi_0, \ldots, \varphi_n$ gibt mit $\varphi = \varphi_0$, $\varphi_n = \psi$ und für alle $i \in \{0, \ldots, n-1\} : \varphi_i \Rightarrow \varphi_{i+1}$

  $\varphi_0 \ldots \varphi_n$ heißt dann ***Ableitung*** von ψ aus φ.

  Beispiel: $\psi = (S()((S)))$ ist aus $\varphi = (S)$ ableitbar.

  Betrachte z.B. folgende Ableitung:

  $(S) \Rightarrow (SS) \Rightarrow (SSS) \Rightarrow (S(S)S) \Rightarrow (S()S) \Rightarrow (S()(S)) \Rightarrow (S()((S)))$

- Eine Ableitung $\varphi_0 \ldots \varphi_n$ heißt ***Linksableitung*** (bzw. ***Rechtsableitung***), wenn in φ_i jeweils nur das am weitesten links (bzw. rechts) stehende Nichtterminal ersetzt wird.

  Linksableitungsschritte werden als $\varphi \underset{lm}{\Rightarrow} \psi$, Rechtsableitungsschritte als $\varphi \underset{rm}{\Rightarrow} \psi$ notiert. lm steht dabei für leftmost und rm für rightmost.

  Beispiel:
  Eine Linksableitung von $\psi = (()(()))$ aus $\varphi = (S)$ ist:

  $(S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()((S))) \Rightarrow (()(()))$

  Eine Rechtsableitung von $\psi = (()(()))$ aus $\varphi = (S)$ ist:

  $(S) \Rightarrow (SS) \Rightarrow (S(S)) \Rightarrow (S((S))) \Rightarrow (S(())) \Rightarrow ((S)(())) \Rightarrow (()(()))$

- $L(\Gamma) = \{z \in T^* \mid S \Rightarrow^* z\}$ heißt die von Γ *erzeugte* ***Sprache***.

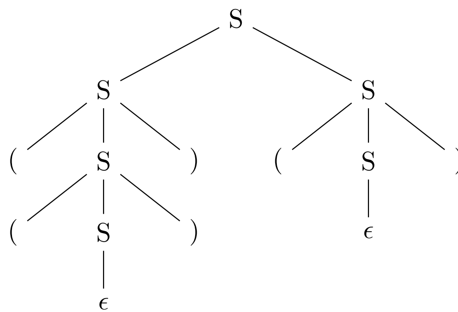
- $x \in L(\Gamma)$ heißt ein ***Satz*** von Γ.
- $\psi \in (N \cup T)^*$ mit $S \Rightarrow^* \psi$ heißt eine ***Satzform*** von Γ.

Die baumartige Darstellung einer Ableitung nennt man ***Syntaxbaum*** (***Struktur-***, ***Ableitungsbaum***).

Beispiel: Syntaxbaum für eine Ableitung Für die Ableitung

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$$

ergibt sich folgender Syntaxbaum:



> **Frage 3:** Was ist der Syntaxbaum für die folgende Ableitung?
>
> $$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$

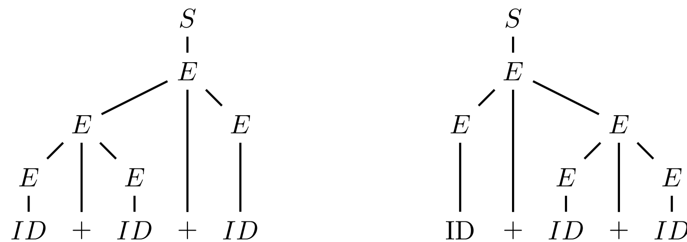
2.2 Mehrdeutige Grammatik

- Ein Satz heißt ***mehrdeutig***, wenn er mehr als einen Syntaxbaum besitzt.
- Eine Grammatik heißt ***mehrdeutig***, wenn sie (mind.) einen mehrdeutigen Satz besitzt; andernfalls ist sie ***eindeutig***.

Die Grammatik Γ_0 für Ausdrücke mit den Produktionen

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E + E \ \mid \ E * E \ \mid \ (E) \ \mid \ ID \end{array}$$

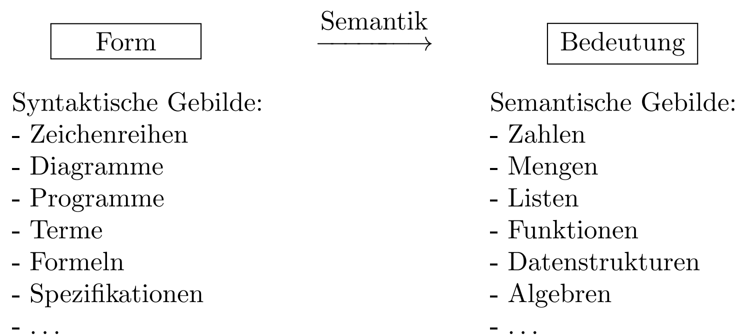
ist ein Beispiel für eine mehrdeutige Grammatik, denn für den Ausdruck $ID+ID+ID$ gibt es zwei verschiedene Syntaxbäume:



