

Informatik II

Sommersemester 2004
Prof. Dr. Martina Zitterbart



Übungsleitung:
Erik-Oliver Blaß
Hans-Joachim Hof



Folien basieren sehr stark auf den von Prof. Lockemann im Sommersemester 2003 verwendeten



Medien und Unterlagen:

- Web: http://www.tm.uka.de/lehre/aktuell/vorlesungen/V_INFO2.html
- News: <news://news.tm.uka.de/tm.lehre.informatik2>

Veranstaltungen:

- Vorlesung: Mo, Mi 14:00 Uhr, Hörsaal am Forum
- Übung: Do 14:00 Uhr, Hörsaal am Forum
- Tutorien: WebInScribe

Übungsblätter:

- Ausgabe: Do unmittelbar vor der Übung im Web
- Abgabe: bis Do 12 Uhr, Kästen im Keller Infobau (Geb. 50.34)

Klausur:

- Voraussichtlich 02.08.2004, Nachklausur 04.10.2004
- Bonus für Bearbeitung der Übungsblätter



Feedback und Informationsfluß

Premium
Support



Sprechstunde Prof. Zitterbart:

- Falls Probleme in Level 1-3 nicht gelöst

Third Level
Support



Sprechstunden Übungsleiter:

- Falls Problem nicht per News-group od. Tutor gelöst werden konnte.

Second Level
Support

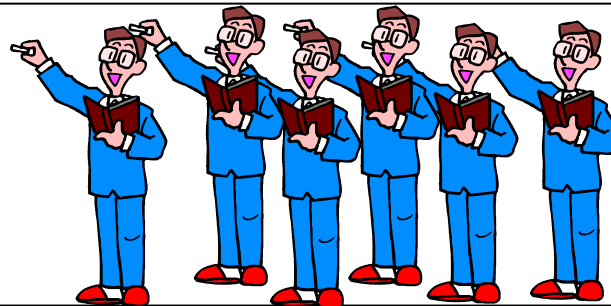


tm.lehre.informatik2

Newsgroup:

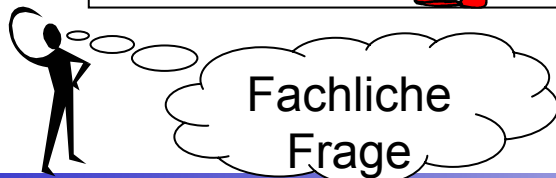
- Für die Allgemeinheit interessante Fragen
- Übungsleiter und Tutoren lesen mit

First Level
Support



Persönlicher Tutor:

- Erster Ansprechpartner
- Gibt Fragen im Zweifelsfall weiter, Feedback über News

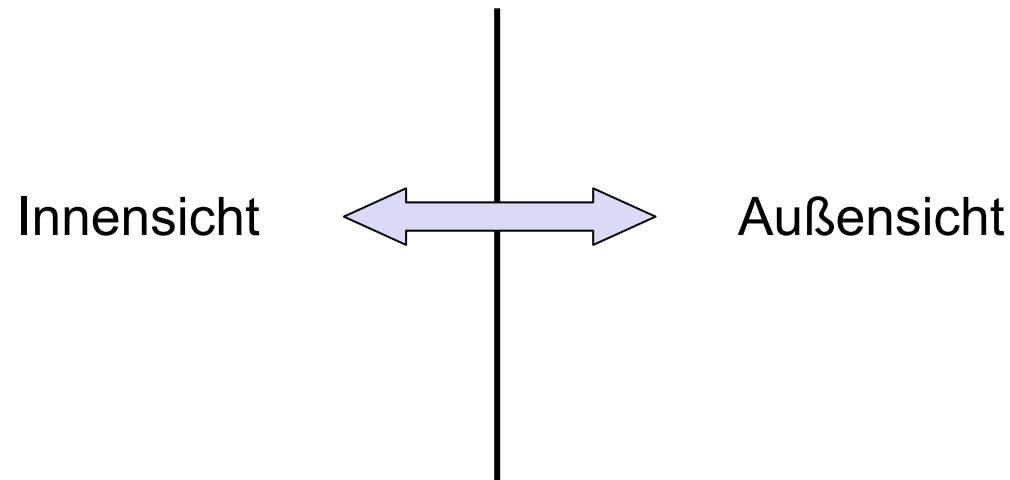


- Bitte die Reihenfolge einhalten, damit Probleme schnell und effizient gelöst werden können

- Für Prüfungsangelegenheiten ist an der Fakultät primär Herr Barthelmeß zuständig

Vorgehen

Übersicht Informatik II: Systemverständnis



System:

- Abgegrenzter Ausschnitt aus der realen oder gedanklichen Welt zur Erfüllung eines gegebenen Zwecks. Ein System wird bestimmt durch die Beziehungen zu seiner Umwelt, durch die in ihm enthaltenen Bestandteile und deren Beziehungen zueinander sowie durch das dynamische Verhalten.



H.-J. Schneider; Lexikon der Informatik und Datenverarbeitung;
Oldenbourg-Verlag, 1997

- Kollektion von Gegenständen, die in einem inneren Zusammenhang stehen, samt den Beziehungen zwischen diesen Gegenständen.



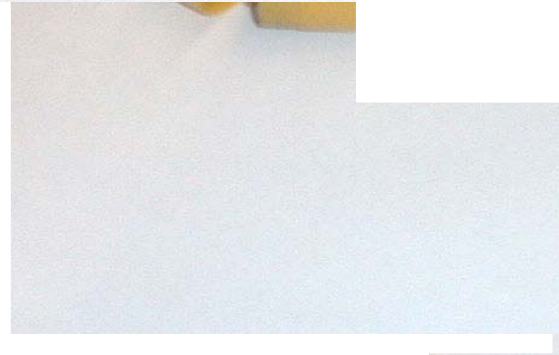
G. Goos; Vorlesungen über Informatik: Band 1;
Springer-Verlag, 1995

Es lässt sich unterscheiden:

- *Innensicht*
 - Bausteine, deren Beziehungen
- *Außersicht*
 - Systemgrenze, Beziehungen über diese Grenze

Beide Sichten werden in Informatik 2 betrachtet.





Innensicht

Imperatives Programmieren

(Konstrukte aus Java)

- Praxis: Basistypen, Anweisungen, Verbunde, Felder, Schleifen, Rekursion
- Theorie: Syntaxbeschreibung, Induktion, Schleifeninvarianten
- Ausnahmebehandlung

Objektorientiertes Programmieren

(Java)

- OO-Klassen
- Objekt und Zustände
- Methoden und Vererbung in Java
- Java: Einführung aller restlicher OO-Sprachkonstrukte

Außensicht

Dienste

- Funktionalität/Schnittstellen
- Qualitätsparameter: Aufwand, Zuverlässigkeit, Skalierbarkeit, Persistenz
- Algorithmenwahl

Objektorientierung

- Objekte, Zustände, Methoden
- Entwurf mittels UML
- Vererbung



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

- Enge/lose Kopplung
- Protokolle/Automaten
- Verteilung



**1.1 Funktionales und imperatives
Programmieren**

1.2 Zustandsbeschreibung

1.3 Ablaufsteuerung

1.4 Methoden

1.5 Rekursion

1.6 Java-Spezifikation



Innensicht

Imperatives Programmieren

(Konstrukte aus Java)

- Praxis: Basistypen, Anweisungen, Verbunde, Felder, Schleifen, Rekursion
- Theorie: Syntaxbeschreibung, Induktion, Schleifeninvarianten
- Ausnahmebehandlung

Objektorientiertes Programmieren

(Java)

- OO-Klassen
- Objekt und Zustände
- Methoden und Vererbung in Java
- Java: Einführung aller restlicher OO-Sprachkonstrukte

Außensicht

Dienste

- Funktionalität/Schnittstellen
- Qualitätsparameter: Aufwand, Zuverlässigkeit, Skalierbarkeit, Persistenz
- Algorithmenwahl

Objektorientierung

- Objekte, Zustände, Methoden
- Entwurf mittels UML
- Vererbung



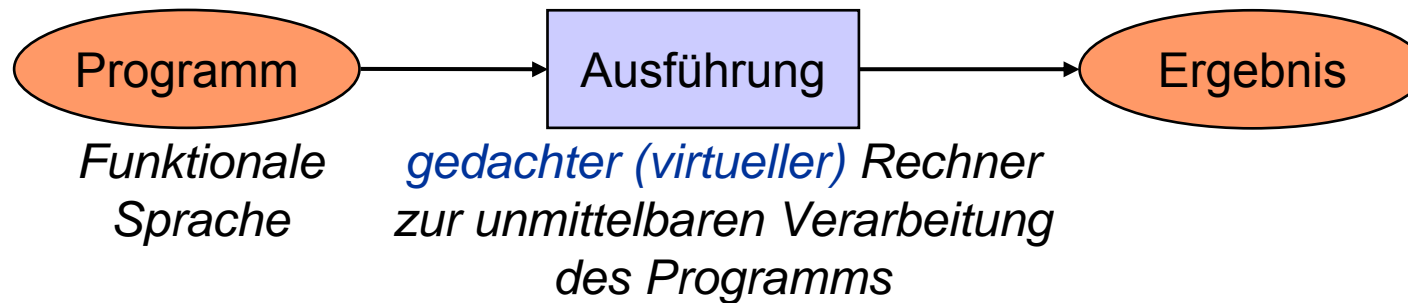
In Informatik 1:

- Funktionale Programmierung
 - Programme werden abstrakt formuliert – kein Bezug zur Maschine, enger Bezug zur Mathematik
 - Lambda-Kalkül
 - Recht einfache Semantik
 - Erhöhte Beweisbarkeit der Programme
 - „zeitlos“ – d.h. Zustand der Maschine spielt keine Rolle
 - Rekursive Deklaration von Funktionen
 - Verarbeitungsprinzip: Substitution



- Zusätzliche Verarbeitungsregeln in Haskell
 - Vorrangregelung für Operatoren
 - Faule Auswertung
 - Erste anwendbare Funktionsdefinition

Operationale Semantik einer funktionalen Sprache:



Haskell-Beispiel

Fibonacci-Zahlen:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib (n+2) = fib n + fib (n+1)
```

Ausführung durch schrittweise Substitution bei `fib 3`:

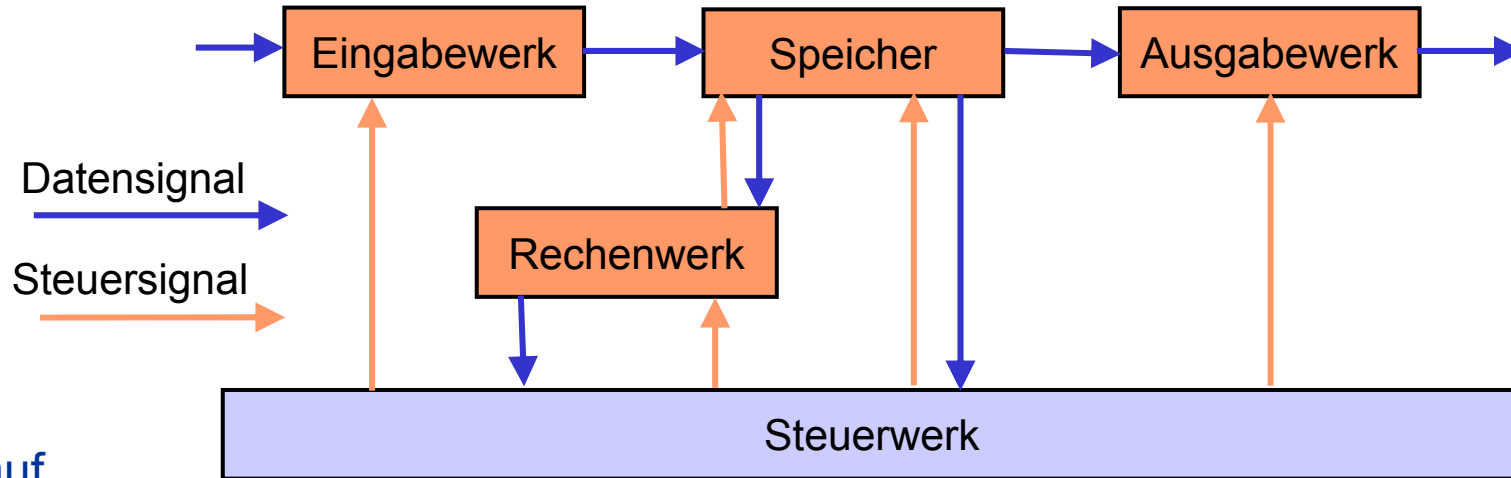


Realer Rechner

- von-Neumann-Rechner: Grundlage für die Konzipierung heutiger Rechner
 - Definition von Burke, Goldstine und von Neumann, 1946



von Neumann

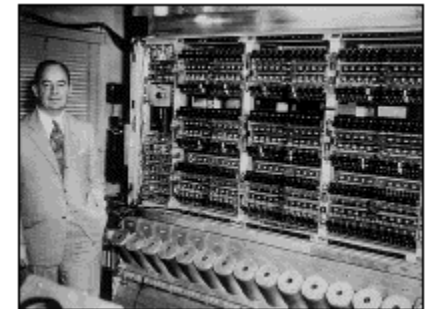


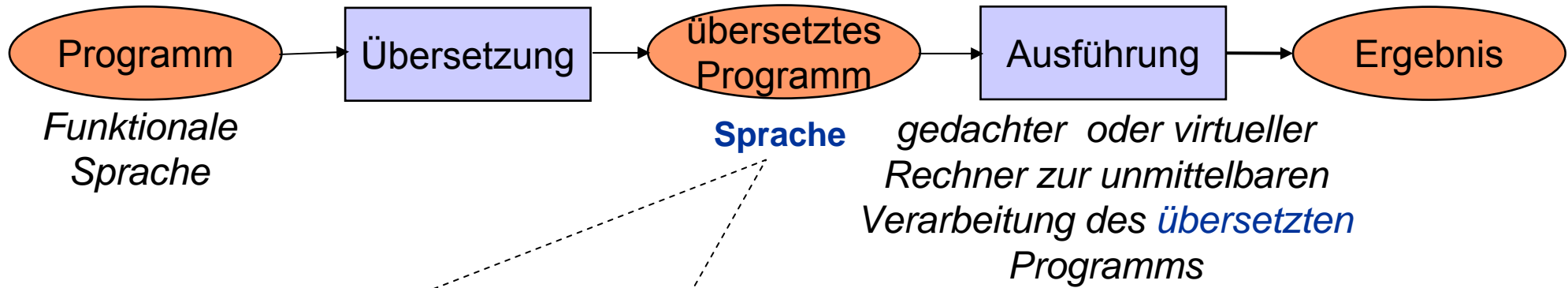
Ablauf

- Programm und Daten sind in einheitlichem Speicher
- Die Befehle werden nacheinander aus den *Zellen* des Speichers geholt
- Ein Befehl wird dekodiert, die entsprechenden Daten werden aus dem Speicher geholt, und der Befehl wird ausgeführt
- Das Ergebnis ist ein Wert, der als in einer Zelle abzulegender Inhalt interpretiert wird oder als Referenz auf die Zelle mit dem nächsten Befehl
- Danach wird Zelle mit dem nächsten Befehl betrachtet und der nächste Befehl abgearbeitet



- * 28.12. 1903 in Budapest
† 8.12.1957 in Washington
- Studium Mathematik in Budapest
1928 Habilitation in Berlin
1933 Professor für Mathematik in Princeton
- Wegbereiter der amerikanischen
Computerentwicklung
- Hauptidee: Gemeinsamer Speicher für
Programme und Daten, wodurch sich
das Programm selbst verändern kann (1946)
- So aufgebaute Computer mit einem Prozessor
bezeichnet man heute als „von Neumann“-
Computer, bzw. von Neumann-Architektur
- Wesentlicher Beiträge zu den Computern
 - Havard Mark I (ASCC)
 - ENIAC





Unser nächstes Ziel:

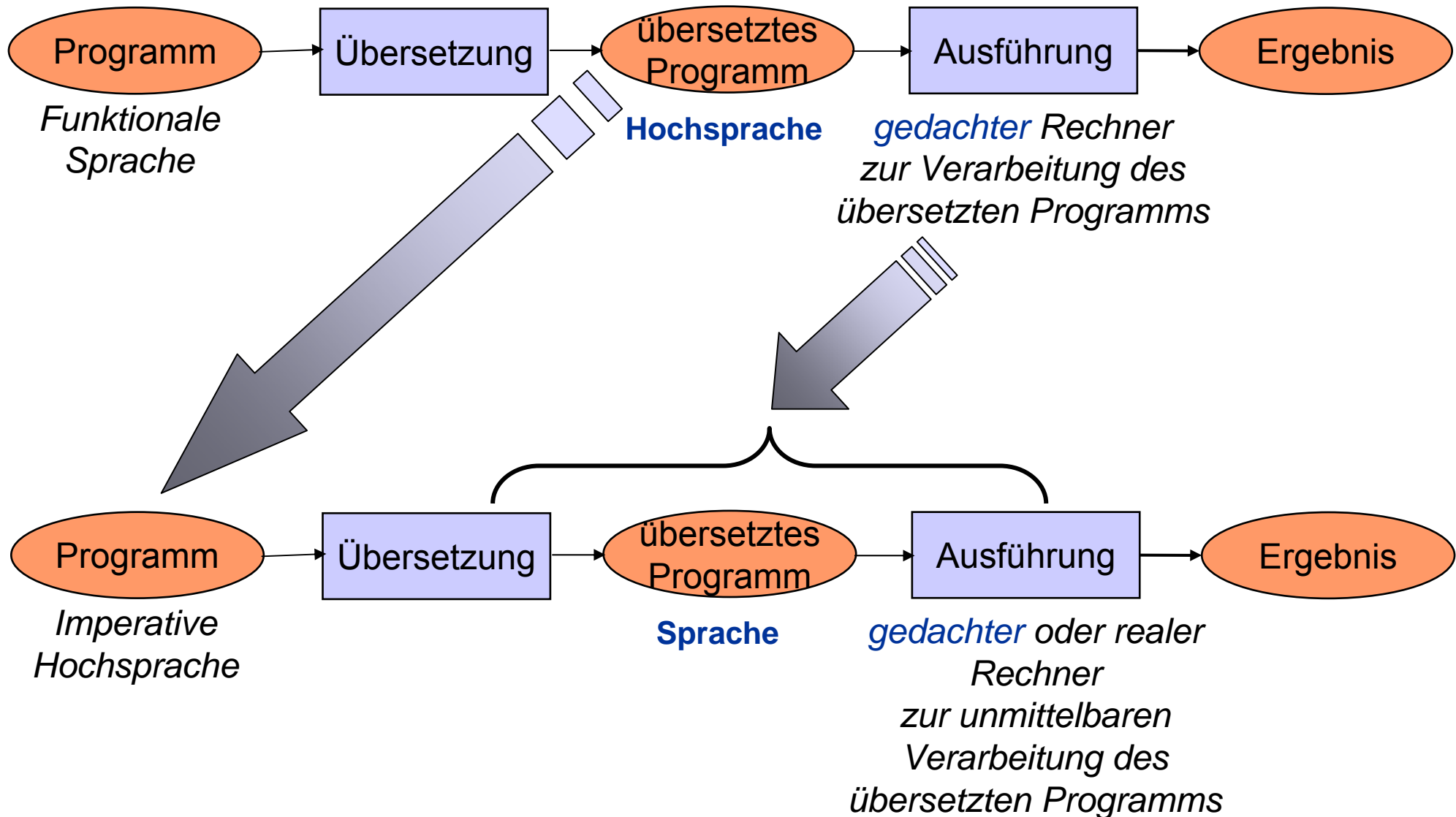
Wir suchen nach einer Programmiersprache (für das übersetzte Programm), die

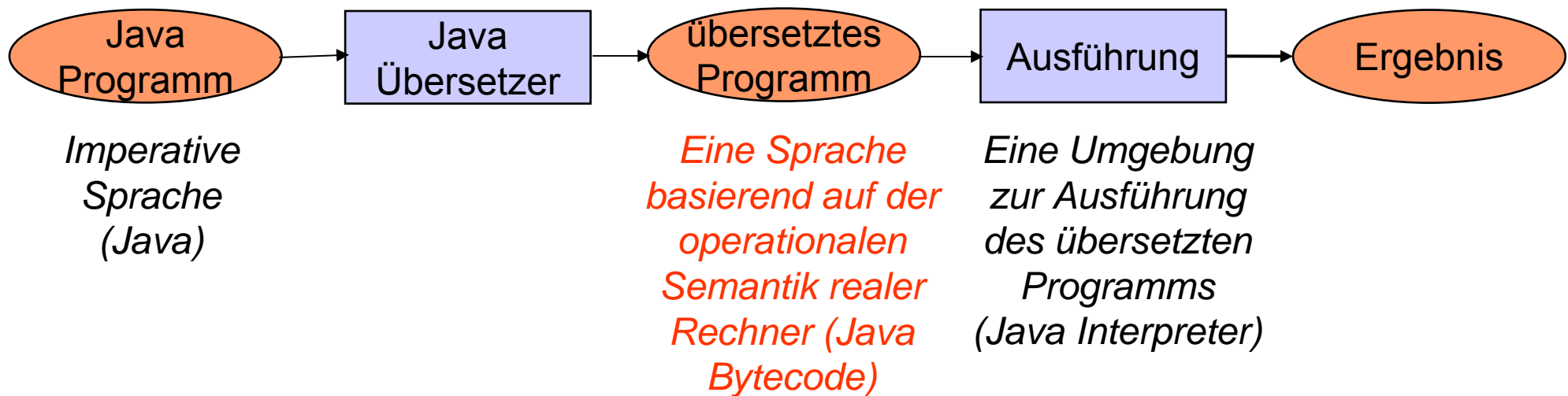
- einerseits sich an die operationelle Semantik realer Rechner anlehnt,
- andererseits aber dem Programmierer das Denken in dessen Problemstrukturen und -begriffen erlaubt.

Wir nennen eine derartige Programmiersprache **zustandsorientierte** oder **imperative Programmiersprache**.

Zur Illustration verwenden wir als Sprache **Java**.







Bei der Übersetzung (zur Übersetzungszeit) eines Java-Programms wird es in gewissem Rahmen auf seine Korrektheit geprüft.

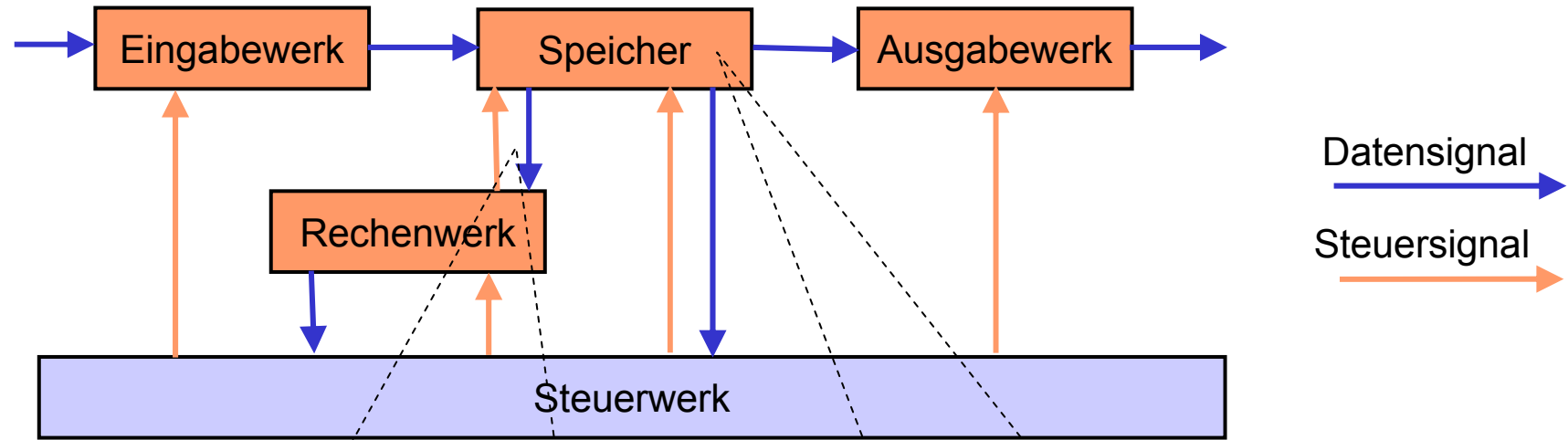
- Beispiel: Die syntaktische Struktur der Programms

Bei der Ausführung (zur Laufzeit) finden weitere Prüfungen statt, die erst beim Programmlauf bekannt werden können.

- Beispiel: Division durch 0 aufgrund einer falschen Benutzereingabe

Daher ist es wichtig, Übersetzungszeit und Laufzeit zu unterscheiden.





Zustandsübergang: Überführung des Inhalts des Speichers (eine oder mehrere Speicherzellen) in einen neuen Inhalt durch Anwendung einer Operation.

Zustand (der Ausführung, des (gedachten) Rechners): Momentaner Inhalt der Speicherzellen

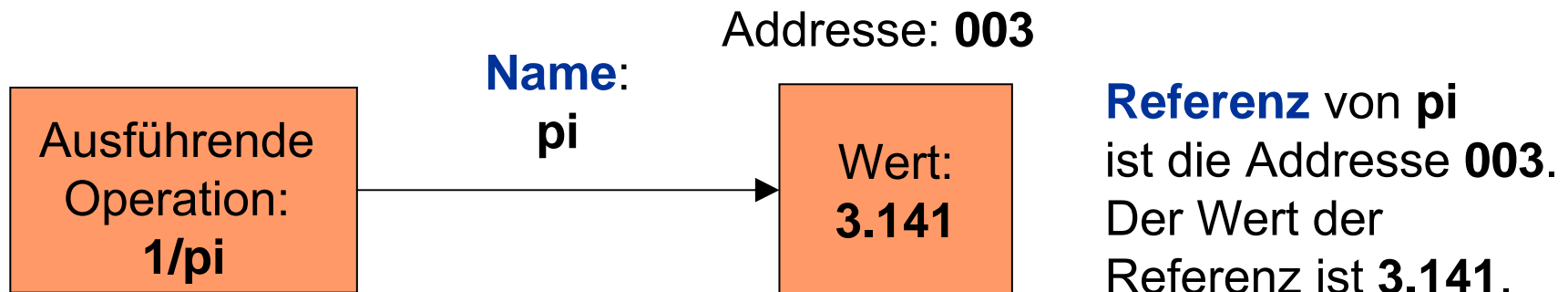
Eigenschaften einer Speicherzelle:

- Sie stellt einen Behälter dar, der über eine **Adresse** angesprochen wird und einen Inhalt (**Wert**) aufnimmt.
- Wir sagen: „Die Zelle mit Adresse **a** hat den Wert **w** zum Inhalt“.

Abstraktion für den Programmierer: Die Adresse interessiert nicht.

- Verwendung einer die Zelle identifizierenden Zugriffsfunktion: **Name**.
- Ergebnis der Berechnung der Adresse über die Zugriffsfunktion: **Referenz**.

Beispiel:



Ablauf des Zugriffs über Namen:

- Lesend
 - Vorgabe des Namens → Beschaffung der Referenz → Zugriff auf den Behälter → Lieferung des Wertes als Ergebnis
- Schreibend
 - Vorgabe des Namens → Beschaffung der Referenz → Zugriff auf den Behälter → Ersetzen des Wertes

Variablen sind Namen, die lesenden und schreibenden Zugriff erlauben.

Konstanten erlauben nur Lesezugriffe.

Beachte: Der schreibende Zugriff hat einen sogenannten Seiteneffekt, d.h. eine Veränderung des Zustands (im Speicher) des Rechners



Typen:

- Jede Variable oder Konstante besitzt einen Typ
- Vergleiche geometrischen Formen: Kreise, Dreiecke, Vierecke, Geraden...

Typsicherheit:

- Auf Variable oder Konstante können nur die dem Typ entsprechenden Operationen angewandt werden.
- Z.B. Operation „Radius-Berechnen“ sinnvoll nur bei Kreisen, nicht bei Dreiecken, Geraden, ...

Auffassung von Typen gemäß Abstrakten Datentypen (ADTs):

- Eine Grundmenge von Werten und auf ihnen definierte Operationen.

Somit lassen sich mit Typen manche Fehler ermitteln, bevor ein Schaden angerichtet wird.

- Verbesserte Korrektheit von Programmen (statische Typsicherheit).



... für Werte Variablen, Konstanten und Zuweisungen

Art	Syntaktische Form	Beispiele
unbenannte Konstante (Literal)	<Wert>	1, true, 'c'
Variablenvereinbarung	<Typ> <Name>;	int a;
lesender Zugriff	<Name>	a + 1
schreibender Zugriff (Zuweisung)	<Name> = <Wert>;	a = 1;
Variablenvereinbarung mit Vorbesetzung	<Typ> <Name> = <Wert>;	int a = 1;
Vereinbarung einer benannten Konstante mit Einmalzuweisung	final <Typ> <Name> = <Wert>;	final float pi = 3.14159;

<Wert> kann ein zu berechnender Ausdruck sein

Folge der Zugriffsfunktion:

- Es wird immer der (einzig verfügbare) aktuelle Wert gelesen.

Beispiel:

Wert von **a** nach der Anweisung

Zeit ↓

