7. Syntax: Grammatiken, EBNF

Teil 1

Sehr schönes Beispiel für Notwendigkeit der Theoretischen Informatik für Belange der Praktischen Informatik

Vertiefung in:
Einführung in die Theoretische Informatik

Syntax: Grammatiken, EBNF

Information für Programmierer:
Wie muss ich mein Programm aufbauen?

Strenge Überprüfung

Information für Compiler:
Wie muss ein korrektes Programm aussehen?

Ziel: Aufbau korrekter Programme exakt festlegen

Ein Java-Programm:

```
class T1 {

   publick static main (...) {

   { x = 2 }
}
```

Was ist falsch ?

Aspekte der Korrektheit von Programmen

Lexik:
Symbole korrekt ?

Kontextfreie Syntax:
Reihenfolge der Symbole korrekt
(Struktur des Programms)?

```
class T1 {
   publick static main (...) {
   { x = 2 }
}
```

Kontextabhängige Syntax:
Symbole in die Umgebung
korrekt eingebunden?

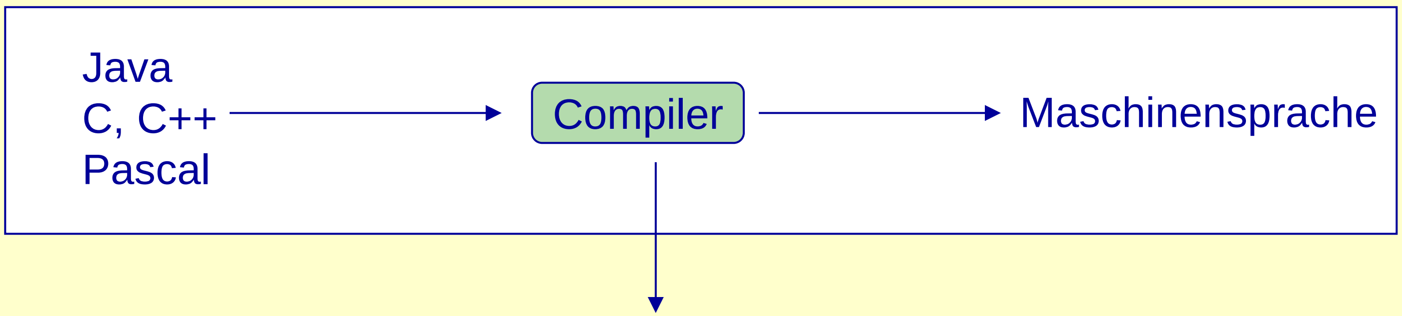
Semantik:
- Keine Laufzeitfehler?
- Abarbeitung des Programms korrekt?

Welche der Fehler kann ein Compiler erkennen?

Welche Fehler führen zu Compilationsfehlern?

Compilationsfehler = Compilerfehler?

Fehleranalyse durch Compiler-Komponenten



Compilerkomponenten:

Scanner:
Lexik
Parser:
kontextfreie Syntax
semantische Analyse:
kontextabhängige Syntax
Codegenerierung

Keine semantischen Fehler

Syntaxdefinition von Programmiersprachen

- **Kontextfreie Syntax** (+ als Voraussetzung: ***Lexik***):

  → mit **kontextfreien Grammatiken (EBNF)**

  (vollständig formalisiert)

In diesem Kapitel:
kontextfreie Syntax

- **Kontextabhängige Syntax**:

  → **verbal**

  (z. B. Jeder Bezeichner (Variable) muss vor der Benutzung vereinbart werden.)

Aspekte der Korrektheit von Programmen

Grammatiken (kontextfrei)

Lexik:
Symbole korrekt ?

Kontextfreie Syntax:
Reihenfolge der Symbole korrekt ?

```
class T1 {
   publick static main (...) {
   { x = 2 }
}
```

Kontextabhängige Syntax:
Symbole in die Umgebung korrekt eingebunden?

Semantik:
- Keine Laufzeitfehler?
- Abarbeitung des Programms korrekt?