Jeder Ableitung entspricht dabei genau ein Syntaxbaum. Umgekehrt kann es zu einem Syntaxbaum mehrere Ableitungen geben. Man kann beweisen, dass ein Satz genau dann eindeutig ist, wenn er *genau eine* Linksableitung (bzw. Rechtsableitung) besitzt. Bei Programmiersprachen spielen *eindeutigen* Grammatiken eine zentrale Rolle, da die Semantik (und Übersetzung) der Sprache über die syntaktische Struktur definiert wird.

3 Syntax vs. Semantik

Die ***Syntax*** einer formalen Sprache gibt an, wie die Elemente der Sprache (Sätze, Worte, Diagramme) zusammengesetzt sind, d.h. welche *Form* sie haben. Für sich allein genommen sind diese Elemente aber zunächst bedeutungslos. Die ***Semantik*** gibt den Elementen einer formalen Sprache eine *Bedeutung*.



Unterschiedliche Beschreibungen können die gleiche Semantik haben. So beschreiben sowohl 3 also auch 3.0 den gleichen Zahlenwert in der Mathematik.[2]
Moderne Programmier- und Spezifikationssprachen besitzen eine formale Syntax und eine wohldefinierte Semantik. Diese wird beispielsweise informell in den zugehörigen Sprachspezifikationen erläutert, in der Regel in natürlicher Sprache. Für einige Sprachen, insbesondere Sprachen, die bei der Programmierung von sicherheitskritischen Systemen zum Einsatz kommen, gibt es aber auch formale Semantiken. Eine formale

[2]Achtung: In den meisten Programmiersprachen bezeichnen diese Ausdrücke verschiedene Zahlenwerte! Wir werden auf den Unterschied zwischen Integer- und Gleitkommazahlen im Laufe der Vorlesung noch im Detail eingehen.

Semantik definiert die Bedeutung mit mathematischen Mitteln. Die Implementierung einer Sprache in Form von Übersetzern/Interpretern liefert auch eine Semantik, die allerdings oft rechnerabhängig ist.
Es gibt auch formale Sprachen ohne Semantik (z.B. die Sprache der balancierten Klammerausdrücke).

Beispiel: Informelle Semantik von Femto In Abschnitt 2.1.1 wurde die kontextfreie Syntax der Programmiersprache Femto festgelegt. Femto-Programme bestehen aus einer Vereinbarungsliste mit nachfolgender Ausgabevereinbarung. Aber was ist eigentlich die Bedeutung eines Femto-Programms?
Die Semantik weist jedem Ausdruck und dem ganzen Programm einen Wert im Bereich $[-2^{31}, 2^{31} - 1]$ zu.[3] Dieser Wert wird folgendermaßen ermittelt:

- Angewandter Bezeichner: Liefert den an den Bezeichner gebundenen Wert.
- Zahl: Liefert den von der Zahl beschriebenen Wert.
- Zusammengesetzter Ausdruck:
  - Werte die Teilausdrücke aus; die Ergebnisse seien lw und rw.
  - Ist das Operatorzeichen $'+'$, addiere lw und rw; ist das Operatorzeichen $'*'$, multipliziere lw und rw; das Ergebnis sei mit w bezeichnet.
  - Liegt w im Bereich -2^{31} bis $2^{31} - 1$, ist w das Ergebnis der Ausdrucksauswertung; andernfalls ist das Ergebnis unbestimmt.
- Wertvereinbarung: Werte den Ausdruck aus und binde das Ergebnis an den Bezeichner.
- Programm: Drucke den Wert des Ausdrucks in der Ausgabevereinbarung.

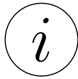
Beispielsweise weist die Semantik im folgenden Femto-Programms dem Bezeichner `a` den Wert 73 zu, dem Ausdruck `(8 * a)` und dem Bezeichner `b` den Wert 584, dem Ausdruck `(a + b)` den Wert 657 und gibt am Ende 657 aus:

```
int a = 73;
int b = (8 * a);
print(a + b);
```

Damit die Semantik von Femto wohldefiniert ist, sind weitere Einschränkungen an die Femto-Programme notwendig (***Kontextbedingungen***):

- Ein Bezeichner darf nur dann in einem Ausdruck angewendet werden, wenn er vorher vereinbart wurde.
- Die Zahlen müssen Werte im Bereich von -2^{31} bis $2^{31} - 1$ beschreiben.

