

# Korrekte Klammersausdrücke

(( ( ) ( ) ) ( ) ( ) )

Korrekt heißt: • zu jeder Klammer auf "(" gehört genau eine Klammer zu ")"

Ein solches Paar bildet einen Block

- Block beginnt mit (, endet mit )
- Blöcke dürfen sich nicht echt überlappen



echt überlappend  
nicht korrekt mit dieser  
Zuordnung von Klammern



korrekt, keine echt überlappend

Korrekte Klammersausdrücke wichtig für Prog. Sprachen

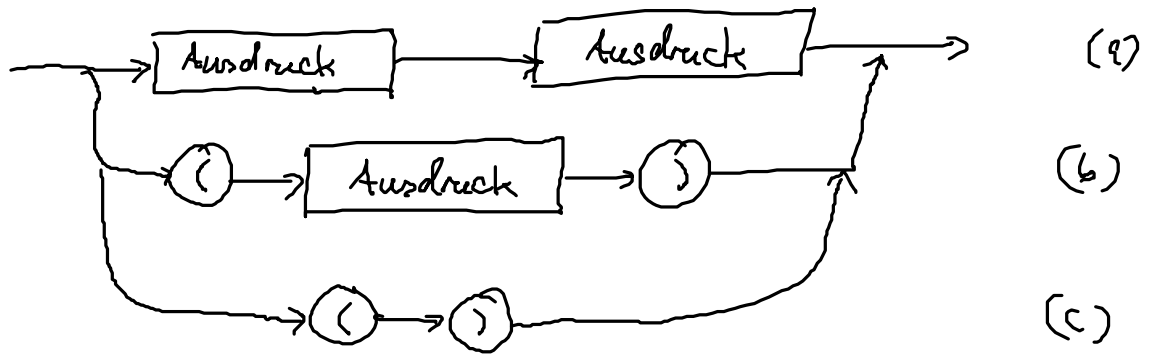
Compiler und Editoren müssen Korrektheit erkennen und die  
Zuordnung ( - ) finden

dafür muss man Bildungsgesetze kennen

↳ Grammatik

# Syntaxregeln

von letzter VL: Grammatik entwickelt



Bsp: (c) ( )

(a) ( ) , ( ( ) ) korrekt  $\Rightarrow$  ( ) ( ( ) ) korrekt

(b) ( ( ) ) korrekt  $\Rightarrow$  ( ( ( ) ) ) korrekt

SATZ: Jede korrekte Klammerausdruck mit Länge  $> 2$  entsteht durch Anwendung der Regel (a) oder (b) aus kleineren korrekten Klammerausdrücken

(Folgerung: wiederholte Anwendung dieses Satzes ergibt Aufbau des Ausdrucks)

Beweis: Sei A korrekt  $\stackrel{\text{Def}}{\Rightarrow}$   $\exists$  Zuordnung für die Blockbed. erfüllt sind

$\Rightarrow$  A muss mit ( anfangen

Betrachte die zugehörige )



Syntax läuft in Teil - C3 illegale Turmzüge

brauchen zusätzliche Semantikregeln  
für Turmzüge

könnte es teilweise auch in Syntax bringen

(berücksichtigen der Bewegung nur auf Reihe  
oder nur auf Linie)

auch dann brauchen wir noch semantische Regeln

die z.B. aussagen, dass ein Turm nicht über  
ein Feld ziehen darf, auf dem eine Figur steht

korrekte Turmzüge benötigen Kontextinformationen

Syntaxregeln (BNF, Syntaxdiagramme) sind Kontextfrei

Prog. Sprachen sind in großen Teilen Kontextfrei:

! dies sind die Teile, die durch  
Syntaxregeln beschreibbar sind

(BNF ...)

wesentliche Teile sind aber Kontextabhängig

(Groß, Kleinschreibung bei Identifizieren

Parameter Korrespondenz bei Methodenaufrufen

...

## Bedeutung von Fehlern (literale)

Compiler überprüft alle Syntaxregeln, Semantikregeln  
jedoch nur partiell

syntaktisch korrekte Ausdrücke können semantisch sinnlos  
sein ohne dass der Compiler es feststellt

Schreibe den Namen des 13. Monats im Jahr

Semantikfehler nur aufdeckbar, wenn Compiler mehr über  
Semantik weiß, aber auch dann schwer, da manche  
Semantikfehler erst zur Laufzeit auftreten

Wähle  $n$  aus  $\{1, 2, \dots, 13\}$

Schreibe den Namen des  $n$ . Monats

Neben Syntax- und Semantikfehlern gibt es  
auch die Logikfehler

Berechne den Umfang durch Multiplikation des Radius mit  $\pi$

Kann kein Compiler aufdecken

heute im Jahr sind Grenzen zwischen Syntax und Semantik  
fließend:

(1) Ist  $1/0$  syntaktisch oder semantisch unzulässig?

