4. Die Basiskonstrukte von Java

Kapitel 4,5 Ratz, Scheffler, Seese

Kapitel 6.1 bis 6.3 Küchlin, Weber

und die offizielle Java Beschreibung

von Arnold, Gosling

http://java.sun.com/docs/books/jls/

second_edition/html/j.title.doc.html.

Syntax = Wie Objekte (hier Programme) aussehen.

Semantik = Bedeutung eines Objekts. (Was ein Programm endet, welche Zustandstransformation?)

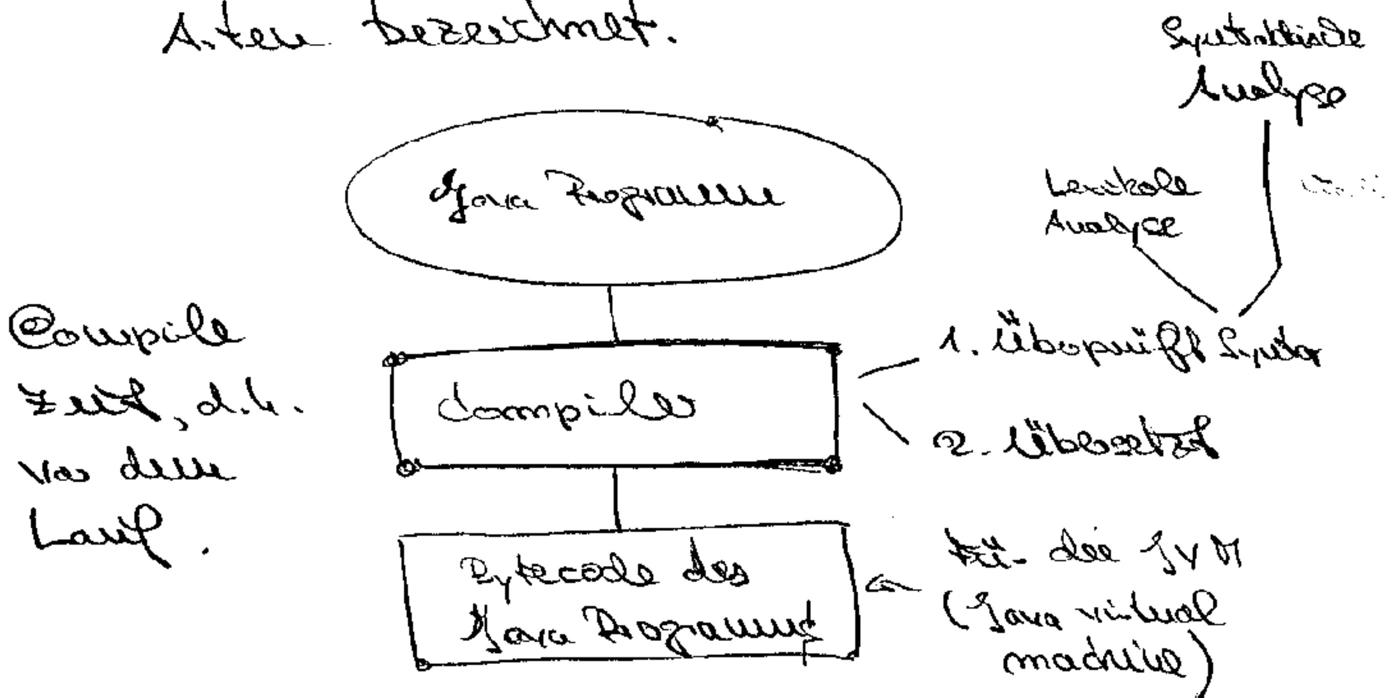
Ein Beispiel

$(+0,z_{-1}z_{-2}z_{-3} \dots)_{10}$ ist die syntaktisch korrekte Bezeichnung einer reellen Zahl, sofern $0 \leq z_{-i} \leq 9$.

Die Semantik, die bezeichnete Zahl, ist

$$x = \sum_{i=1}^{\infty} z_{-i} \cdot 10^{-i} \in \mathbb{R}.$$

Die Zahl $1 = 0,9\overline{9}$ wird auf 2 Arten bezeichnet.



Die lexikale Analyse bestimmt die Struktur der Basiselemente von Lsprachen. Wenn der Compiler ein Programm bearbeitet, erscheint es zunächst einfach als Folge von Bits, 0, 1. Das Ziel der lexikalen Analyse ist es, zu erkennen, was (= welche Bitfolgen)

- Namen oder Bezeichner
- Schlüsselworte
- Literale (und von welchem Typ)
- Trennzeichen
- Operatoren

sind. Man bezeichnet die so

erkannten Bestandteile als
Token, aus denen das Programm dann
weiter aufgebaut ist.