```
int a = 1;  
a = a + 1;  
a++;  
a = a;
```

Einfache Datentypen in Java:

Typ	Grundoperationen (Auswahl)	Beschreibung
boolean	<code>==, !=, !(not), &(and), (inkl. or), ^(exkl. or), &&(and), (inkl. or)</code>	Werte: <code>true, false</code> strikte Auswertung faule Auswertung
byte, short, int, long	<code>==, !=, <, >, <=, >=, +, -, ++ (unär), -- (unär), *, /, % (mod), << (left shift), >> (right shift)</code>	vorzeichenbehaftete Ganzzahlen von 8, 16, 32, 64 bit Länge. Einschränkungen beachten!
float, double	<code>==, !=, <, >, <=, >=, +, -, ++ (unär), -- (unär), *, /, % (mod)</code>	Gleitpunktzahlen von 32 bzw. 64 bit gemäß IEEE 754-1985. Einschränkungen beachten!
char		Einzelzeichen (16-bit Unicode)

■ Faule Auswertung:

```
y = 0; x = 1;
if (x == 1 || (x/y)) { //Faule Auswertung: hier kein Fehler
    y = 1;
}
```

■ Strikte Auswertung:

```
y = 0; x = 1;
if (x == 1 | (x/y)) { //Strikte Auswertung: hier DIV 0 Fehler!
    y = 1;
}
```

■ Überläufe:

```
x = 2147483647;
x = x+1;
Wert von x?
-2147483648
```

$$=((2^{32})/2)-1$$

Wieso einmal 8 am Ende und einmal 7?!

$$=-((2^{32})/2)$$

$((2^{32})/2)-1$
positive Zahlen +
 $((2^{32})/2)$
negative Zahlen +
0 = 2^{32} Zahlen

Zusammensetzung von Operanden und Operatoren. Sei F Ausdruck.

Es gilt zulässig(F) in einem Zustand z , wenn

- das Ergebnis nach den Regeln des jeweiligen ADT existiert;
- das Ergebnis arithmetischer Ausdrücke innerhalb der Genauigkeit der Rechnerarithmetik berechenbar ist;
- alle Operanden in F im Zustand z einen Wert besitzen.

Achte auf Unterschied zwischen fauler und strikter Auswertung:

- zulässig ($(x \neq 0) \ \&\& \ (1-x)/x > 1$),
- aber: nicht zulässig ($(x \neq 0) \ \& \ (1-x)/x > 1$) für $x=0$

Beachte:

- Ausdrücke haben einen Ergebnistyp!
- Auch Zuweisungen liefern Werte, nämlich das jeweilige Ergebnis der rechten Seite.
- Beispiel:

```
int i, j; i = j = 0;
```

```
//i und j wird der Wert 0  
//zugewiesen, da j = 0 den  
//Wert 0 liefert
```



(Statische) Typen:

- Der Typ eines Ausdrucks kann zur Übersetzungszeit bestimmt werden und leitet sich im wesentlichen aus den Typen der Teilausdrücke und des angewendeten Operators ab.
- Bei Zuweisungen müssen der Typ der Variablen und des Ausdrucks zusammenpassen.
- Analog bei anderen Operationen.

Beispiel:

```
int a = 0;  
float b = 2;  
boolean c;  
  
c = a == b;  
c = a * b;  
c = c * c;
```



- Gemäß Präzedenz, innerhalb dieser: links vor rechts.

$a + (b=2) * c * d;$

- Mit Klammern lässt sich die Präzedenz-Regelung umgehen.

$a + (b * c) ;$

- Strikte Evaluation (Berechnung der Operanden vor Anwendung des Operators) außer für $\&\&$, $\|$, $?$ (zu $?$ siehe später).

$(a \ \& \ (b=c+1) > 5) ;$

- Beachte bei Reihenfolge die Seiteneffekte! Diese spielen eine Rolle, wenn Teilausdrücke zum Beispiel Zuweisungen enthalten.

$(a=b-2) < 10 \ \& \ a > 4$



- Erforderlich, wenn geforderter Typ eines Wertes nicht mit tatsächlichem Typ übereinstimmt.
- Keine Freizügigkeit wegen Typsicherheit!
- Implizite Umwandlung bei Zuweisung $a = b$, wenn

Also: Zulässig, wenn Wertebereich nicht verkleinert wird ("Widening")

Typ von b	Typ von a	
byte	short, int, long, float, double	
short, char	int, long, float, double	
int	long, float, double	Genauigkeitsverluste möglich!
long	float, double	Genauigkeitsverluste möglich!
float	double	

Implizite Umwandlung in Ausdrücken, um definierten Operator anzuwenden.

- Bei ++, -- auf `byte`, `short`, `char` wird Wert zu `int`.

Explizite Umwandlung (Casting).

- Voranstellen des Zieltyps, z.B.

```
long a = 35; int b = (int) a;
```

wandelt evtl. unter Informationsverlust.

- Einschränkungen beachten, jedoch Wandlungen zwischen allen numerischen Typen erlaubt

Beispiele:

```
long a = 35;  
byte b = 10;  
int c = 20;  
a = (long) b;  
c = ++b;  
b = (byte) c;
```



Crash der Ariane 5 Rakete am 4. Juni 1996

- An overflow occurred in the Inertial Reference System (SRI) computer when converting a 64-bit floating point to 16-bit signed integer value.
- There was no error handler for that specific overflow. The default handler (wrongly) shut down the SRI unit.
- The standby SRI unit had previously shut itself down for the same reason. The hot SRI and the standby were running the same software.
- The shutdown caused the SRI to output a core dump on the bus. The main computer interpreted the core dump as flight data, causing such a violent trajectory correction that the rocket disintegrated.
- The SRI software had been ported from the previous generation rocket Ariane 4. The original software designers made a deliberate decision not to protect the conversion because overflow could not occur due to the physical characteristics of Ariane 4.
- The program that failed was a pre-flight program, and should not have been running during the flight. (In the Ariane 4 design, this program was allowed to run during flight to guard against some rare condition, but this was a poor decision in the first place; when the software was ported to Ariane 5 all justification for it was gone but nobody bothered to turn it off.)
- The investigation team concluded that the designers of the computer system put in protections against hardware faults but did not take into account software faults. Furthermore the SRI had not been tested with realistic Ariane 5 flight data, and there had been no integration tests of the SRI with the rest of the new rocket.



<http://tinyurl.com/3yb3g>



... oder nicht-primitive Typen

- Java behandelt alle zusammengesetzten Typen nach einem einheitlichen Mechanismus: Klassen.
- Wir betrachten zunächst nur drei Sonderfälle, die in allen Programmiersprachen vorkommen.
- Wir gehen zunächst nur auf grundlegende Formen ihrer Zusammensetzung ein.



Klasse `String` in Java

- Folge von Unicode-Zeichen.
- Darstellung:
 - `""` leerer String
 - `"hello"` normaler String
 - `"hello " + "girls!"` konkatenierter String
- Es gilt
 - `"hello " + "girls!" == "hello girls!"` liefert `true`



- Geordnete Menge typgleicher Variablen.
- Vereinbarung:
 <Typ> [] <Name>.
- Beispiele:
 `int[] ai;`
- Die Größe wird bei Erzeugung des Arrays festgelegt und kann dann nicht mehr geändert werden.
 - Die Erzeugung kann *bei* oder *nach* der Variablendeklaration stattfinden.
- Beispiel für die Erzeugung eines `int`-Arrays der Größe 10:

```
ai = new int[10];
```

```
// nach Deklaration
```

oder

```
int[] ai = new int[10];
```

```
// bei Deklaration
```

Immer dann notwendig, wenn Array-Größe
zum Deklarations-Zeitpunkt unbekannt



- Die Größe eines erzeugten Arrays kann mit `<Array-Name>.length` gelesen werden,

Beispiel:

```
ai.length == 10      // liefert (nach der Erzeugung von  
                      // oben) true
```

- Durch die Erzeugung gibt es die Zugriffsfunktionen:

```
ai[0], ai[1], ... , ai[ai.length-1]
```

Beispiel-Zugriffe:

```
ai[1] = ai[3 + 1];    // Schreibt Array-Element 1  
                      // und liest Array Element 4
```



Beispiel:

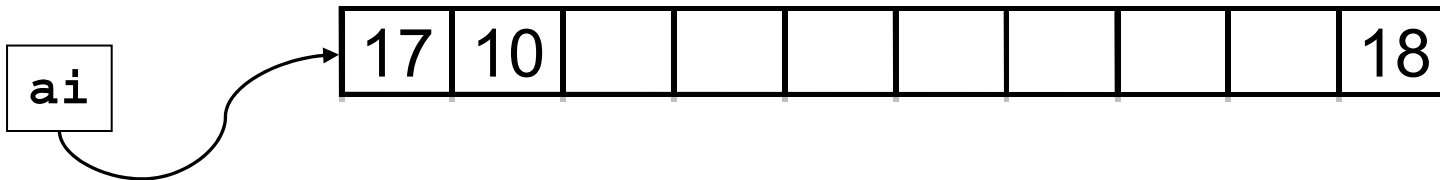
```
int ai[10];           //reserviert ein Array aus 10 Integeren.

ai[0] = 17;          //schreibt an das erste(!) Element eine 17.

ai[1] = ai.length;   //in das zweite Element wird die Länge (=10)
                    //geschrieben

ai[9] = 18;           //an das letzte Element wird 18 geschrieben,
                    //ebenso ai[ai.length-1]=18;

ai[ai.length] = 19;  //FEHLER!
```



- Der Typ eines Arrays kann beliebig sein (Klasse, primitiver Typ, Array-Typ).
- Daher sind mehr-dimensionale Arrays oder auch String-Arrays möglich:

- `int[][] aai; // Array aus int-Arrays`
- `String[] aString; // Array speichert Strings`

z.B. um die Elemente einer zwei-dimensionalen Matrix zu speichern...

- Auch im mehrdimensionalen Fall müssen die Arrays jeweils einzeln erzeugt werden:

```
aai = new int[2][]; // Erzeuge Array zur
                  // Speicherung von 2 int-Arrays
```

```
aai[0] = new int[1]; // Erzeuge erstes int-
                  // Array der Laenge 1
```

```
aai[1] = new int[2]; // Erzeuge zweites Array
                  // der Laenge 2
```



Fußballturnier mit 3 Mannschaften

■ Ergebnisse

1 vs. 2 2:0

1 vs. 3 1:1

2 vs. 3 1:0

In Java:

■ `int a[3][3];`

`a[0][1] = 2;`

`a[1][0] = 0;`

`a[0][2] = 1;`

`a[2][0] = 1;`

`a[1][2] = 1;`

`a[2][1] = 0;`

Was ist mit
`a[0][0]`, `a[1][1]`, `a[2][2]`?



- Bei der Erzeugung dürfen die Array-Größen aus Ausdrücken berechnet werden:

```
int a = b = 10;
```

```
String[] aString = new String[a + b];
```

- Es sind auch direkte Array-Initialisierungen möglich:

```
int[] quadrat = {1, 4, 9, 16, 25, 36, 49};
```

```
int[][] aai = { { 1, 2 }, { 3, 4 } };
```

- Bei nicht initialisierten Arrays werden die Elemente implizit auf Standardwerte gesetzt. Zum Beispiel 0 bei `int`, `short`, `byte`, `float`, `double` und `false` bei `boolean`.



<http://tinyurl.com/3afx6>



... in Java über class-Vereinbarung

- Mengen von sogenannten Feldern, die durch einen Bezeichner identifiziert werden

- Beispiel einer Klassen-Vereinbarung:

```
public class Point {  
    public static int x;           // Feld x wird deklariert.  
    public static float y,z;      // Felder y und z  
                                   // werden deklariert.  
}
```

- Aufgrund der Vereinbarung für Point existiert ein entsprechender Typ zur Deklaration von Variablen, Beispiel:

```
Point p;           // Variable p vom Typ Point  
                   // wird deklariert.
```



- Der Punkt . notiert den Feld-Zugriff.
- Somit Funktionen zum Zugriff auf die Felder von Point: `p.x`, `p.y`, `p.z`.

- Beispiel:

```
Point p; int xvar; // Deklariert Verbund p und
                  // Integer xvar
p.x = 1;          // Initialisiert Feld x in
                  // Point mit 1
xvar = p.x;        // Weist xvar den Wert 1 zu
```

- Klassen bieten noch viele weitere Möglichkeiten. Sie werden genauer in Kapitel 4 behandelt.



Anweisungen

- Eine Anweisung ist eine Abstraktion von (ein höherwertiges Äquivalent zu) den Operationen des realen Rechners.
- Anweisung wird daher als eine Ausführungseinheit betrachtet.
- Da imperatives Programmieren Zustandsübergänge bestimmt:
 - Das Ziel der Ablaufsteuerung ist die Steuerbarkeit von Schreib/Lese-Zugriffen von Variablen und Feldern anhand von vorausgehenden Zuständen.

Eine Anweisung ist entweder

- eine einfache Anweisung mit Terminator “;”
Beispiel: `int c = a*a + b*b;`
- oder ein Block, der eine Anweisungsfolge einschließlich Vereinbarung lokaler Variablen mit { und } klammert



- Ein Gültigkeitsbereich ist der Bereich, in dem ein (Variablen)-Name eine und dieselbe Referenz hat und somit einen Wert liefert.
 - Der Gültigkeitsbereich einer Variablen beginnt nach ihrer Deklaration und endet mit dem Ende des Blocks, in dem sie deklariert wurde.
 - Bei Wiederverwendung des Namens in einem inneren Block gilt dort der innere Name.
 - Der Gültigkeitsbereich einer Variablen kann somit zur Übersetzungszeit bestimmt werden.
- Beispiel für geschachtelte Blöcke mit Gültigkeitsbereichen:

```

{
    int a = 2 * 2;
    int b = 3 * 3;
    {
        int c = a + b;
    }
    c++;
}

```

Diagram illustrating the scope of variables in the example code:

- The innermost block (the curly braces around `int c = a + b;`) defines the scope for variable `c`. The text "c gültig hier" is placed next to this block.
- The middle block (the curly braces around the inner block and `c++;`) defines the scope for variable `b`. The text "b gültig hier" is placed next to this block.
- The outermost block (the curly braces around the entire code snippet) defines the scope for variable `a`. The text "a gültig hier" is placed next to this block.

- Die Gültigkeitsbereiche von Feldern einer Klasse unterscheiden sich wesentlich von denen von Variablen.

Reine Hintereinanderschreibung.

- Beispiel: Anweisungsfolge A1; A2; A3; ...

Anweisungsfolgen führen einen Anfangszustand z_0 in einen Endzustand z_e über.

- Beispiel: Vertauschungsblock

```
{ int h;  h = i;  i = j;  j = h; }
```

ij ij

ij h

ij h

ij h

ij h ij

zustandsdefinierende Variablen

Allgemeine Form:

```
if (<boolean-Ausdruck>) <Anweisung1> else <Anweisung2>
```

Beispiel: `if (i > j) max = i; else max = j;`

else-Teil kann entfallen \Rightarrow einseitige bedingte Anweisung

Beispiel: `if (i > j) { int h; h = i; i = j; j = h; }`

Bedingter Ausdruck mit Ergebnis:

- Mit dem ersten Beispiel gleichbedeutend ist

```
max = ( i > j ? i : j );
```

- ? heißt bedingter Operator und hat im Gegensatz zu if ein Ergebnis.

- Ähnlichkeit zu Haskell:

```
max i j = if i > j then i else j
```

mit Funktionsergebnis.



Kaskaden von bedingten Anweisungen sind möglich. Beispiel:

```
int signum;           // Berechne das Vorzeichen von a
if (a > 0)             //(a weiter oben definiert)
    signum = 1;
else if (a < 0)
    signum = -1; else signum = 0;
```

Beachte: eine geeignete Klammerung und Einrückung macht den Code besser lesbar und vermeidet Programmierfehler.

Hier besser:

```
int signum;           // Berechne das Vorzeichen von a
    if (a > 0)         // (a weiter oben definiert)
        signum = 1;
    else if (a < 0)
        signum = -1;
    else signum = 0;
```



Die spezielle Form der Kaskade (A_i Anweisungsfolge)

```
if (i == k1) A1;  
else if (i == k2) A2;  
else if (i == k3) A3;  
...  
else A0;
```

kann übersichtlicher als `switch`-Anweisung geschrieben werden

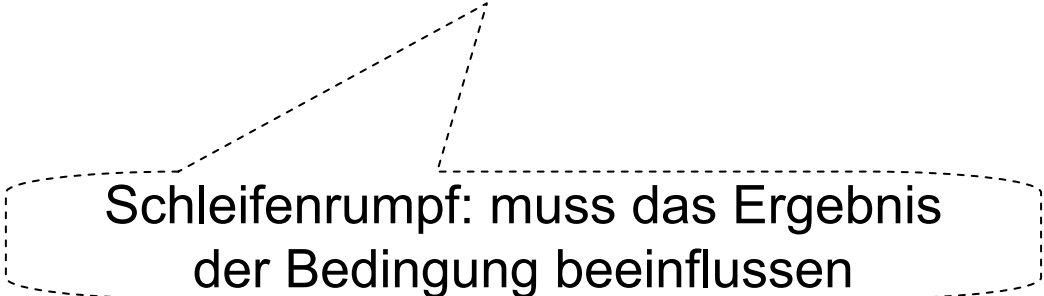
// i muss vom Typ `byte`, `short`, `int` oder `char` sein

```
switch (i) {  
    case k1: A1; break; // ki muessen Literale  
    case k2: A2; break; // vom Typ byte, short, int, char  
                        // sein,  
    case k3: A3; break; // und i muss vom entsprechenden Typ  
                        // sein.  
    ...  
    default: A0;  
}
```



Grundgedanke der Steuerung: Modifikation der Werte von Zustandsvariablen, bis eine Zielbedingung erfüllt ist.

Standardform: **while** (<boolean-Ausdruck>) <Anweisung>



Schleifenrumpf: muss das Ergebnis der Bedingung beeinflussen

Anmerkungen:

- **while**-Schleifen werden u.U. niemals durchlaufen (die Evaluierung der Bedingung <boolean-Ausdruck> vor Betreten des Schleifenrumpfes ist dann **false!**).
- Abbruch jedes Blocks und damit auch einer **while**-Schleife möglich mit **break**.
- Terminator von <Anweisung> ist zugleich Terminator der **while**-Anweisung.



Beispiel für eine **while**-Schleife:

kgV(a, b, c) dreier ganzer Zahlen

```
{ // a, b, c weiter oben definiert...  
  int aRes = a, bRes = b, cRes = c;  
  
  while (aRes != bRes || bRes != cRes) {  
    if (aRes < bRes) {  
      aRes = aRes + a;  
    }  
    else if (bRes < cRes) {  
      bRes = bRes + b;  
    }  
    else {  
      cRes = cRes + c;  
    }  
  }  
  // Hier gilt: aRes == bRes == cRes == kgV(a, b, c)  
}
```



Mindestens einmaliger Durchlauf wird garantiert mit der do-while-Schleife (Evaluierung der Bedingung zum Abschluss des Schleifenrumpfes!):

```
do <Anweisung> while (<boolean-Ausdruck>);
```

Bei Iteration über einen Wertebereich wird anstelle der while-Schleife die bequemere Form der for-Schleife verwendet:

```
for (<Initialausdruck>; <boolean-Ausdruck>;<Inkrementausdruck>)  
    <Anweisung>
```

Beispiel: Aufsummieren der Werte einer Reihung `int [] a`:

```
int i, summe;
```

```
for (i = 0, summe = 0; i < a.length; i++) {  
    summe = summe + a[i];  
}
```

Eine for-Schleife kann (im Wesentlichen) nach folgendem Muster auf eine semantisch äquivalente while-Schleife abgebildet werden:

```
for (<Initialausdruck>; <boolean-Ausdruck>;  
    <Inkrementausdruck>)  
    <Anweisung>
```

wird zu:

```
<Initialausdruck>;  
while (<boolean-Ausdruck>) {  
    <Anweisung>;  
    <Inkrementausdruck>;  
}
```

Ähnliches gilt für do-while-Schleifen.



Beispiel: Algorithmus zur Berechnung der reflexiven, transitiven Hülle einer Relation

```
{
boolean[][] a;           // Adjazenzmatrix
boolean[][] s;           // Ergebnismatrix
... kopiere a nach s    // nicht detailliert
int i, j, k;

                        // Reflexivitaet von s herstellen.
for (i = 0; i < s.length; i++) { s[i][i] = true; }
for (i = 0; i < s.length; i++) {
    for (j = 0; j < s.length; j++) {
        for (k = 0; k < s.length; k++) {
            // Transitivitaet: von Knoten i nach
            // j ueber Knoten k
            s[i][j] = s[i][j] ? true : s[i][k] & s[k][j];
        }
    }
}
// s ist jetzt die Adjazenzmatrix der reflexiven,
// transitiven Huelle von a
}
```



- Eine Methode ist ein *benannter* Anweisungsblock, dessen Inhalt durch eine Ein/Ausgabe-Schnittstelle gekapselt ist und dessen Funktionalität evtl. an mehreren Stellen verwendet werden kann. Der Anweisungsblock heißt auch *Methodenrumpf*.
- Eine Methode hat ein Ergebnis von einem festen Typ (der Ausgabe-Teil der Schnittstelle).
- Die *Parameter* der Methode repräsentieren die Eingabe-Schnittstelle. Es handelt sich dabei um eine feste Liste von n getypten Variablen. Sie gelten im Methodenrumpf als lokale Variablen.



Vereinbarung:

```
public static <Ergebnistyp> <Methodenname>  
    (<Parametertyp 1> <Parametername 1>, ... ,  
     <Parametertyp n> <Parametername n>)
```

**Signatur oder
Methodenkopf**

```
{
```

```
    // Anweisungen hier.
```

```
}
```

Methodenrumpf

Methoden ohne Ergebnis sind auch möglich:

- Vereinbarungen wie zuvor, aber mit `void` als <Ergebnistyp>
- In Java werden Methoden *immer* in Klassen definiert!



- Bei eigenständigen Programmen (keine Applets) muss es eine main-Methode geben, die so aussieht:

```
public static void main (String args[]) { ...
```

- Jede Klasse kann eine (einzige) solche main-Methode enthalten; sie wird ausgeführt, wenn der entsprechende Klassenname beim „java“- Kommando genannt wird.
- Klassen können getrennt übersetzt werden.



- Ein Methodenaufruf hat die Form
 <Methodenname> (<Argument 1>, ... , <Argument n>)
- Bei der Abarbeitung des Methodenaufrufs werden die Anweisungen im Methodenrumpf (der Methode mit dem angegebenen Namen) ausgeführt.
- <Argument 1> bis <Argument n> sind dabei Ausdrücke, die zuvor berechnet werden und deren Werte den Parametern 1 bis n der Methode zugewiesen werden.
- Da die Parameter typisiert sind, müssen die jeweiligen Ausdruckstypen zu den Parametertypen passen (gleicher Typ oder implizit konvertierbar).
- Im Methodenrumpf sind nur die dort deklarierten Variablen und die Parameter sichtbar.
- Da die Werte der Argumentausdrücke direkt an die Parameter zugewiesen werden, spricht man von **Wertaufruf**.



- Bei Methoden mit echtem Ergebnistyp (ungleich `void`) ist der Methodenaufruf ein Ausdruck, dessen Typ identisch ist mit dem Ergebnistyp der Methode.
- Der Rückgabewert der Methode wird im Methodenrumpf mit `return <Wert>;` zurückgegeben.
- Der Typ des Werts muss natürlich zum Ergebnistyp der Methode passen.



```
public static <Ergebnistyp> <Methodenname>
    (<Parametertyp 1> <Parametername 1>, ... ,
     <Parametertyp n> <Parametername n>)
{
    Parametername 1 = Argument 1;
    ...
    Parametername n = Argument n;
    // Anweisungen hier.
    verdeckteErgebnisvariable = Wert;
}
```

Eine Methode, die ihren Parameterwert um 1 erhöht...