[3] Der Wertebereich $[-2^{31}, 2^{31} - 1]$ ergibt sich aus der 32-Bit Binärdarstellung von ganzen Zahlen.



Die Grammatiken aus Abschnitt 2.1.3 sind kontextfrei, da die Nichtterminale an jeder verwendeten Stelle auf die gleiche Weise ersetzt werden können. Für praktisch relevante Programmiersprachen kann man normalerweise keine kontextfreie Grammatik angeben, um alle gültige Programme zu beschreiben. Beispielsweise lässt sich damit nicht ausdrücken, dass ein Bezeichner definiert sein muss, bevor er verwendet werden kann. Grammatiken, die kontextabhängige Nichtterminale enthalten, heißen ***kontextsensitiv***. Da kontextfreie Grammatiken einfacher zu verstehen und zu verwenden sind, werden sie zur Syntaxdefinition von Programmiersprachen verwendet und dann durch Kontextregeln ergänzt.

Syntaktisch inkorrekte Programme bzw. Programme, die die Kontextbedingungen verletzen, haben keine Semantik. Es ist z.B. unklar, was folgendes Programm bedeuten soll:

```
int a = print;
int b = (8 * c);
print(a+b);
```

Syntaxfehler werden vom Übersetzer bzw. Interpreter entdeckt und führen in der Regel zu einem Abbruch, da das Verhalten des Programms in diesem Fall unbestimmt ist.

4 Zur Beschreibung von Software-Systemen

Wie wir in Abschnitt 1 gesehen haben, werden bei der Entwicklung von Software-Systemen unterschiedliche Arten von Beschreibungen auf verschiedenen Ebenen eingesetzt. Dabei werden sowohl informelle als auch formale Sprachen eingesetzt und teilweise auch kombiniert. Anforderungen werden anfangs informell erhoben. Dies geschieht oft im Gespräch mit Auftraggebern, die nur wenig Erfahrung mit formalen Methoden haben und ihre Anforderungen und Ideen nur informell ausdrücken können. Dann erfolgt ein schrittweiser Übergang zu Entwurfsbeschreibungen, zu Programmen in Programmiersprachen und letztendlich zu ausführbaren Programmen in Maschinensprache.

Frage 4: Was sind die Vor- und Nachteile von informeller gegenüber formaler Beschreibung?

5 Beschreibung der Java-Syntax

In den nächsten Kapiteln dieser Vorlesung werden wir uns mit der Programmiersprache Java beschäftigen. Die Spezifikation von Java[4] verwendet die in diesem Kapitel

[4]siehe `https://docs.oracle.com/javase/specs/index.html`

beschriebenen Techniken. So beschreibt Kapitel 3 der Spezifikation ("Lexical Structure") die Beziehung zwischen Eingabe-Text und den von Java verwendeten Terminalsymbolen. Die Grammatik von Java wird dann über die folgenden Kapitel verstreut eingeführt. Kapitel 14 ("Blocks and Statements") beschreibt die verschiedenen Anweisungen in Java und Kapitel 15 ("Expressions") die Ausdrücke. Die verwendete Notation ist in der Java-Spezifikation noch um Konstrukte wie Wiederholungen und optionale Elemente erweitert, die Prinzipien sind aber die gleichen.
In den folgenden Kapiteln dieser Vorlesung werden wir nach und nach verschiedene Sprachkonstrukte von Java betrachten und jeweils auch die Syntax mit entsprechenden Produktionen einführen. Zur besseren Lesbarkeit werden wir Terminalsymbole und Nichtterminalsymbole nicht explizit definieren, sondern verwenden die folgenden Konventionen:

- Wir markieren Nichtterminalsymbole in den Produktionen durch Unterstreichen.
- Für Terminalsymbole, welche genau eine Zeichenkette repräsentieren, verwenden wir die Zeichenkette selbst.
- Andere Terminalsymbole, wie Zahlen oder Bezeichner, schreiben wir in doppelte spitze Klammern (z.B. $\ll$ *Zahl* $\gg$).

Als Beispiel für diese Notation betrachten wir hier noch einmal die Grammatik für Femto:

Programm →
Vereinbarungsliste `print` Ausdruck `;`

Vereinbarungsliste →
Wertvereinbarung Vereinbarungsliste
| ϵ

Wertvereinbarung →
`int` $\ll$ *Bezeichner* $\gg$ `=` Ausdruck `;`

Ausdruck →
$\ll$ *Bezeichner* $\gg$
| $\ll$ *Zahl* $\gg$
| `(` Ausdruck `+` Ausdruck `)`
| `(` Ausdruck `*` Ausdruck `)`