In einem 1. Schritt interpretiert
der Compiler die Bitfolge als
Folge von gemäß Unicode codierten
Zeichen.

http://www.unicode.org
2 Byte = 16 Bit pro Zeichen
Insgesamt $2^{16}-1$ = 65.535 Möglichkeiten.
Vorher ASCII (American Standard
Code for Information Interchange)

ASCII: 128 Zeichen 7 Bit.
33 nicht druckbare Steuerzeichen
(Zeilenvorschub)

Tab. 2.2. Der ASCII Zeichensatz

oct	hex	0	20	40	60	100	120	140	160
		0	10	20	30	40	50	60	70
0	0	nul	dle		0	@	P	`	p
1	1	soh	dc1	!	1	A	Q	a	q
2	2	stx	dc2	"	2	B	R	b	r
3	3	etx	dc3	#	3	C	S	c	s
4	4	eot	dc4	$	4	D	T	d	t
5	5	enq	nak	%	5	E	U	e	u
6	6	ack	syn	&	6	F	V	f	v
7	7	bel	etb	'	7	G	W	g	w
10	8	bs	can	(	8	H	X	h	x
11	9	ht	em	)	9	I	Y	i	y
12	A	lf	sub	*	:	J	Z	j	z
13	B	vt	esc	+	;	K	[	k	{
14	C	ff	fs	,	<	L	\	l	|
15	D	cr	gs	-	=	M	]	m	}
16	E	so	rs	.	>	N	^	n	~
17	F	si	us	/	?	O	_	o	del

Kodierung = Summe von

Zeilenwert + Spaltenwert

a wird in ASCII dargestellt durch

$$(60+1)_{16} = (61)_{16} = (97)_{10} = (\underbrace{1100001}_{7\ \text{Bits}})_2$$

ASCII → Unicode: Die ersten 8 Positionen auf 0 setzen. Also wird a zu

$$\overbrace{00000000}\,\overbrace{0110}^{6=}\,\overbrace{0001}^{1=}$$

Oder die 0 wird zu

$$(30)_{16} = (\overbrace{0011}^{3=}|\overbrace{0000}^{0=})_2 = (48)_{10},$$

und in Unicode 0...0|0011|0000.

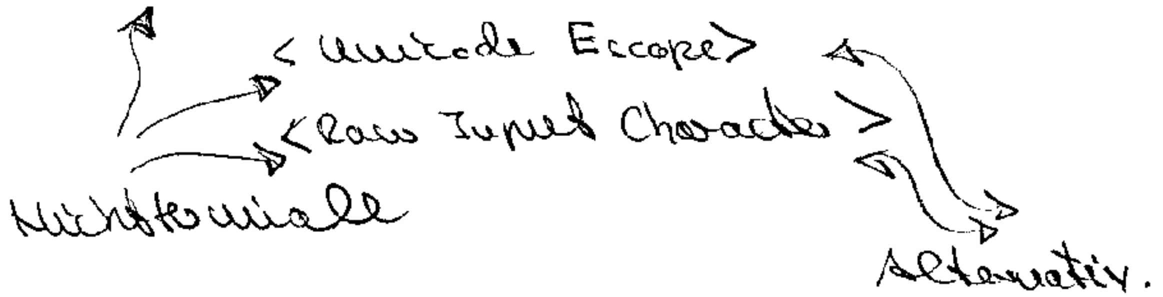
Wie kommt der Compiler an die Unicode Zeichen? Teile jeweils 16 Bits ab (oder entspr. auch 7 falls ASCII) und betrachte diese Bits als ein Zeichen.

Ganz so leicht ist es nicht, wegen der Unicode escapes. So steht \u abcd *nicht* für diese Folge von Zeichen, sondern für das Zeichen, dessen Unicode Kodierung $(abcd)_{16}$ ist. So ist \u0061 als a zu lesen, \u0030 als 0, ... Man kann alle Unicode Zeichen explizit ausgeben durch \u $x_3x_2x_1x_0$, wobei x_i hexadezimale Ziffern sind.

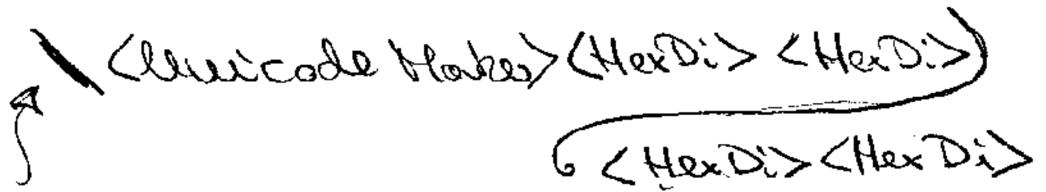
Damit sind wir also bei folgendem Problem:

Zunächst müssen die Unicode Zeichen richtig erkannt werden.

<Unicode Inp. Character> :

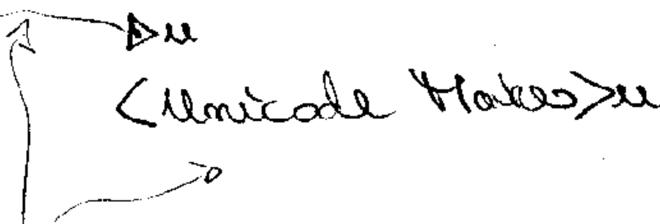


<Unicode Escape> :



Terminalzeichen

<UnicodeMarker> :



Rekursive Definition. Gleichbedeutend zu: UnicodeMarker ist folgende Menge:

- $u \in$ UnicodeMakro
- Ist $w \in$ Unicode Makro, dann auch $wu \in$ UnicodeMakro.

Also Unicode Makro $= \{u, uu, uuu, \ldots, uuuuuu, \ldots\}$,

sofern man impliziert, daß die Menge Unicode Makro nur mit den obigen Regeln aufgebaut ist.