```
public class MethodenTest {  
    // Methode erhoehe gibt int zurueck und  
    // nimmt int als Argument auf  
    public static int erhoehe(int x) {  
        // Parameter x wird verwendet  
        int y = x + 1;  
        // Der berechnete Wert (x + 1) wird  
        // zurueckgegeben  
        return y;  
    }  
  
    // Eine argument- und parameterlose Methode...  
    public static void andereMethode() {  
        int z = 3;  
        // erhoehe wird mit Argumentwert 3 aufgerufen  
        int zPlusEins = erhoehe(z);  
        // zPlusEins ist jetzt 4;  
    }  
}
```



kgV(a, b, c) als Methode in einer Klasse KgV:

```
public class KgV
{
    public static int kgV(int a, int b, int c) {
        int aRes = a, bRes = b, cRes = c;

        while (aRes != bRes || bRes != cRes) {
            if (aRes < bRes) {
                aRes = aRes + a;
            }
            else if (bRes < cRes) {
                bRes = bRes + b;
            }
            else {
                cRes = cRes + c;
            }
        }
        return aRes;
    }
}
```



Algorithmus zur Berechnung der transitiven reflexiven, Hülle einer Relation als Methode in der Klasse RTHuelle:

```
public class RTHuelle
{
    public static boolean[][] berechneRTHuelle(boolean[][] a) {
        // Ergebnismatrix
        boolean[][] s = new boolean[a.length] [];
        int i, j, k;

        // kopiere a nach s
        for (i = 0; i < a.length; i++) {
            s[i] = new boolean[a.length];
            for (j = 0; j < a.length; j++) s[j][i] = a[j][i];
        }
        // Reflexivitaet von s herstellen.
        for (i = 0; i < s.length; i++) { s[i][i] = true; }
        for (i = 0; i < s.length; i++) {
            for (j = 0; j < s.length; j++) {
                for (k = 0; k < s.length; k++) {
                    s[i][j] = s[i][j] ? true : s[i][k] & s[k][j];
                }
            }
        }
        return s;
    }
}
```



Rekursion muss mit Hilfe von Methoden formuliert werden.

Gegeben:

- Reihung `int[] a = new int[n]`, aufsteigend sortiert.
- Ganze Zahl `x` (Suchwert).

Gesucht:

- Angabe, ob `x` in `a` vorkommt.

Suchwert

sortierte Folge

Methode für rekursive Lösung der Binärsuche:

```
public static boolean sucheRek (int x, int[] a,
                                int u, int o) {
    int m = (u + o) / 2;
    if (u > o)
        return false;
    else if (x == a[m])
        return true;
    else if (x < a[m])
        return sucheRek(x, a, u, m-1);
    else
        return sucheRek(x, a, m+1, o);
}
```

zu durchsuchender Teil
von a (wg. Rekursion!)



Aufruf von außen, zum Beispiel so:

```
boolean found = sucheRek(x, a, 0, a.length - 1);
```

- Es ist typisch für die Rekursion in imperativen Sprachen, dass einige Parameter nach außen getragen werden müssen, die nur der Steuerung der Rekursion dienen.
- Dies sind hier die Parameter `int u`, `int o`.

Daher übliche Lösung:

- Die rekursive Prozedur wird „versteckt“.
- Für die Sicht von außen wird die Prozedur vereinfacht.

Beispiel:

```
public static boolean suche(int x, int[] a) {  
    return sucheRek(x, a, 0, a.length - 1);  
}
```

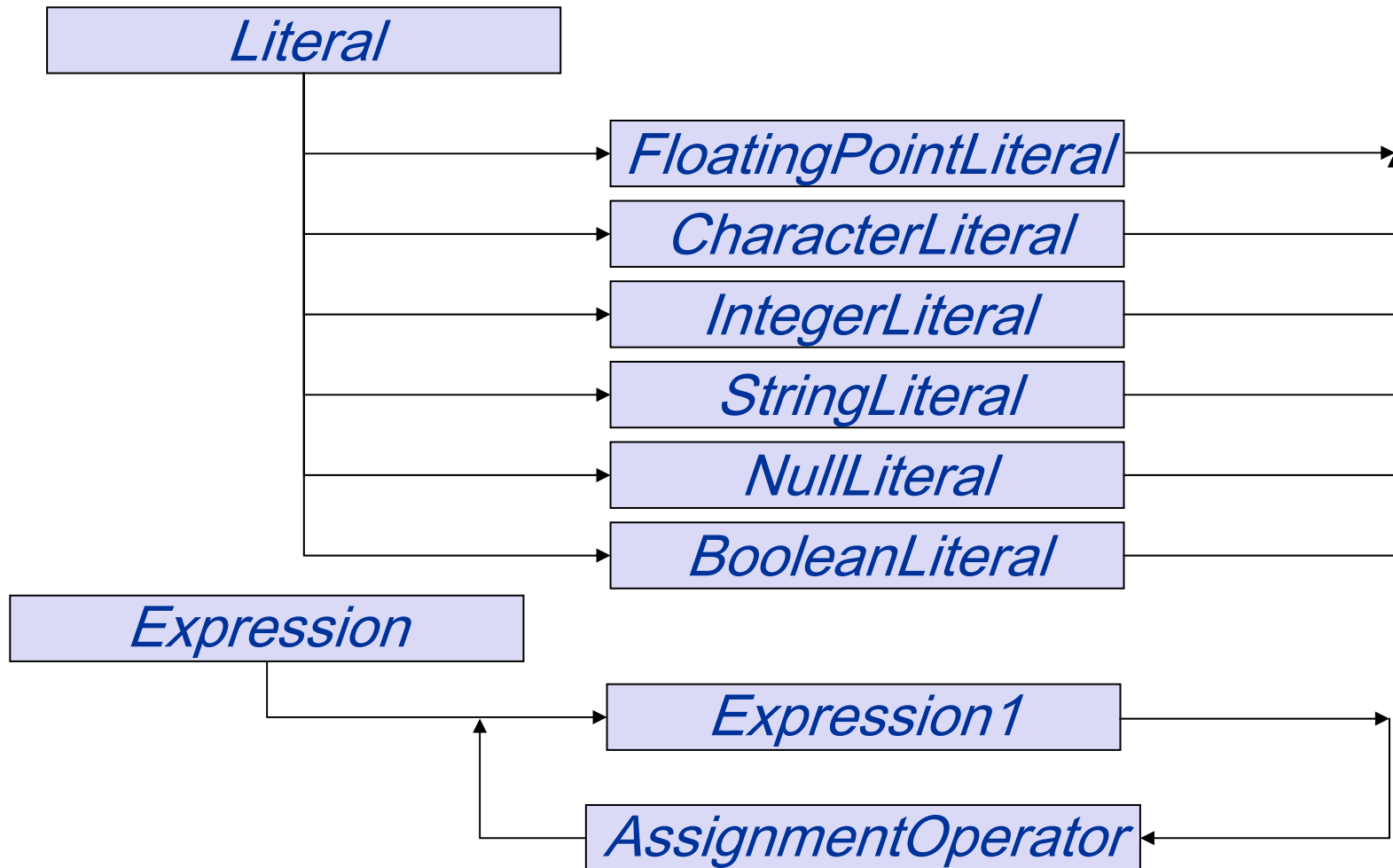


- Java ist als Programmiersprache standardisiert und spezifiziert.
- Die Syntax ist ein wichtiger Teil der Sprachspezifikation, aber natürlich nicht der einzige.
- Als Programmierer ist es manchmal wichtig, die „Grenzen“ einer (Programmier) Sprache genau zu kennen und das erfordert das Lesen und Verstehen der Spezifikation.