(2) Ist  $a = 0$ ;  $b = 1/a$ ; - u -

(3) Ist  $a + b$  syntaktisch zulässig wenn  $a$  int Variable ist  
und  $b$  eine double Variable ist

zu (1) erst zur Laufzeit, d.h. syntaktisch korrekt

ArithmeticException: Division by zero ← Fehlermeldung  
zur Laufzeit

zu (2) synt. korrekt diese Fehlermeldung bei int Variablen

bei double Variablen ok, Ergebnis ist Infinity

↑  
Konstante der Klasse Double

zu (3) zulässig, Ergebnis ist vom Typ double

Kap 5: Objekte, Typen, Datenstrukturen  
Einführung und Beispiele

Datentypen (in Java insb. Klassen)

- besteht aus
- Wertebereich
  - Operationen (in Java Methoden) die mit den  
Werten (in Java Objekte) operieren

Jede Prog Sprache hat erzeugbare Standardtypen

und trotzdem können wir selbstdefinierte Typen zu schreiben

in Java int double ...

in Java Klassen

Beispiele: int in Java

Wertebereich  $\{ N_{\min}, N_{\min} + 1, \dots, 0, 1, 2, \dots, N_{\max} \}$

$\swarrow$   
 $-2^{31}$

$\uparrow$   
 $2^{31} - 1$

Operationen

=

+

-

\*

/

%

==

!=

...

Beispiel: Ein selbstdefinierter Typ Fraction für Brüche

Wertebereich: alle Brüche der Form  $r = p/q$  wobei  $p, q$  int Werte sind und  $q > 0$  und in gekürzter Form

Operationen:

Fraction(a, b) Erzeugt Bruch r aus vorgegebenen int Zahlen a und b

$a = 2, b = -4 \rightarrow$  interne Darstellung  $-1/2$

toString() Schreibt Bruch  $r$  in der Form Zähler/Nummer  
↑ ↑  
ist Literale

doubleValue() berechnet Dezimalwert

$$\frac{3}{4} \rightarrow 0.75$$

getNumerator() Zähler als Wert bekommen

getDenominator() Nenner -u-

multiply(s) multipliziere mit s

add(s) addiere s hinzu

... equals(s) Test auf Gleichheit mit s

Verwendung dieser Methoden:

Fraction r, s; Deklaration

r = new Fraction(3, 4); ← Erzeugung Bruch  $\frac{3}{4}$

String str = r.toString(); str hat als Wert "3/4"

double d = r.doubleValue(); d hat Wert 0.75

s = new Fraction(4, 8); Erzeugt Bruch  $\frac{4}{8} \hat{=} \frac{1}{2}$

int a = s.getNumerator(); a hat Wert 1

r.multiply(s); r wird mit s multipliziert

$$r \text{ wird zu } \frac{3}{4} \cdot \frac{1}{2} = \frac{3}{8}$$

if ( r.equals(s) ) ... ergibt false



Urteil selbstdef. Typen (Klassen):

Muss Implementierung der Methoden nicht kennen, kann sie sofort verwenden

Beispiel (Spiele, Skatspiel)

Klasse / Typ Skatkarte

Wertebereich: { Karo 7, ..., Karo Ass, ... } 32 Werte

Methoden getFarbe() gibt Farbe (Karo, Pik, ...)  
getWert() gibt Wert 7, 8 -- Bube ...

Skatkarte(f, w) erzeugt Karte mit Farbe f und Wert w

Skatkarte() erzeugt zufällige Skatkarte

⋮

Wichtig ist Unterscheidung zwischen

- (abstrakten) Datentypen
  - Implementierungen der Typen
- ← bessere Arbeitsteilung  
← bessere Wiederverwendbarkeit von Code
- ↑ können verschieden sein oder noch nicht existieren

Datentypen sind extra wichtig für Kompilierung. Compiler

- Kann den erforderlichen Speicherplatz aus Typ ablesen
- Typinformationen zum Test auf Korrektheit von Anweisungen nutzen
- Typ eines Ausdrucks bereits ermittelbar ohne Reduzierung durch führen zu wissen

Setzt voraus, dass Typen dem Compiler bekannt gemacht werden  
 Deklarieren, Definieren

Typen müssen zur Compilezeit bekannt sein

"statisch getypte Prog. Sprachen"

Übergänge zwischen "verwandten" Typen sind möglich,  
 sonst wäre Typkonzept zu einschränkend

Casting Regeln (Übung)

```
double a = 3.14;
```

```
int b = (int) a;
```