Alternativ: Die kleinste Menge mit obigen Eigenschaften. Man spricht von einer induktiven Definition einer Menge. Ebenso kann $\mathbb{N}$ definiert werden

- $0 \in \mathbb{N}$
- Ist $n \in \mathbb{N}$, dann auch $n+1 \in \mathbb{N}$.

<RawInputChar> :

„Jedes Unicode Zeichen"

Hier sind auch die Steuerzeichen dabei.

<HexDigit> :

0
1
2
⋮
a
⋮
f
A
⋮
F

Alternative.
Lies: one of
0 1 2 3 — F.

Ein Problem \ :

Einmal Unicode Escape,
einmal \ an sich. \ an sich: \\

„Die Eingabe \\u2297 = \u2297
\ an sich.) Unicode Escape.
ergibt die Unicode Zeichenfolge
„ \u2297 = ⊗ ".

\\\\\\ gibt \\\. Ein Fehler ist

\uuuXYab, da nicht hexadezimal.

\u005cu005a wird zu \u005a,
Unicode von "\", binär: 00000000 0101 1100 (5 = 0101, 12 = 1100),
was nicht als Unicode escape gelesen wird, sondern so bleibt, wie es ist.

Nach Bearbeitung des Eingabestroms nach angegebenen Regeln hat der Compiler eine Folge von Unicode Zeichen. Diese wird als nächstes mittels der vorhandenen Zeilenendezeichen in Zeilen eingeteilt (z.B. zwecks Nummerierung, Erkennung des Endes von Kommentaren //...)

<Line Term>:

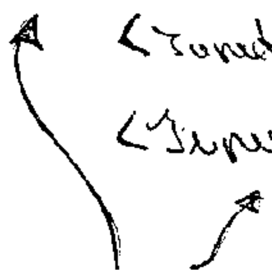
LF Zeichen (line feed, newline)
CR " (carriage return, return)
CR LF "

Haben jetzt Eingabezeichen in Zeilen getrennt.
Jetzt hat man (der Compiler) eine Folge von InputChar's gemäß Unicode und LineTerm's vorliegen.

Es ist alles bereit, die Basisbestandteile eines Java Programms, die Token, zu identifizieren.

<Input Elements> :

<Input Element>
<Input Elements> <Input Element>



Rekursiv

<L... >:

... CR, LF

<Input Element> :

<White Space>
<Comment>
<Token>

<Token> :

<Identifier>
<Keyword>
<Literal>
<Separator>
<Operator>

<White Sp> :

SP char (space)
FF char (form feed)
⋮

<Comment> :

<Trad. Comment>
<End of line Comment>

<End of line Comment> :

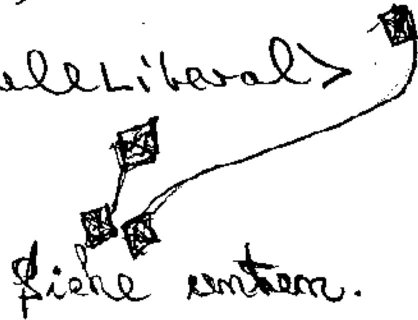
// <Character in line>$_{opt}$
<Line Term>

<Identifier>:

<IdentiChars> nicht

<Keyword>, <BooleanLiteral>

oder <NullLiteral>



siehe unten.

<IdentiChars>:

<JavaLetter>

<IdentifierChars> <JavaLetterOrDig.>

Rekursiv

<JavaLetter>:

A...

<JavaLetterOrDig>:

...

A,..,Z, a..,z,
_ (\u005F)
$ (\u0024)
auch α, β, ...

Zusätzlich
0,..,9

<keyword> : one of

```
abstract   double      interface  switch
assert     else        long       synchronized
boolean    extends     native     this
break      final       new        throw
byte       finally     package    throws
case       float       private    transient
catch      for         protected  try
char       goto        public     void
class      if          return     volatile
const      implements  short      while
continue   import      static
default    instanceof  strictfp
do         int         super
```

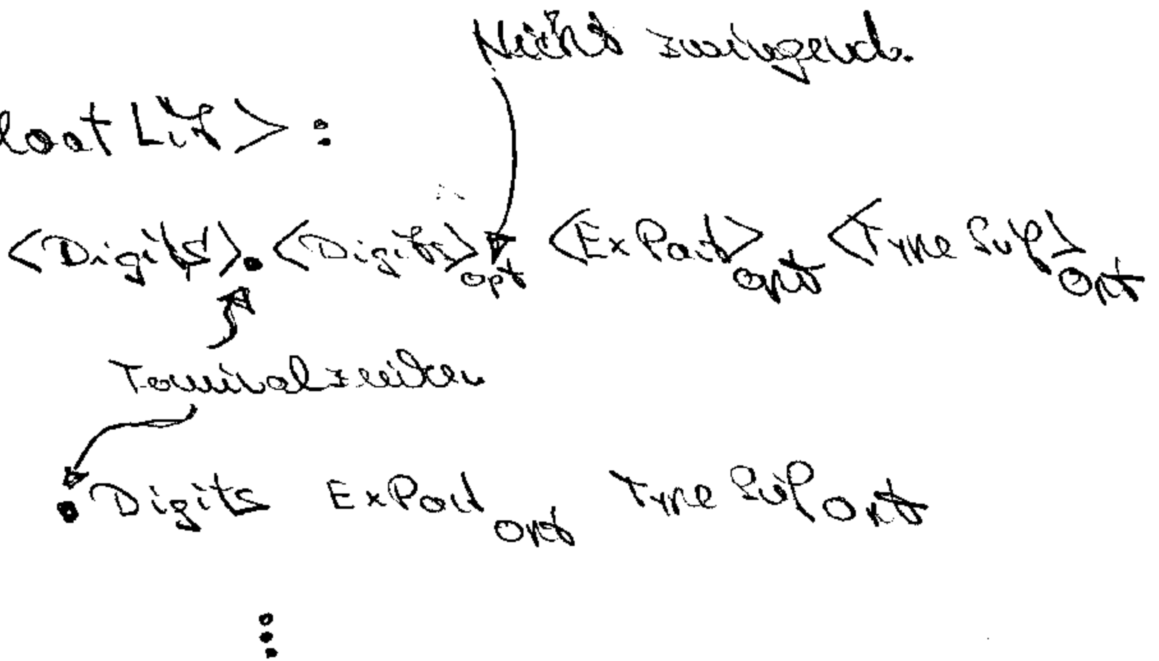
Die Schlüsselwörter `goto` und `const` werden momentan nicht verwendet; `assert` und `strictfp` sind in Java 2 neu hinzugekommen. Zusätzlich sind die Wörter `true`, `false` und `null` reserviert.