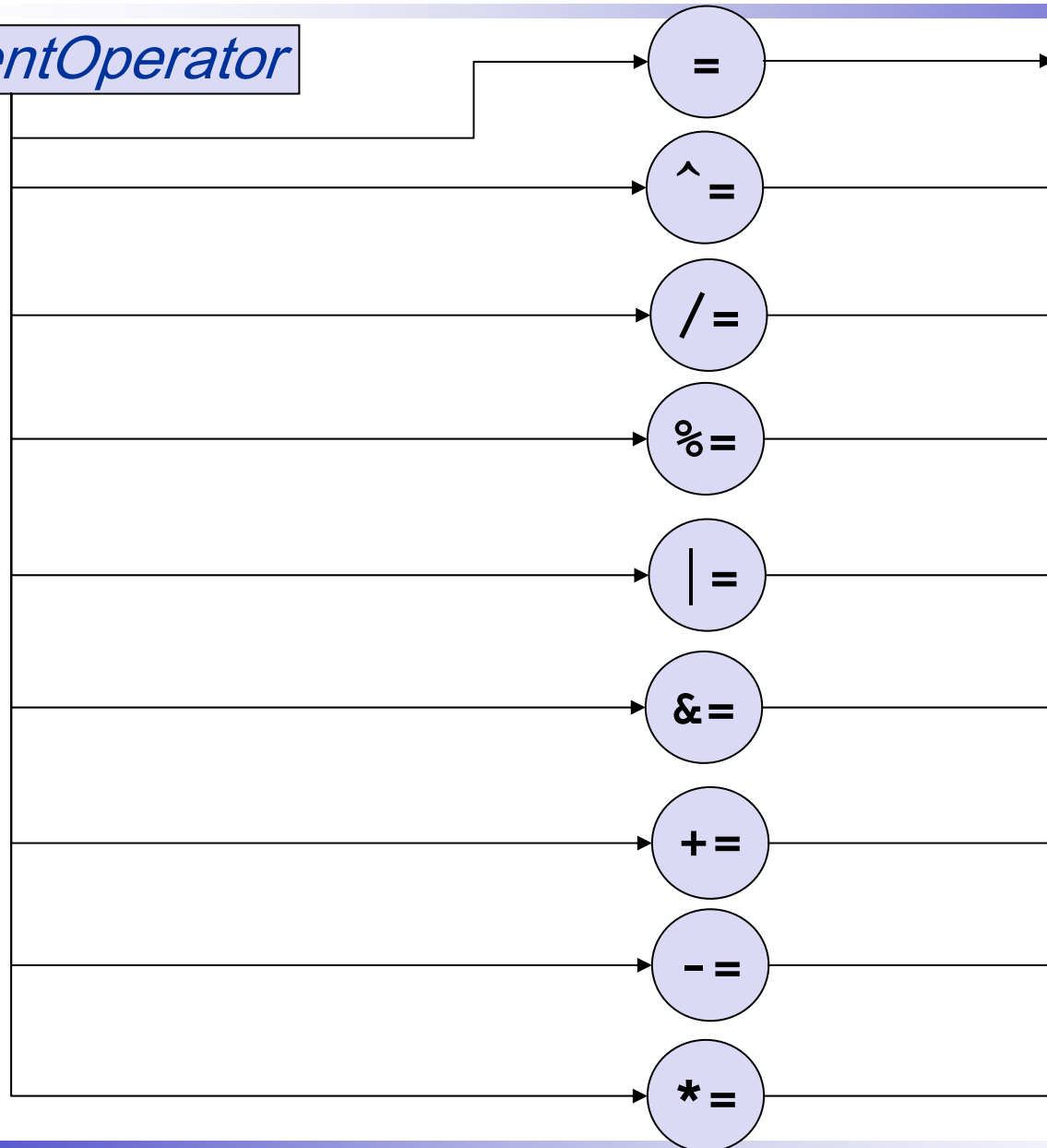
Die Java-Spezifikation ist erreichbar unter:

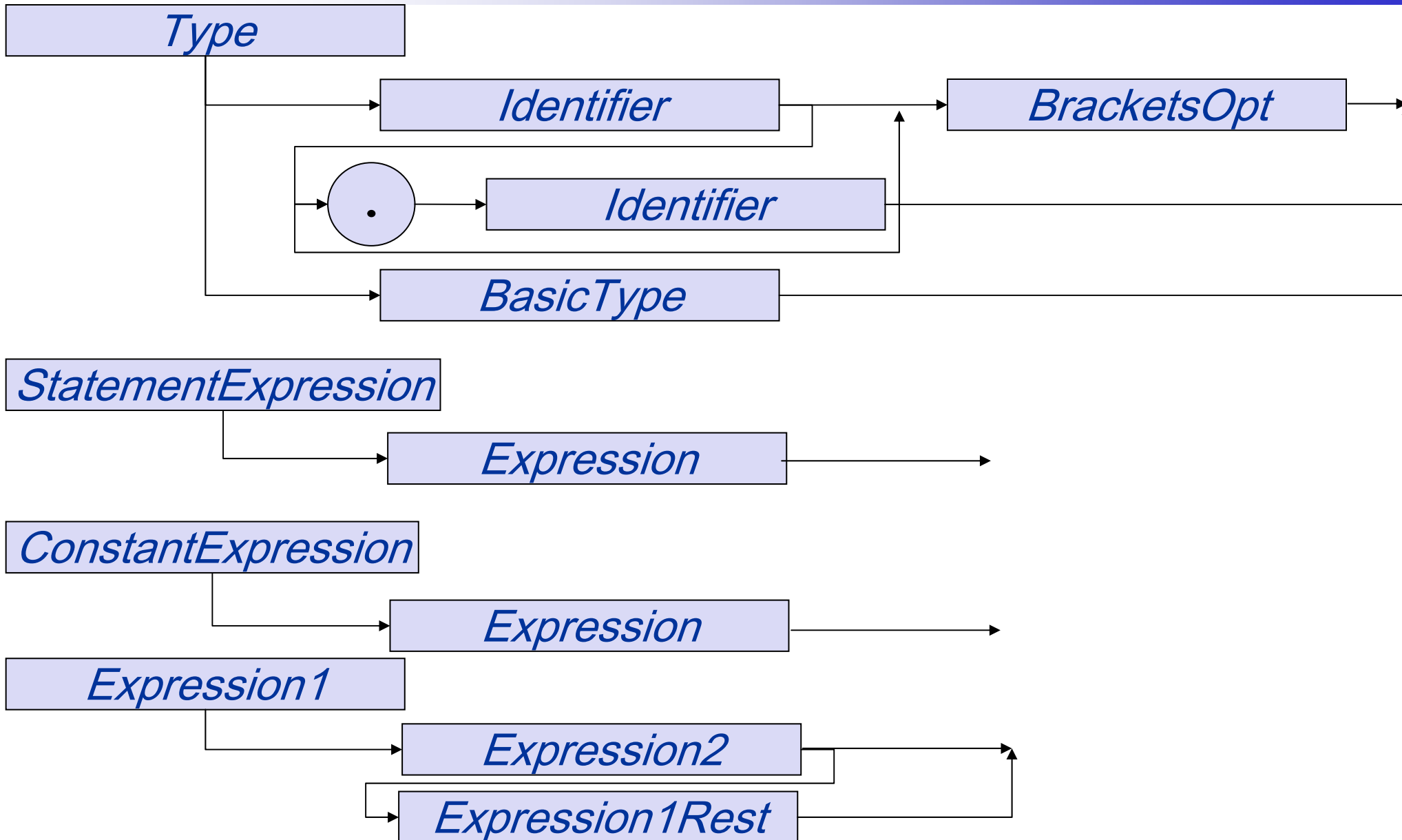
<http://java.sun.com/docs/books/jls/>

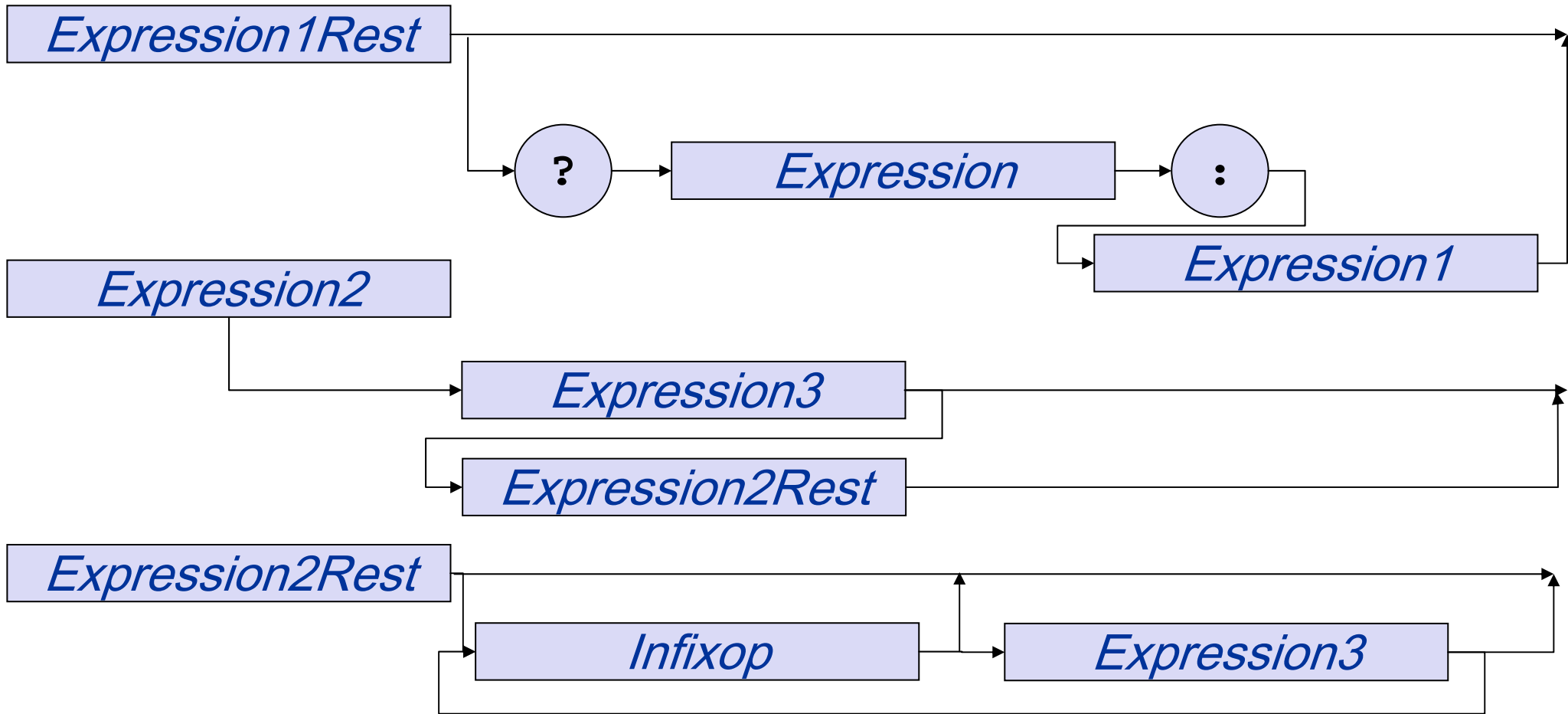




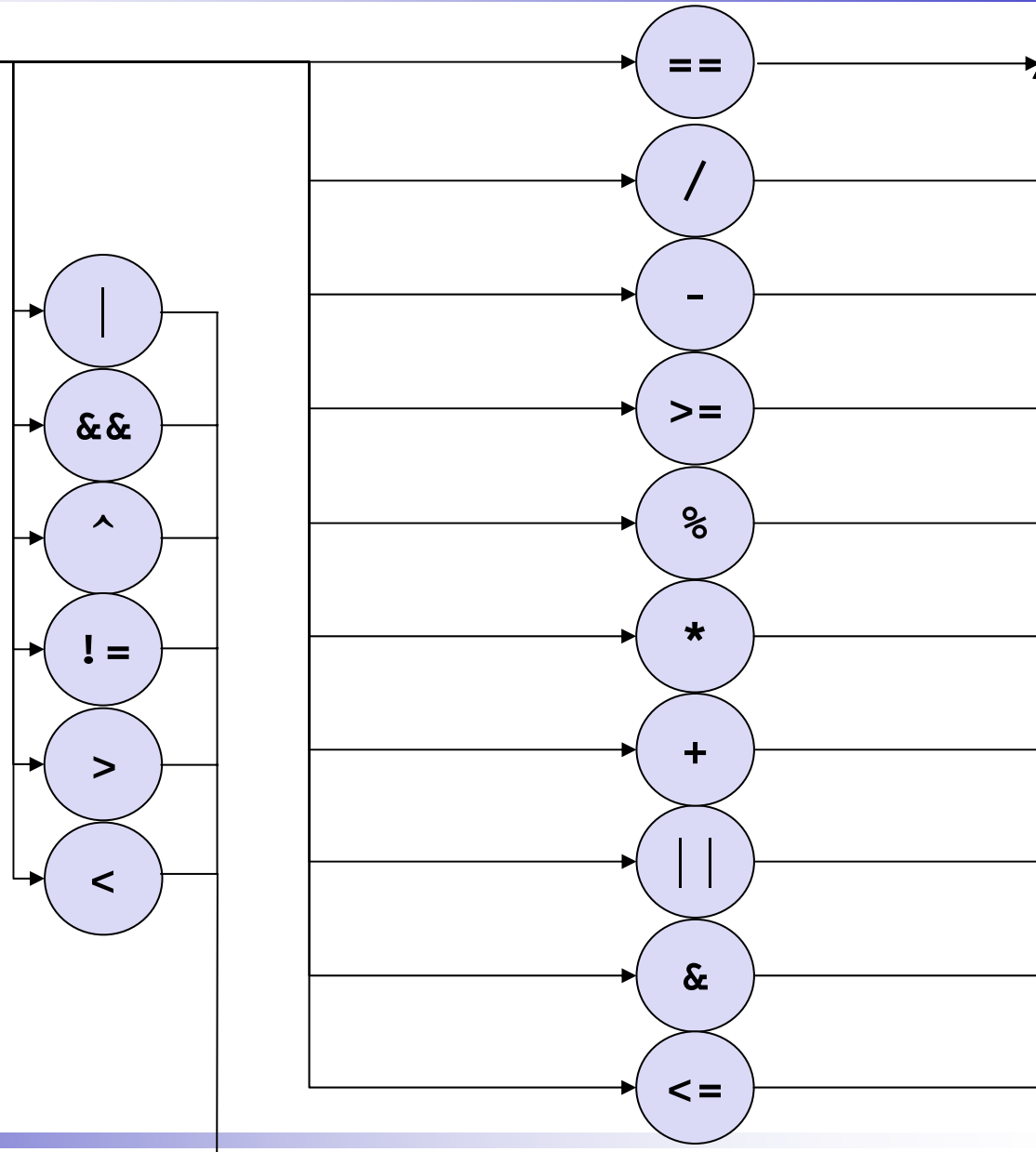
AssignmentOperator

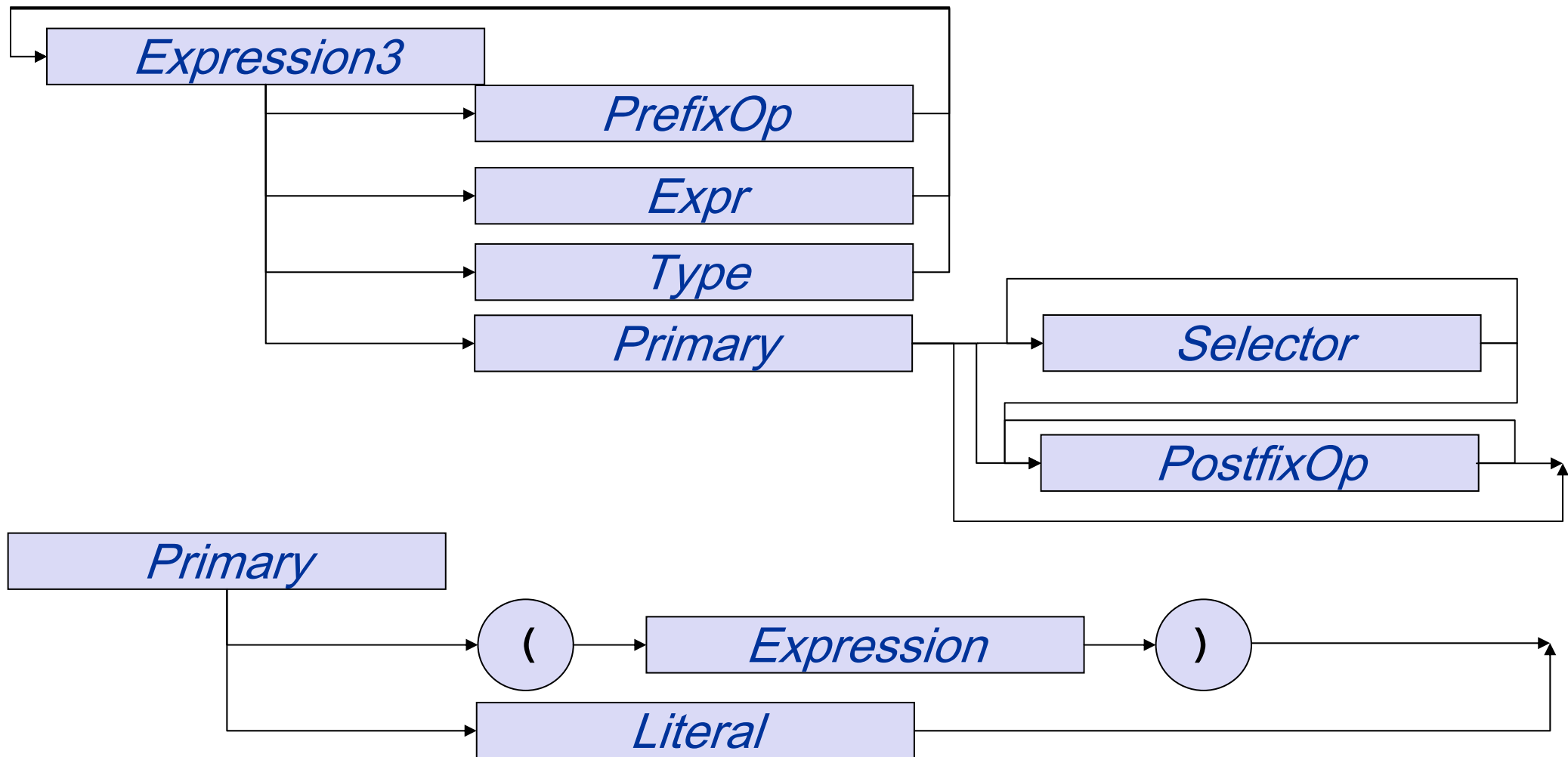


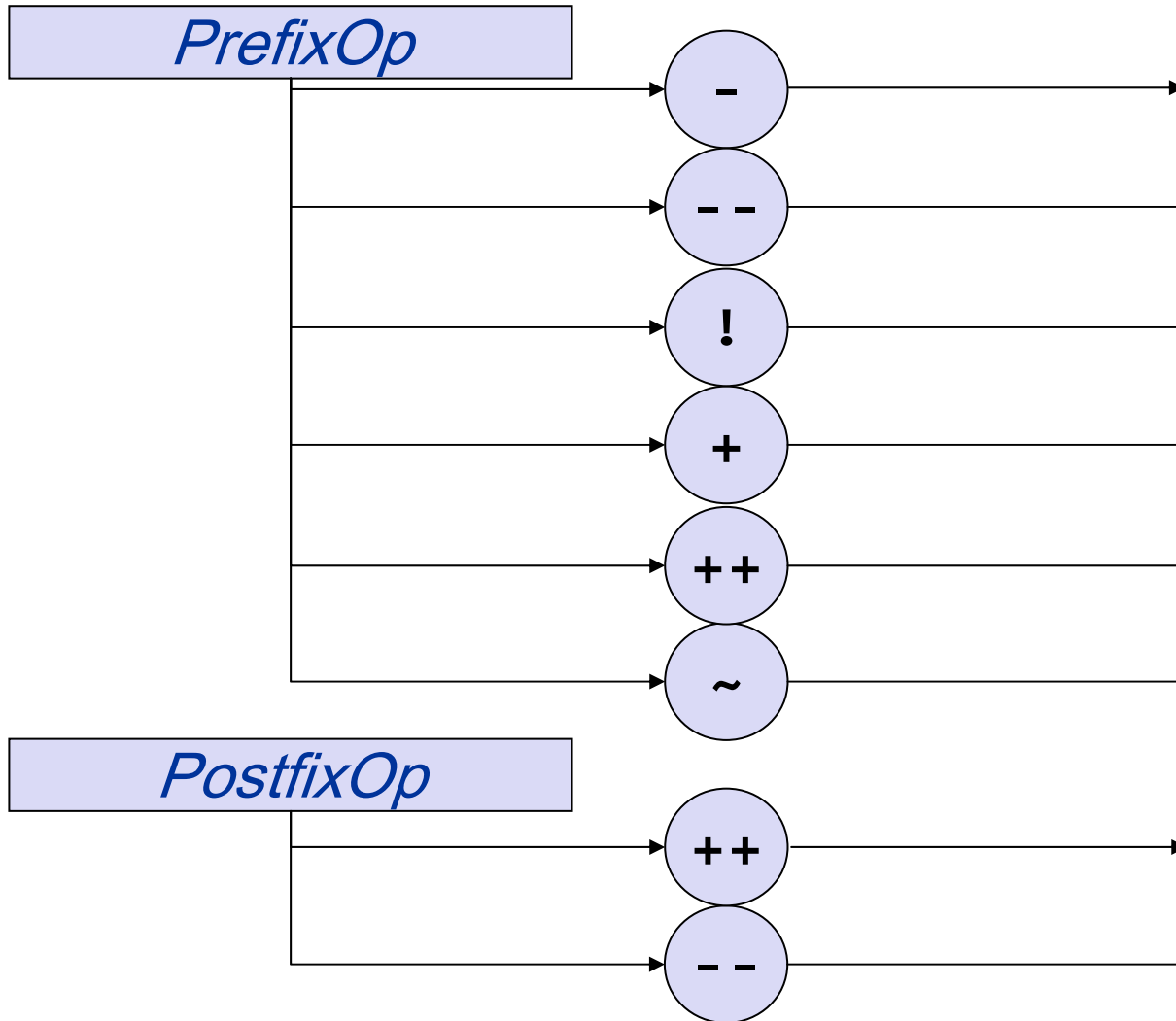


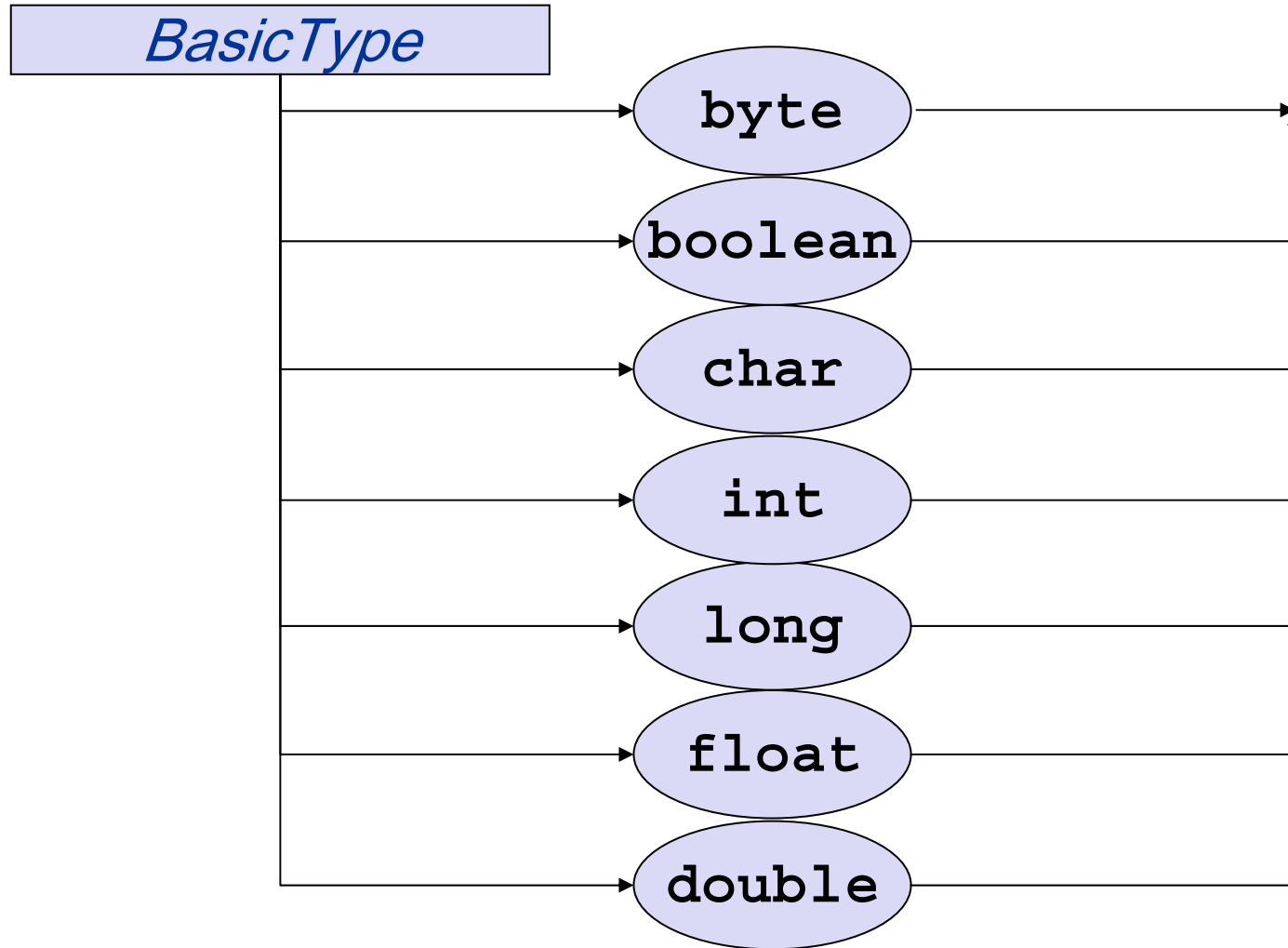


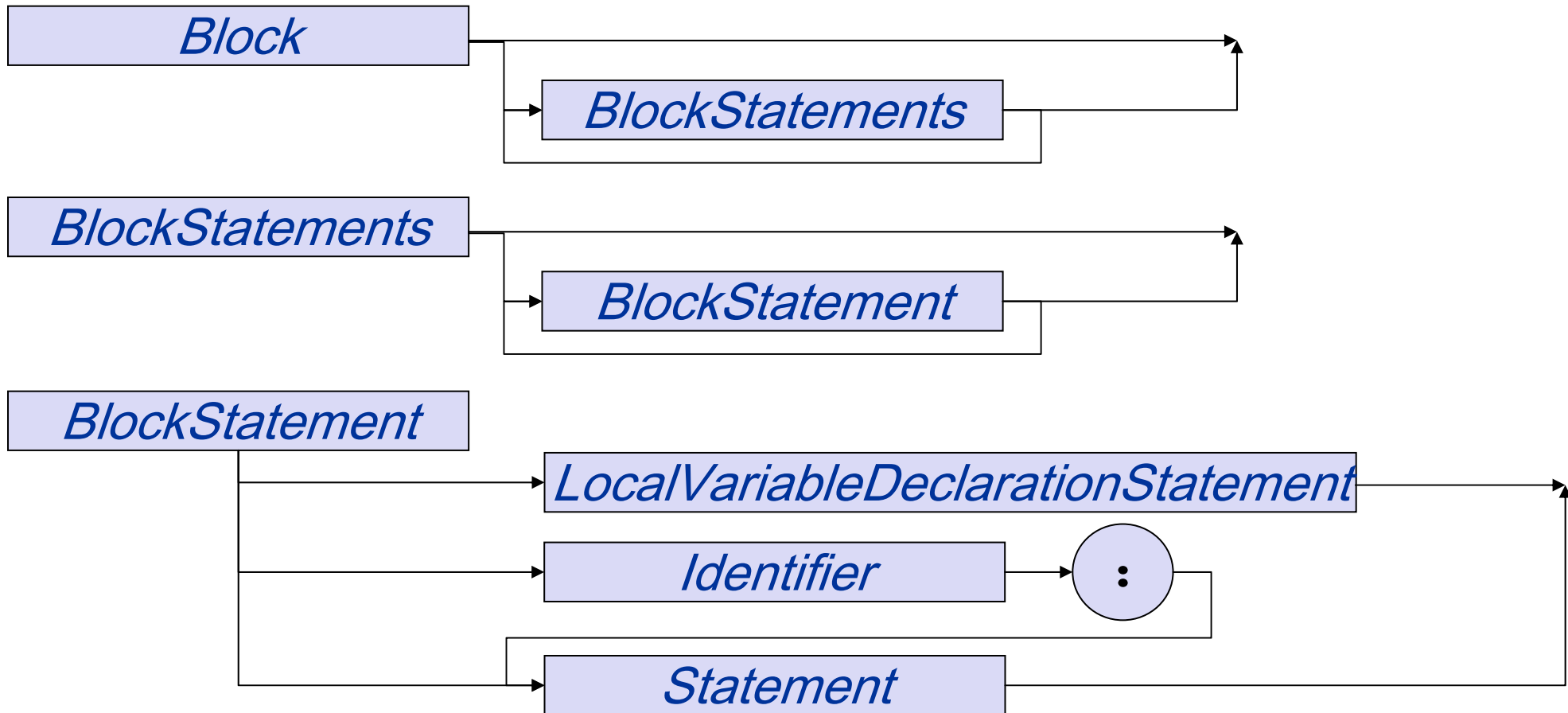
Infixop