Aufgabe (kontextfreier) Grammatiken

```
class c { int x = 1; }
```

```
class { int x:= 1 }
```

Unterscheidung korrekter Programme
von fehlerhaften Programmen

Ohne Beachtung des Kontextes -
z. B. benutzte Variablen müssen vereinbart werden

Definitionen:

- Alphabet
- Wortmenge
- Grammatik

Alphabet

Alphabet:

Endliche nicht-leere Menge A von Symbolen $a \in A$

z.B. Grundsymbole einer Programmiersprache:

A_{Java} = { **class**, **public**, **{**, **}**, **<**, **>=**, **==**, **=**, **...** }
A_{Pascal} = { **program**, **procedure**, **{**, **}**, **<**, **>=**, **=**, **:=**, **...** }

Anmerkung:
Grundsymbole
≠ Zeichen
= Folge von Zeichen
z.B. Einzelzeichen: <
Doppelzeichen: <= >=
länger: Buchstabenfolgen

Wörter über dem Alphabet

Wortmenge A* über dem Alphabet A
(= Menge von Symbolfolgen)

Induktive Definition:

- Das leere Wort ε gehört zu A* ("nichts", leere Symbolfolge).

- Die Verkettung einer Symbolfolge $x \in A^*$ mit einem Symbol $a \in A$ ergibt wieder eine Symbolfolge $xa \in A^*$.

- Weitere Elemente von A* gibt es nicht.

Anmerkung:
Verkettete Symbole bleiben aber unterscheidbar,
nicht: classpublic - sondern: class public

Beispiele

(zu Elementen aus $A_{Java}*$)

```
class c { int x = 1;}

class class c { ; x ====
```

Elemente $x \in A^*$ heißen
Wörter über dem Alphabet A

Ziel (von Grammatiken):
Unterscheidung zwischen
korrekten Symbolfolgen (Programme)
↕
fehlerhaften Symbolfolgen (keine Programme)

Kontextfreie Grammatik

G = [A, M ,s ,R] heißt **kontextfreie Grammatik**,

falls

- A Alphabet
  (Grundsymbole, terminale Symbole, Terminale)

- M Alphabet
  (Metasymbole, Nicht-Terminale, Hilfssymbole, Variablen)

  $A \cap M = \emptyset$ (disjunkte Mengen)

- $s \in M$ (Satzsymbol, Startsymbol)

- R endliche Menge von Regeln
  (Syntaxregeln, Produktionsregeln, Ersetzungsregeln)

  $R \subseteq M \times (A \cup M)^*$

Regeln

Regeln:

$R \subseteq M \times (A \cup M)^*$

d.h. Menge von Tupeln der Form (l, r) mit $l \in M$, $r \in (A \cup M)^*$

$V =_{def} (A \cup M)$ (alle Symbole)

V^*: beliebige Symbolfolgen

Beispiele: Kreuzprodukt

1. $A = \{1, 2, 3\}$
   $B = \{x, y\}$

   $A \times B = \{ (1,x), (2,x), (3,x), (1,y), (2,y), (3,y) \}$ 6 Elemente

2. $N = \{0, 1, 2, 3, \ldots\}$ (natürliche Zahlen)
   $L = \{a, b, c, \ldots z\}$ (Buchstaben)

   $N \times L = \{ (0,a), (0,b), \ldots , (0,z), (1,a), \ldots \}$ unendlich

3. Allgemein für Grammatiken:

   M – Metasymbole
   A – terminale Symbole
   beide endliche Mengen

   $M \times (A \cup M)^*$ endlich oder unendlich?

Beispiel: Grammatik zur Definition einfacher Ausdrücke

z. B. a, a + a, a * a, a + a + (a * a + a) usw.