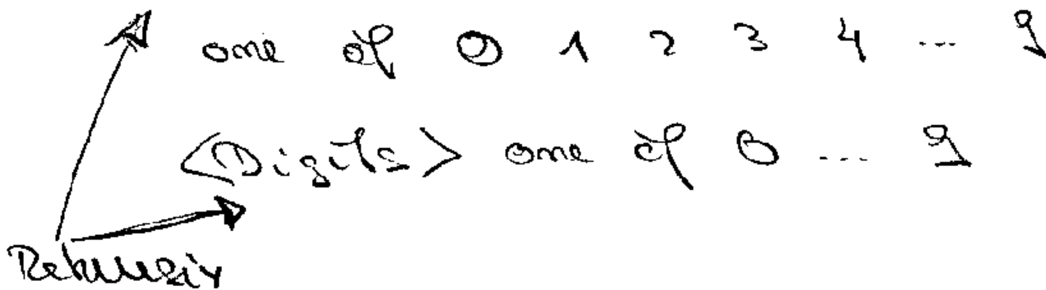
const, goto kommen augenblicklich nicht als Schlüsselworte vor.

true, false, null scheinen nur Schlüsselworte, sind aber Literale.

<Literals>:

- <IntLit>
- <FloatLit>
- <BoolLit>
- <CharLit>
- <StringLit>
- <NullLit>





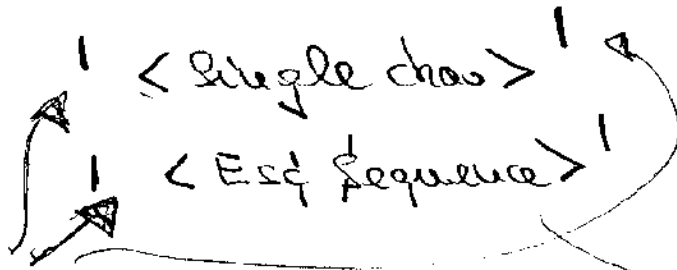
<Type Suf>:

one of f F d D.

<Boolean Lit>:

one of true false

<Character Lit>:



Terminalzeichen.

Ist dieses anders als Unicode Escape.

<Single char>:

<Input Char> aber nicht ' \ ,

CR, LF nicht hier, da <Line Term>.

Auch Unicode escapes

Zum Beispiel sind 'a',
'\uFFFF' CharLit's.
Das Unicode escape für line feed
'\u000a' ist nicht zulässig,
Die line feed Zeichen sind ja
auch schon „weg", d.h. als „neue
Zeile" interpretiert. Stattdessen
benötigen wir character escapes:

<EscSequence> :

```
\b      // \u0008 , backspace
\t
\n      // Linefeed
\r
\"      // "
\'      // '
 :
```

Terminalzeichen (1.19)

<StringLit> :

" <String Chars>$_{Opt}$ "

<String Chars> :

<String Char>

<String Chars> <String Char>

Rekursiv

<String Char> :

<Input Char> ohne " , \

<Esc Seq> \ Vergleiche S. 4.18

Auch Unicode escapes

Etwa " " (leerer string)

"Das ist" + "zu lang"

(Zusammengesetztes StringLit)

<NullLit>:

null

<Separator>:

one of (,) { } [ ] ; , .

<Operator>: one of

```
      =     >     <     !     ~
      ==    <=    >=    !=    &&
      +     -     *     /     &
      +=    -=    *=    /=    &=
-----------------------------------------
?     :
||    ++    --
|     ^     %     <<    >>    >>>
|=    ^=    %=    <<=   >>=   >>>=
-----------------------------------------
```

Namen
Schlüsselworte
Literale
Trennzeichen
Operatoren

Nach der lexikalen Analyse liegt dem Compiler das Programm strukturiert in obige Token vor. Er kennt die einzelnen Worte und weiß, ob es welches Schlüsselwort, ob es welcher Identifier, ...

Bisher Aufbau der Grundbestandteile.
Dazu lexikale Analyse des Compilers.

Jetzt kommt es zum schwierigeren

Teil: Aufbau des Programms. Dazu syntaktische Analyse.