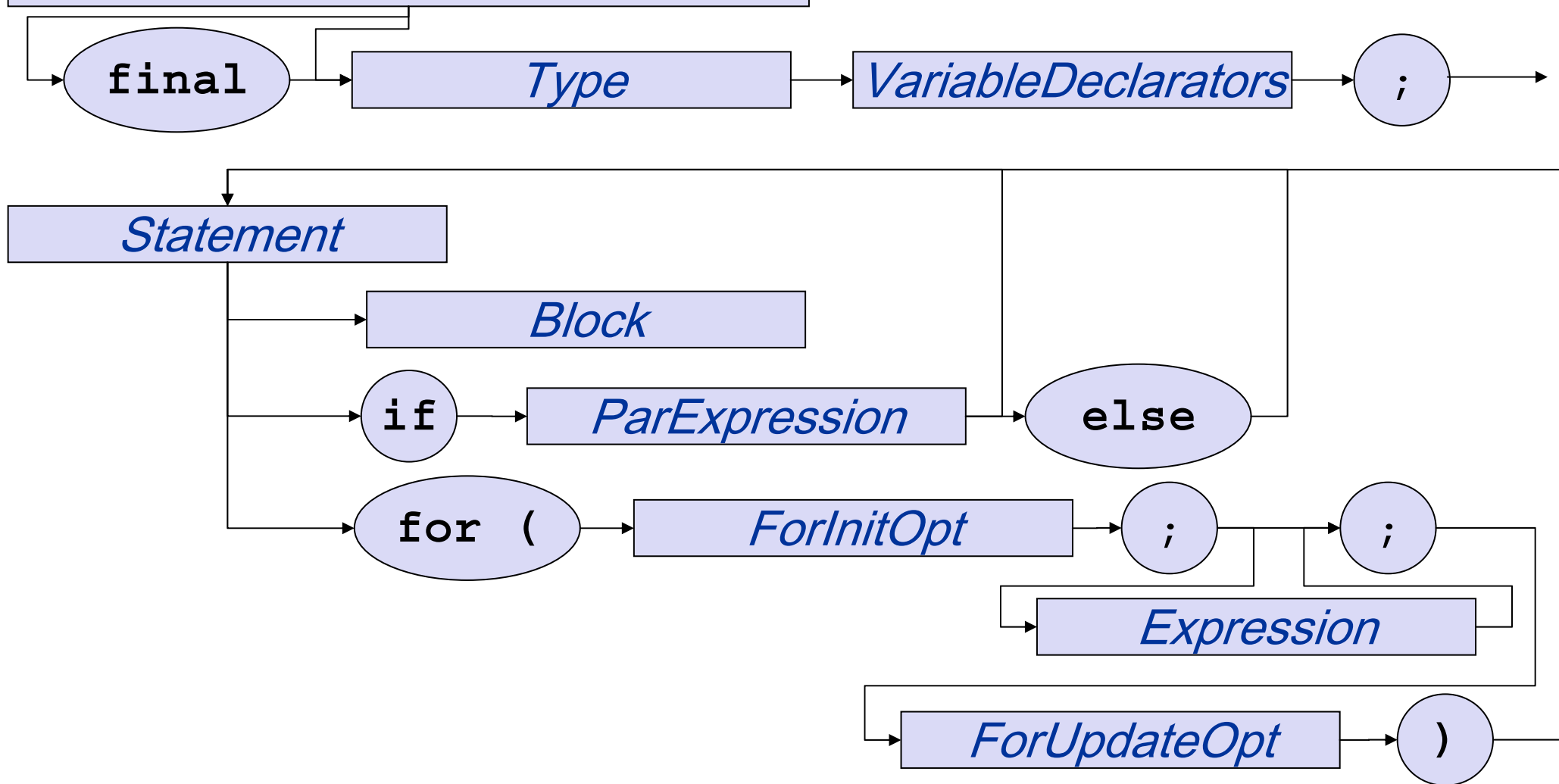


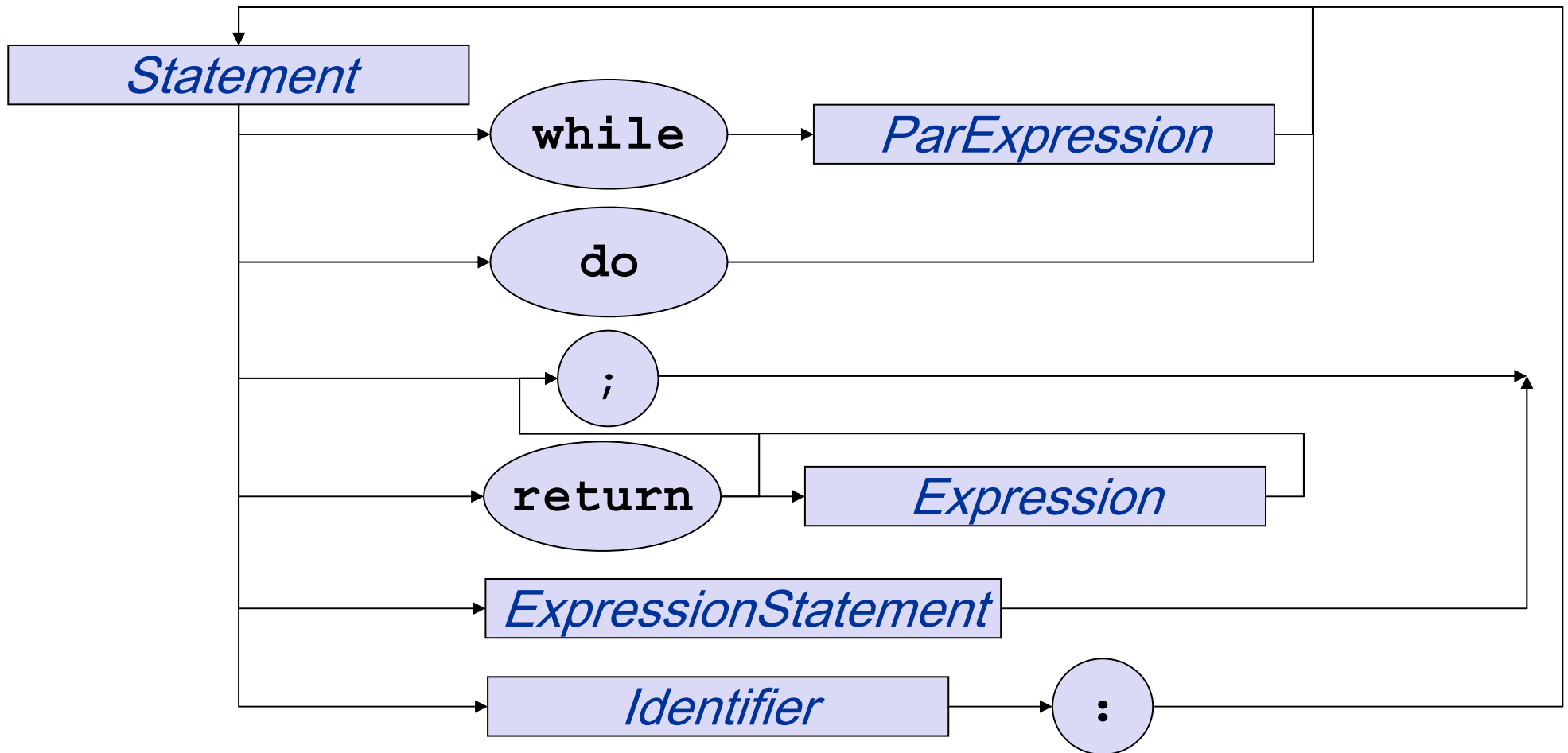


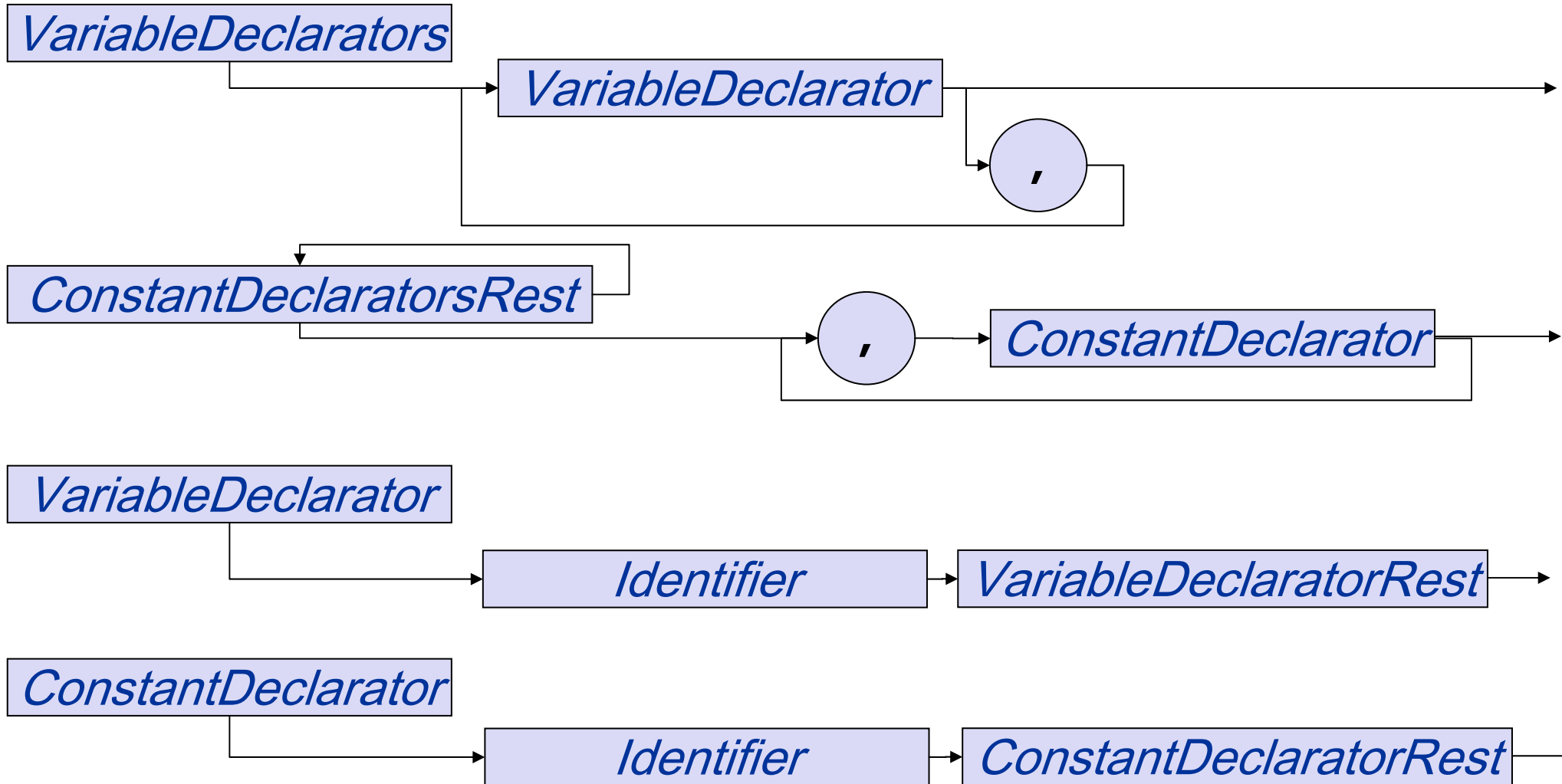


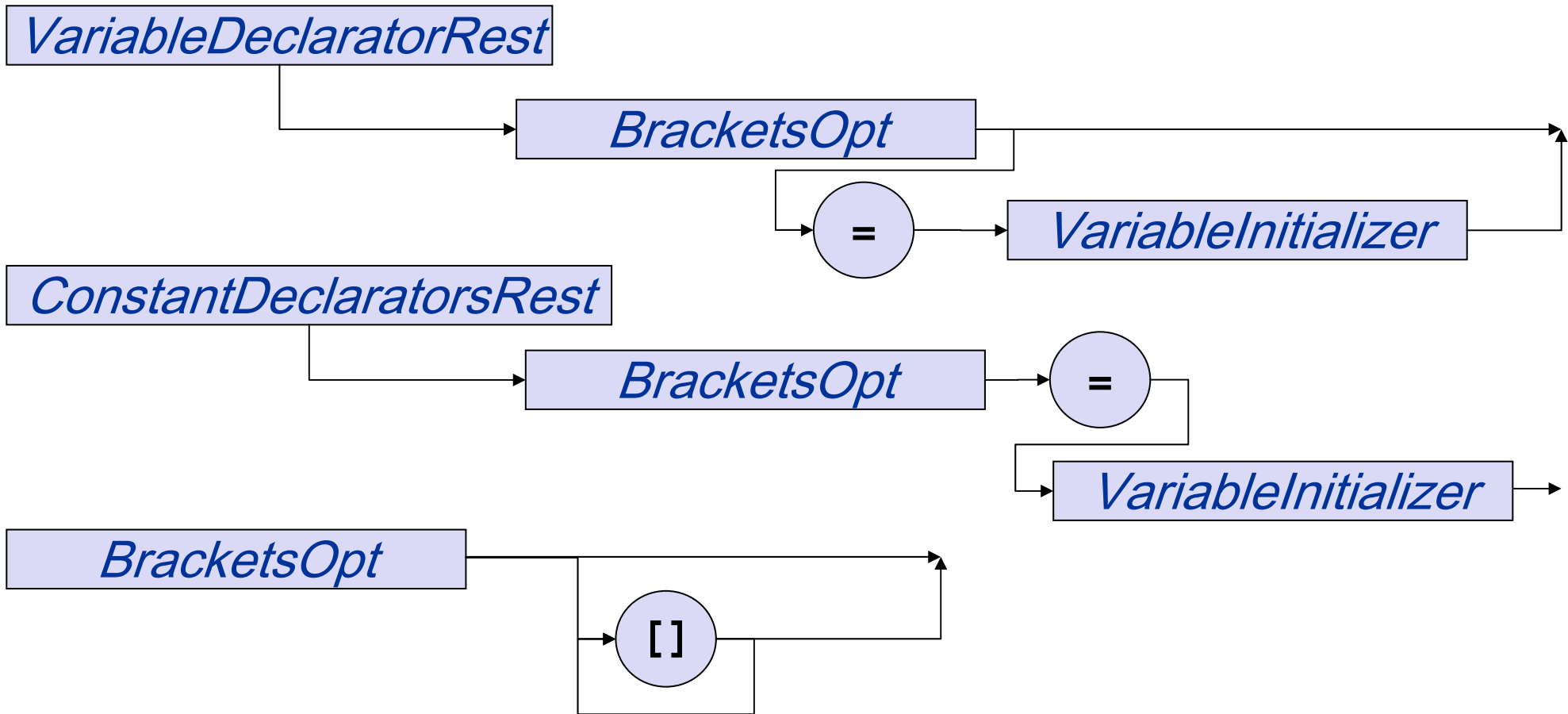


LocalVariableDeclarationStatement





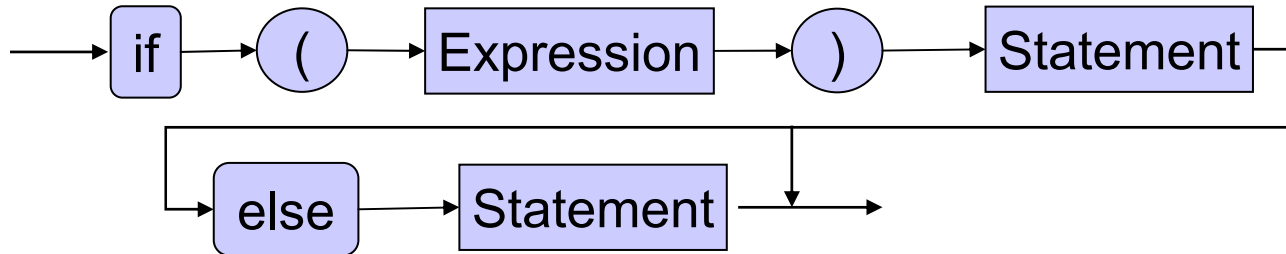




Syntax-Graphen sind anschaulicher als Syntax-Diagramme, z.B.

if *ParExpression Statement* [**else** *Statement*]

ifStatement



for (*ForInitOpt* ; [*Expression*] ; *ForUpdateOpt*) *Statement*

forStatement