$G_1 = [A_1, M_1, s_1, R_1]$

$A_1 = \{a, +, *, (,)\}$ **Grundsymbole, terminale Symbole, Terminale**

$M_1 = \{expr, exprrest, term, termrest, factor\}$ **Metasymbole, Nicht-Terminale, Hilfssymbole, Variablen**

$s_1 = expr$ **Satzsymbol, Startsymbol**

R_1 mit 8 Regeln:

(expr, term exprrest),	r1	(exprrest, + term exprrest),	r2
(exprrest,),	r3	(term, factor termrest),	r4
(termrest, * factor termrest),	r5	(termrest,),	r6
(factor, a),	r7	(factor, (expr))	r8

Wie weiter ?

- bisher:

  Grammatikbegriff nur formale Definition

- nächster Schritt:

  Zusammenhang:

  G → definierte (Programmier-)Sprache

  z. B. G_1 → einfache Ausdrücke

  Prinzip: Regeln werden angewendet durch
  Ersetzung der linken durch die rechte Seite einer Regel

Definitionen:

- **Ableitungen**
- **erzeugte Sprache**

Direkte Ableitung

(Ableitbarkeit in einem Schritt: Anwendung einer Regel)

Notation: $v_1 \rightarrow v_2$ $(v_1, v_2 \in V^*)$

Sprechweisen:

v_2 **direkt abgeleitet aus** v_1 oder
v_2 **in einem Schritt abgeleitet aus** v_1 oder
v_1 **erzeugt direkt** v_2

$V =_{def} (A \cup M)$ (alle Symbole)

$=_{def}$

Es gibt eine Regel $(l, r) \in R$,
wobei die linke Seite l der Regel in v_1 vorkommt:

$$\underline{v_1 = w_1\, l\, w_2} \qquad (w_1, w_2 \in V^*).$$

Wenn l in v_1 durch r ersetzt wird (Regelanwendung),
so erhält man v_2:

$$\underline{v_2 = w_1\, r\, w_2}$$

Beispiele: direkte Ableitungen mit G_1

term + <u>term</u> exprrest

→ term + <u>factor termrest</u> exprrest

wegen der Regel (term, factor termrest) (r_4)

term + term <u>exprrest</u> → term + term

wegen der Regel (exprrest,) (r_3)

<u>expr</u> → <u>term exprrest</u>

wegen der Regel (expr, term exprrest) (r_1)

vgl. nächste Folie

Beispiel: Grammatik zur Definition einfacher Ausdrücke

z. B. a, a+a, a * a, a + a + (a * a + a) usw.

$G1 = [A_1, M_1, s_1, R_1]$

A_1 = {a, +, *, (,)}
M_1 = {expr, exprrest, term, termrest, factor}
s_1 = expr

R_1 mit 8 Regeln:

(expr, term exprrest),	r1	(exprrest, + term exprrest),	r2
(exprrest,),	r3	(term, factor termrest),	r4
(termrest, * factor termrest),	r5	(termrest,),	r6
(factor, a),	r7	(factor, (expr))	r8

Allgemeine Ableitung

(Kombination von Ableitungsschritten: Mehrfache Regelanwendung)

Notation: $v_1 \rightarrow^* v_2$ ($v_1; v_2 \in V^*$)

Sprechweisen:

v_1 **erzeugt** v_2 oder

v_2 **aus** v_1 **ableitbar**

$V =_{def} (A \cup M)$ (alle Symbole)

$=_{def}$

a) $v_1 = v_2$ (identisch)

oder

b) $v_1 \rightarrow v_2$ (erzeugt direkt)

oder

c) es existieren $w_1, w_2, \ldots, w_n$ ($w_i \in V^*$, $n \geq 1$)

mit $v_1 \rightarrow w_1$, $w_1 \rightarrow w_2$, … , $w_n \rightarrow v_2$ (**Folge** direkter Ableitungen)

Kurzschreibweise:

$$v_1 \rightarrow w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_n \rightarrow v_2$$

Beispiel: Ableitung mit G_1

expr →* a + a, weil:

expr

→ term exprrest (r_1)