Hier spielt es eine wichtige Rolle, daß Java eine stark getypte Sprache (strongly typed language) ist. Was heißt das?

In einer typfreien Sprache werden alle möglichen Wertebereiche als Mengen von Bitfolgen gesehen und gleich behandelt.
Typisch typfrei sind

```
x := 5;          oder    x = 3.70;
y := true;               y = 'a';
                         y = x + y;
z = x + y;                  :
if (z = true) then y = 5
              else x = true;
    :
```

Das ist sehr fehleranfällig.

Die erste Möglichkeit ist es, die Typen zur Laufzeit zu überprüfen. Das Programm

```
int x;
boolean y;
x := 5;
y := true;
z := x + y;
if (z = true) ...
```

wird dann zwar problemlos übersetzt, aber die Zuweisung z = x + y gibt zur Laufzeit einen Typfehler. Typüberprüfung zur Laufzeit erfordert natürlich Aufwand!

Im stark getypten Falle ist die Sprache derart, daß Typfehler wie x + y oben von vornherein, syntaktisch ausgeschlossen sind. Der Sinn einer starken Typisierung ist es, Fehlerquellen schon zur Compile-Zeit aufzudecken.
! Jeder Ausdruck hat deshalb bereits zur Compile-Zeit einen Typ. Und es ist sichergestellt, daß dieser Typ auch bei jedem Lauf – wie abgelegen auch immer – respektiert wird. Zur eigentlichen Laufzeit ist dann keine Typüberprüfung mehr erforderlich.

Noch einmal Definition der Typen

<Type>:
<Primitive Type>
<Reference Type>

<Prim. Type>:

boolean

byte
short
int
long
char
float
double

(byte, short, int, long, char: Integral types)
(float, double: Floating point types)
(byte … double: Numerical types)

Semantik:

boolean : true, false

byte : Von −128 bis 127 einschließlich

⋮

char : Von '\u0000' bis zu '\uffff'. (0 … 65535)

(char zählt mit zu den integral types.)

Operatoren für Werte des integral
types: & Argumente.

Ergebnis boolean:

< , <=, >, >=   Semantik: Vergleiche

== , !=   Semantik:
Gleichheitstest,
Ungleichheitstest.

Ergebnis im allgemeinen int oder long:
außer bei den shift Operatoren.

+ , -   Einstelliges + , - .
Semantik:
+ positiv, - negativ.

* , / , %   Multiplikative Operatoren.
/ rundet zu Null hin.
% kann negative Zahl
geben.

Operator	Anmerkung	Beschreibung
+ , -		Zweistellige additive Operatoren
--		Dekrement um 1 Präfix oder Postfix

++	Präfix: Erst erhöhen, dann Wert geben.
	Postfix umgekehrt.
	a = 5;
	b = a++;
	ergibt b=5!
		Inkrement um 1
		Präfix oder Postfix

<<		x<<n x um n Bits nach links. Von rechts Nullen. Mult. mit 2^n. $(-1 \ll 1) = -2$
>>		x>>n n Bits nach rechts. Von links Vorzeichenbit. Arithmetische shifts $(-1 \gg 1) = -1$ $(-4 \gg 2) = -1$ $(-1/2) = 0$.

`>>>` — `x>>>n`: x um n nach rechts. Von links Nullen. Logischer shift.

$(-1>>>1) = 2^{31} - 1$.

Ist bei den shift Operationen das n größer als die #Bits von x, so wird um Mod(n, #Bits von x) geshiftet!

`~` — Bitweise Negation

`&`, `|`, `^` — Bitweises Und, Oder, exklusives Oder.

`? :` — `(x==0) ? a : b`
Falls x==0, dann a, sonst b.

`(byte)`, `(short)`, ... — Type cast. Typumwandlung

`+` — String Konkatenation
`"abc" + 3.5 = "abc3.5"`

Für floating point types.

Ebenso außer, shift und ~, &, |, ^.

Operatoren für boolean sind

== , !=

! Negation, etwa !b

Overloading, da auch bei integral types

&, ^, |

&& wie &, aber ist linkes Argument false keine weitere Auswertung des rechten.

Lazy evaluation

|| wie | aber links true, rechts keine Ausw.

$\frac{d}{4}$:

\+ "abc" + true = "abctrue".

Beachte type cast bei boolean

nur wenn

(boolean)(<Ausdruck>)

und <Ausdruck> hat Typ boolean.

Zur Verdeutlichung. Andere casts

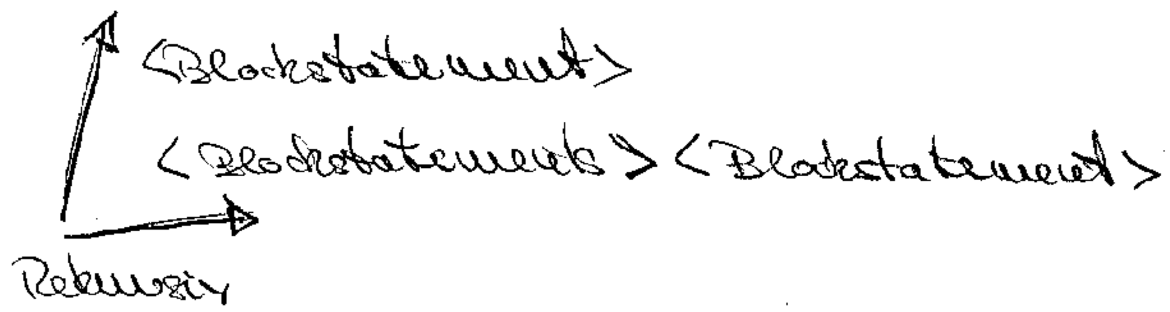
sind mit boolean nicht zulässig!

