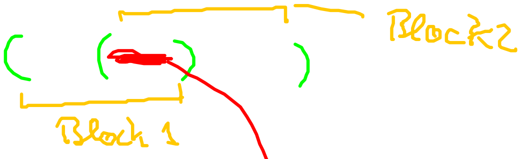
Korrekte Klammerausdrücke

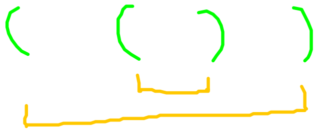
((( ) ( )) ( ) ( ))

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

  Ein solches Paar bildet einen <u>Block</u>
- Block beginnt mit (, endet mit)
- Blöcke dürfen sich nicht <u>echt</u> überlappen



echte Überlappung,
nicht korrekt mit dieser
Zuordnung der Klammern



korrekt, keine echte Überlappung

Korrekte Klammerausdrücke wichtig für Prog. Sprachen

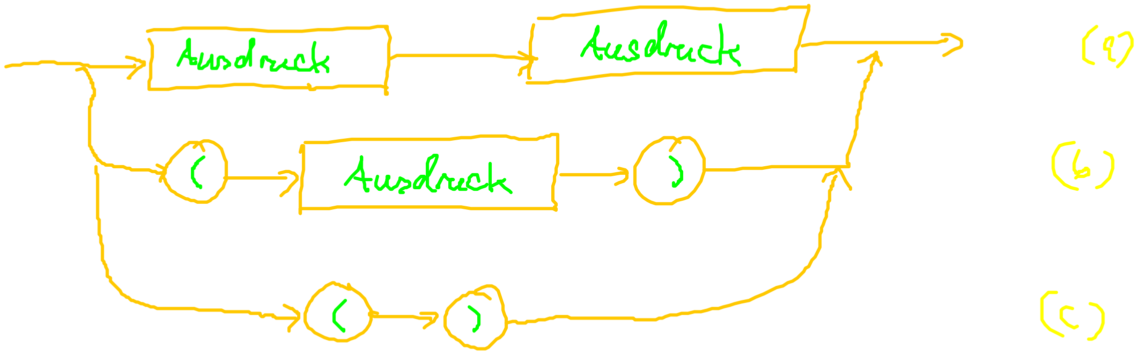
Compiler und Editoren müssen Korrektheit erkennen und die
Zuordnung (...) finden

dafür muss man <u>Bildungsgesetze</u> kennen

└ Grammatik

Syntaxregeln

in letzter VL: Grammatik entwickelt



Bsp: (c) ()

(a) (), (()) korrekt ⇒ ()(()) korrekt

(b) (()) korrekt ⇒ ((())) korrekt

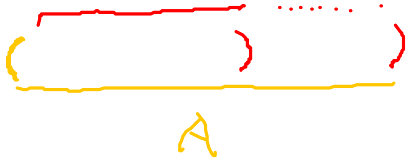
SATZ: Jeder korrekte Klammerausdruck mit Länge > 2 entsteht durch Anwendung der Regel (a) oder (b) aus kleineren korrekten Klammerausdrücken

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

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

⇒ A muss mit (anfangen

Betrachte die zugehörige)



2 Fälle:

(1)) steht ganz am Ende von A

(2)) steht nicht am Ende von A

zu (1)

A korrekt ⇒ B korrekt (sonst A inkorrekt)

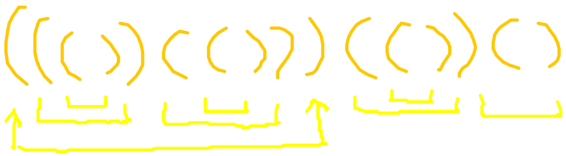
⇒ A ist aus B durch Anwendung von Regel (6) entstanden

zu (2)

A korrekt ⇒ B korrekt, A' korrekt

⇒ A durch Regel (4) aus A' und B entstanden □

Bsp:



4.4 Syntax versus Semantik

Grammatik

Bedeutung

keine klaren Grenzen

Erläuterung an Syntax für Stack im Skript

Syntax läßt zu   Ta1 – c3   illegale Turmzug

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 Kontextinformation

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 Identifiern

Parameterkorrespondenz bei Methodenaufrufen

...

Bedeutung von Integer Literalen)

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 $\{1, 2, .., 13\}$
Schreibe den Namen des n. Monats

Neben Syntax- und Semantikfehlern gibt es
noch die Logikfehler

Berechne den Umfang durch Multiplikation des Radius mit π

kann kein Compiler aufdecken

Auch in Java 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 eingebaute Standardtypen

und Möglichkeiten um selbstdefinierte Typen zu schreiben

in Java int double ...　　　　in Java: Klassen

Beispiele: int in Java

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

-2^{31}　　　　$2^{31}-1$

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

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

toString()	Schreibt Bruch r in der Form Zähler/Nenner ↗ ↗ int Literale
doubleValue()	berechnet Dezimalwert 3/4 → 0.75
getNumerator()	Zähler als Wert bekommen
getDenominator()	Nenner -"-
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 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 $4/8 \mathrel{\hat{=}} \frac{1}{2}$
`int a = s.getNumerator();`	a hat Wert 1
`r.multiply(s);`	r wird mit s multipliziert r wird zu $\frac{3}{4} \cdot \frac{1}{2} = \underline{\underline{\frac{3}{8}}}$
`if ( r.equals(s) ) ...`	ergibt false

Vorteil 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 (Kreuz, 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 ⇐ bessere Arbeitsteilung
- Implementationen der Typen ⇐ bessere Wiederverwendbarkeit von Code

↑ können verschieden sein oder noch nicht existieren

Datentypen sind extrem wichtig für Kompilierung. Compiler

- Kann den erforderlichen Speicherplatz aus Typ ablesen
- Typinformation zum Test auf Korrektheit von Anweisungen nutzen
- Typ eines Ausdruckes bereits ermitteln ohne Rechenprozess durchführen zu müssen

Setzt voraus, dass Typen dem Compiler bekannt gemacht werden

Deklaration, Definition

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;
```