→ term + term exprrest

→ term + term

→ term + factor termrest

→ term + factor

→ term + a

→ factor termrest + a

→ factor + a

→ a + a

Unterstreichung, z.B.:
term: durch Regelanwendung im nächsten Schritt ersetzt

Aufgabe:
Angewendete Regeln selbst finden

Sprache

Die durch die Grammatik G erzeugte Sprache L_G:

$L_G = \{x \mid x \in A^*, s \rightarrow^* x\}$

d.h. Menge aller Wörter aus A*, die aus dem Satzsymbol ableitbar sind

$\rightarrow L_G \subseteq A^*$

Beispiel:

Beweis: letzte Folie

$L_{G1} = \{a, \underline{a + a}, a * a, a * a + a, (a + a) * a, ...\}$

Beispiel: erzeugte Sprache

G_2 = [{a, b, c, d}, {S, A, B}, S, R2]

R_2: {(S, AB), (A, a), (A, b), (B, c), (B, d)}

L_{G2} = {ac, ad, bc, bd}

S → AB → aB → ac

Wiederholung: Aufgabe (kontextfreier) Grammatiken

```
class c { int x = 1; }
```

```
class { int x:= 1 }
```

aus G_{Java} ableitbar

nicht aus G_{Java} ableitbar

Unterscheidung korrekter Programme
von fehlerhaften Programmen
(ohne Beachtung des Kontextes -
z. B. benutzte Variablen müssen vereinbart werden)

Compilerkomponente ‘Parser‘ (Syntaxanalyse):

- Gegeben: Programm P, Grammatik G = [A, M, s, R].
- Aufgabe: Entscheide, ob P syntaktisch korrekt aufgebaut.
- Methode: Bilde Ableitung von P aus s.

BNF: Backus-Naur-Form

Spezielle Festlegungen zur Notation einer Grammatik (bessere Lesbarkeit):

(1) Regel (l, r)

notiert als l ::= r .
alternativ: l = r . oder l : r . oder l → r .

(2) Regeln (l, r1), (l, r2), ... , (l, r_n) (gleiche linke Seite)

notiert als l ::= r_1 | r_2 | ... | r_n .

z. B. Ausdrucksgrammatik G_1

Alternativen von l:
r_1, r_2, ... r_n

BNF: Backus-Naur-Form für G_1

(1) Regel (l, r)

notiert als l ::= r .
alternativ: l = r . oder l : r . oder l → r .

(2) Regeln (l, r1), (l, r2), ... , (l, r_n)   (gleiche linke Seite)

notiert als   l ::= r_1 | r_2 | ... | r_n .

(expr, term exprrest), (exprrest, + term exprrest),
(exprrest,), (term, factor termrest),
(termrest, * factor termrest), (termrest,),
(factor, a), (factor, (expr))

```
expr     ::= term exprrest .
exprrest ::= “+“ term exprrest | .          (2 Regeln)
term     ::= factor termrest .
termrest ::= “*“ factor termrest | .        (2 Regeln)
factor   ::= “a“ | “(“ expr “)“ .           (2 Regeln)
```

EBNF (erweiterte BNF, Wirth)

Zusätzliche Festlegungen:

(1) [...] Option:

Folge von Symbolen, die dort stehen kann, aber nicht dort stehen muss

(2) (...) Zusammenfassung (z. B. mehrerer Varianten)

(3) { ... } Wiederholung der Folge von Symbolen (n $\geq$ 0 mal)

Beispiele

G_1 noch kürzer (Metasymbole eingespart)

```
expr     ::= term exprrest .
exprrest ::= "+" term exprrest | .          (2 Regeln)
term     ::= factor termrest .
termrest ::= "*" factor termrest | .        (2 Regeln)
factor   ::= "a" | "(" expr ")" .           (2 Regeln)
```

```
expr   ::= term { "+" term } .
term   ::= factor { "*" factor}.
factor ::= "a" | "(" expr ")" .
```

mit M = {expr, term, factor}

Exponent in Java:

```
Exponent ::= (e | E) [+ | -] Ziffer { Ziffer }
```

Zusammenfassung **Option** **Wiederholung**