Bisher kennen wir nur lokale Variablen. Lokale Variablen beziehen sich auf Blöcke.

<Block> :

{ $\text{Blockstatements}_{opt}$ }

<Blockstatements> :



<Blockstatement> :

<Loc. Var Decl>

<Class Decl>

<Statement>

<LocalVarDecl> :

final_{opt} <Type> <Variable Declarators>;

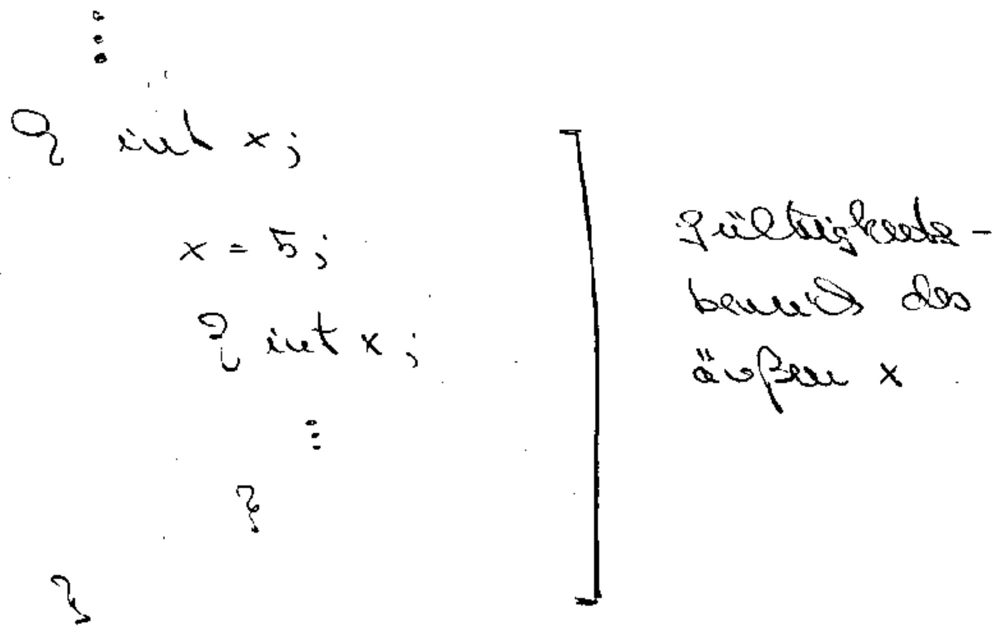
<Variable Declarator>:

Beispielhaft x, y, x = 5, x = 3+7, ...

Jede LocalVarDecl. ist direkt in einem Block enthalten.

Zwei Probleme lokaler Variablen:

- Gültigkeitsbereich (scope)
- Initialisierung



Obiges Programm gibt einen Syntaxfehler.
Lokale Variablen dürfen nicht
innerhalb ihres Gültigkeitsbereichs
noch einmal deklariert werden.
Zulässig ist dagegen

```
{ int x;
    x = 7;
    { int y;              ] Gültigkeitsbereich   |
      x = ;               ]      von y           |  von x.
    }                                            |
    { int y;              ] Gültigkeitsbereich   |
      x = ;               ]      von y           |
    }                     ] (ein anderes y)      |
}
```

Vor Gebrauch muß eine lokale Variable initialisiert sein, sonst Syntaxfehler.

Zuweisung:
- Ändert Wert einer Variablen.
- Hat Wert, wie ein Ausdruck. Den zugewiesenen Wert. ?

Im

```
:
{ int k;
  if (x>0 && (k= IOTools.readInteger())>=0)

  System.out.println(k);

}
```

wird k als initialisiert erkannt. Also kein Syntaxfehler. Ebenso in

```
{ int k;
  while (true)
  { k=n;
    if (k >=5) break;
    n= 6
  }
} System.out.println(k)
```

wird k als initialisiert erkannt.

Dagegen wird in

```
{ int k;
  int n = 5;
  if (n > 2) k = 3;
  System.out.println(k)
}
```

k nicht als initialisiert erkannt.
Syntaxfehler. Boolsche Ausdrücke werden nicht ausgerechnet, selbst wenn das zu Compile-Zeit möglich ist. Es wird nur „schematisch" geprüft, ob eine Initialisierung sicher stattfindet.

Akzeptiert wird auch

```
:
{ int k;
  if (x>y)
     k=3;
     else
        k=4;
  System.out.println(k)
}
```

denn es wird erkannt, daß k vor Gebrauch einen Wert bekommt.

Jedoch nicht geht

```
:
{ int k;
  if (x>y)
     k=3;
  if !(x>y)          // nicht x>y
     k=4;
  System.out.println(k)
}
```

Zusätzlich zu den oben aufgeführten Operatoren hat Java noch die Zuweisungsoperatoren (assignment operators):

= Linke Seite: Speicherplatz (Variable). Rechte Seite wird zugewiesen.

*= X *= e bedeutet
x = (Typ von x)(x * e)

/= , %= ,
+= , -= , <<= ,
>>= , >>>= , &= ,
^= , |=

Analog.

Syntaktisch korrekt ist dann

```
⋮
short x = 3;
x += 4.6;
⋮
```

und ergibt ein Resultat $x = 7$.

Denn es ist äquivalent zu

```
⋮
short x = 3;
x = (short) (x + 4.6)
⋮
```

Das ist normalerweise vom Typ double! (bezogen auf x + 4.6)

Ausdrücke werden mit den bisher erwähnten Operatoren induktiv aufgebaut. Ihr Auswertung (Semantik)

wird von links nach rechts vorgegangen. Mehrere Operatoren gleicher Bindungskraft: linksassoziativ. Nur Zuweisungsoperatoren assoziativ nach rechts:

$a = b = c$ wird als $a = (b = c)$ ausgewertet. Also a bekommt den Wert von b nachdem die Zuweisung $b = c$ stattgefunden hat.

Eine Zuweisung ist auch ein Ausdruck. Ihre Semantik ist 2-fach:

- Einen Wert (normalerweise der zugewiesene Wert, aber i++, ++i)
- Ändert den Wert der Variablen auf der linken Seite (Seiteneffekt)

Operatoren nach ihrer Bindungskraft (Präzedenz):

Tab. 6.2. Präzedenzen von Java-Operatoren

Die Reihenfolge ist von großer Präzedenz absteigend zu kleiner Präzedenz angegeben. Operatoren gleicher Stufe stehen in einer Zeile.

Postfix Operatoren	`[] . (params) expr++ expr--`
Unäre Operatoren	`++expr --expr +expr -expr ~ !`
Erzeugung und Anpassung	`new (type) expr`
Multiplikative Op.	`* / %`
Additive Op.	`+ -`
Schiebe-Op.	`<< >> >>>`
Relationale Op.	`< > >= <= instanceof`
Gleichheits-Op.	`== !=`
Bitweises und log. UND	`&`
Bitweises und log. XOR	`^`
Bitweises und log. ODER	`|`
Bedingtes logisches UND	`&&`
Bedingtes logisches ODER	`||`
Bedingung	`?:`
Zuweisungs-Operator	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

Noch ein paar Beispiele:

```
y = (x >= 0 ? x : -x)
```

setzt y auf |x|, den Betrag von x.

```
a && b
```

ist gleichbedeutend zu `(a ? b : false)`

Also:

```
a * b + c = (a * b) + c
b * a++ = b * (a++)
```

a || b ist gleichbedeutend zu (a ? true : b)

a && b && c ist gleichbedeutend zu

((a ? b : false) ? c : false) .

c wird nur ausgewertet, wenn a und b true sind.

Man beachte, daß alle Bestandteile von Ausdrücken zu Compile Zeit einen Typ erhalten. Man könnte natürlich ganz streng + einmal nur auf 2 float Argumenten, dann auf 2 double Argumenten, ... zulassen. Der Typisierungs-mechanismus erlaubt aber

die Typ Aufweitung (type promotion).

Bei unären arithmetischen Operatoren gilt: Ergebnis hat Typ int.

byte, char, short werden zu int.

Bei zweistelligen Operatoren gilt: Ein Operand double, anderer zu double aufgeweitet. Andernfalls float, wenn einer float. Andernfalls long, wenn einer long, andernfalls beide auf int ausgeweitet. Also ist das Ergebnis immer vom Typ int oder höher.

Außer bei shift. Beachte: Rechnerarithmetik nur für int.

: für best. den Typ

4.42

1 + 1.0f : float

1 + 1.0 : double

1L + 1.0f : float

1 + 1L : long

Bei int → long.
Ersten 32 Bits = 0
falls ≥ 0, sonst 1!
int → float Genauig-
keit geht verloren.

Nach Deklaration byte a, b, c;

a + b : int

-b : int

a = -b   Syntaxfehler

c = a + b   "

Beachte aber korrekt ist

byte b = 7;

byte c = b;

Dagegen geht nicht

byte c = -b.

Interessant ist, daß

```
final short s = 5;
short t = -5;
short t = s + 17
```

wiederum geht, da die Werte rechts zur Compile Zeit feststehen und von der Größe her passen.

Implizite Typkonversionen bei „Vergrößerung" des Wertebereichs wie oben.

Bei einer Verkleinerung ist eine explizite Typkonversion (type cast) erforderlich:

Einige Beispiele, syntaktisch korrekt:

```
int x = (int) 1L

char z = (char) 127
```

127, das Literal vom Typ int wird in ein Zeichen umgewandelt.

```
byte q = (byte) (a+b)
```

Beachte: type cast bindet stärker als +.

Type cast auf boolean geht nicht.

Was geschieht bei einem cast?

Bei integral types fallen die höherwertigen Bits weg. Dabei Vorzeichenänderung möglich.

Bei double nach float geht Präzision verloren oder es entsteht infinity.

Was passiert bei floating point types nach integral types? Probleme der Größe.
Übungsaufgabe.

Einige Beispiele

(byte) 128 = -128
1 0000000

(byte) 129 = -127

(byte) 13.5 = 13

(byte) 255 = -1
11111111

(byte) 13.5 = 13

Bei geschachtelten Ausdrücken:

Typanpassung von innen nach außen.

Ein paar Beispiele

double d = 1/2

ergibt d = 0.0 . Zuerst gibt 1/2

ohne Typanpassung 0 , dann 0.0

durch Typaufweitung.

0.8 + 1/2 = 0.8 ,

da 1/2 zunächst zu 0.0 wird.

double d = 1.0/2

ergibt d = 0.5 und schließlich ist

0.8 + 1.0/2 = 1.3 ,

da zunächst 2 zu double aufgeweitet

wird.

Schließlich noch die Anweisungen
(statements) von Java.

im Unterschied zu
Zuweisungen !
(assignment statements)

Wir geben nur beispielhaft vor:

<Statement> :

<Block>

<Empty Statement>

<Expression Statement>

⋮

<If Then Stat>

<If Then Else Stat>

<Statement No Short If>

⋮

<Expr. Statement>:

<Assignment>

<Pre Increment Expr> //++i

⋮

<If Then Stat>:

if (<Expression>) <Statement>

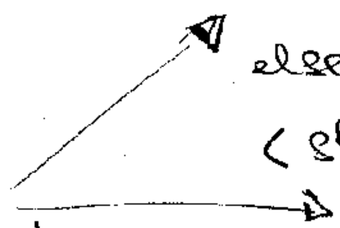
<If then Else Stat>:

if (< Expression>)

<Statement No Short If>

else

<Statement>



Hier rekursive Vorkommen von Nichtterminalen.

Keine if (<Expr>) <Statement>

<statement No Short If> :

Im Prinzip kann

if (<Expression>)

<Statement>

Hier <Expression> vom Type boolean.

Beachte das Problem des dangling else:

```
if (b)
    if (c)
        x = 7
    else
        x = 8
```

Hier kann die leere Anweisung helfen

```
if (b)
    if (c)
        x = 7;
    else ;
else          // Bei b = false
    x = 8
```

Zu welchem if gehört das else?

Java stellt sicher: Im Zweifel zum Inneren.

Das switch statement dient der Fallunterscheidung:

```
switch (ç)
{
    case konst1 : { statement 1; break;}
       :
    case konstn: { statement n; break; }
    default:   { statement d;}
}
```

ç ist ein Ausdruck. Statement i bei ç = konst i. Das break bewirkt daß ans Ende des switch Anweisung gegangen wird. Auch ohne break syntaktisch zulässig. Dann auch die weiteren Zeilen

Zusammenfassen mehrerer Fälle ist möglich:

```
switch (c)
{
   case 0 : case 1: case 2 : case 3 : { res=1; break;}
      ⋮
   default : { res =0;}
}
```

Jetzt noch die Schleifen:

<While Stat> :

while (<Expression>) <Statement>

<Expression> vom Typ boolean.

Beachte:

```
if (b)
   while(c)
      if c then x = a
else y = b
```

Wäre ein while auf Statt If.

Rekursion von oben!

<Do Statement>:

do <Statement> while <Expression>

<Expression> vom Typ boolean. Semantik:

<Statement> wird ausgeführt,

dann wird <Expr.> getestet, usf ...

Die for Schleife. Zunächst ein Beispiel

```
int i;
int res = 0;
for (i=1; i<10; i++)
  {
    res = res+i
  }
```

ist gleichbedeutend zu:

```
int i;
int res = 0;
i = 1;
while i < 10
{
   res = res + i;
   i++;
}
```

```
for (;;)
{
  //
}
```

ist eine Endlosschleife.

<For Statement> :

for (<For Init$_{opt}$> ; <Expression>$_{opt}$; <For Update>$_{opt}$)

<Statement>

Hier Rekursion von oben.

i++, i = 10

<For Init> :

<Statement Expr. List>

<Local Var Decl>

int i = 5

Scope: Die ganze Schleife.

<For Update> :

<Statement Expr. List>

<Statement Expr. List> :

<Statement Expr>

<Stat Expr. List> <Statement Expr.>

<Expression>$_{opt}$ vom Typ boolean.

Semantik:

- Initialisierung mit <ForInit>
- Testen ob <Expression>$_{opt}$ true.
  Nicht vorhanden ist wie true.
- Ausführung von <Statement>
- <ForUpdate>$_{opt}$ wird gemacht
- <Expression>$_{opt}$ wird getestet,
  bei true <Statement> ...

Ist bei <Expression>$_{opt}$ kein Ausdruck, dann Termination nur durch ein break in der Schleife.

Hier noch das leere statement

<EStatement>:

;

Schließlich noch Marken und

Sprunganweisungen.

Jeder Anweisung kann eine Marke

(label) vorgesetzt werden:

<LabeledStatement>:

<Identifier>:<Statement>

Die Marke des Statements.

Das unmarkierte break-statement:

break;

Semantik: Beendet sofort die unmittelbar umgebende switch-, while-, do- oder for-Anweisung. Fehler, falls diese nicht existiert.

Markiertes break:

break <Identifier>;

Beendet die umgebende Anweisung, die durch den Identifier markiert ist.

Die continue Anweisung ohne Marke

continue;

bewirkt, daß bei einer umgebenden Iterationsanweisung zur nächsten Iteration gegangen wird.

Markierte continue Anweisung

continue <identifier>;

entsprechend, aber daß die nächste umgebende Iterationsanw. mit dem label <ident> gewählt wird.

Schließlich noch return; bewirkt das Ende des aktuellen (Unter-) Programms.

Es folgt das Beispiel nach S. 107 Ratz, Scheffler, Seese.