3 Syntax von Programmiersprachen

Syntax (“Lehre vom Satzbau”):

- formale Beschreibung des Aufbaus der “Worte” und “Sätze”, die zu einer Sprache gehören;
- im Falle einer Programmier-Sprache Festlegung, wie Programme aussehen müssen.

Hilfsmittel bei natürlicher Sprache:

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahme-Listen;
- Sprach-“Gefühl”.

Hilfsmittel bei Programmiersprachen:

- Listen von Schlüsselworten wie `if`, `int`, `else`, `while` ...
- Regeln, wie einzelne Worte (Tokens) z.B. Namen gebildet werden.

  Frage:
  Ist `x10` ein zulässiger Name für eine Variable?
  oder `_ab$` oder `A#B` oder `0A?B` ...
- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

  Frage:
  Ist ein `while`-Statement im `else`-Teil erlaubt?

- Kontextbedingungen.

  Beispiel:

  Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

$\Longrightarrow$ formalisierter als natürliche Sprache

$\Longrightarrow$ besser für maschinelle Verarbeitung geeignet

Semantik (“Lehre von der Bedeutung”):

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine Bedeutung, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑Information)

- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine Bedeutung ...

Die Bedeutung eines Programms ist

- alle möglichen Ausführungen der beschriebenen Berechnung (↑operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben (↑denotationelle Semantik).

Die Bedeutung eines Programms ist

- alle möglichen Ausführungen der beschriebenen Berechnung (↑operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben (↑denotationelle Semantik).

Achtung!

Ist ein Programm syntaktisch korrekt, heißt das noch lange nicht, dass es auch das "richtige" tut, d.h. semantisch korrekt ist !!!

3.1 Reservierte Wörter

- `int`

  → Bezeichner für Basis-Typen;

- `if`, `else`, `while`

  → Schlüsselwörter aus Programm-Konstrukten;

- `(`,`)`, `"`,`'`, `{`,`}`, `,`,`;`

  → Sonderzeichen.

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

$$\text{letter} ::= \texttt{\$} \mid \texttt{_} \mid \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}$$

$$\text{digit} ::= \texttt{0} \mid \ldots \mid \texttt{9}$$

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

$$\text{letter} ::= \texttt{\$} \mid \texttt{_} \mid \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}$$

$$\text{digit} ::= \texttt{0} \mid \ldots \mid \texttt{9}$$

- letter und digit bezeichnen Zeichenklassen, d.h. Mengen von Zeichen, die gleich behandelt werden.
- Das Symbol "|" trennt zulässige Alternativen.
- Das Symbol "..." repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

Schritt 2: Angabe der Anordnung der Zeichen:

$$\text{name} \quad ::= \quad \text{letter} \; (\; \text{letter} \mid \text{digit} \;)^*$$

- Erst kommt ein Zeichen der Klasse letter, dann eine (eventuell auch leere) Folge von Zeichen entweder aus letter oder aus digit.
- Der Operator "*" bedeutet "beliebig ofte Wiederholung" ("weglassen" ist 0-malige Wiederholung).
- Der Operator "*" ist ein Postfix-Operator. Das heißt, er steht hinter seinem Argument.

Beispiele:

- ```
  _178
  Das_ist_kein_Name
  x
  _
  $Password$
  ```

... sind legale Namen.
  ```

- `5ABC`
  `!Hallo!`
  `x'`
  `-178`

... sind keine legalen Namen.

- ```
  5ABC
  !Hallo!
  x'
  -178
  ```

... sind keine legalen Namen.

Achtung:

Reservierte Wörter sind als Namen verboten !!!
  ```

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen Konstanten.
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

$$\text{number} \quad ::= \quad \text{digit digit*}$$

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen Konstanten.
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

$$\text{number} \quad ::= \quad \text{digit digit}^*$$

- Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

Beispiele:

- `17`
  `12490`
  `42`
  `0`
  `00070`

  ... sind alles legale `int`-Konstanten.

- `"Hello World!"`
  `0.5e+128`

  ... sind keine `int`-Konstanten.

Ausdrücke, die aus Zeichen (-klassen) mithilfe von

| (Alternative)

* (Iteration)

(Konkatenation) sowie

? (Option)

... aufgebaut sind, heißen reguläre Ausdrücke[a] (↑Automatentheorie).

Der Postfix-Operator "?" besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

---

[a]Gelegentlich sind auch ϵ, d.h. das "leere Wort" sowie $\emptyset$, d.h. die leere Menge zugelassen.

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*

  – alle Wörter gerader Länge (über `a,...,z,A,...,Z`);

- letter* `test` letter*

  – alle Wörter, die das Teilwort `test` enthalten;

- `_` digit* `17`

  – alle Wörter, die mit `_` anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;

- exp ::= (`e`|`E`)(`+`|`-`)? digit digit*

  float ::= digit digit* exp |

  digit* (digit . | . digit) digit* exp?

  – alle Gleitkomma-Zahlen ...

Identifizierung von

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- White Space,
- Kommentaren

... erfolgt in einer ersten Phase (↑Scanner)

$\Longrightarrow$ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter ("Tokens") aufgeteilt.

In einer zweiten Phase wird die Struktur des Programms analysiert (↑Parser).

3.4 Struktur von Programmen

Programme sind hierarchisch aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

program	::=	decl* stmt*
decl	::=	type name (, name)* ;
type	::=	`int`

3.4 Struktur von Programmen

Programme sind hierarchisch aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

program	::=	decl* stmt*
decl	::=	type name (, name)* ;
type	::=	`int`

- Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- Eine Deklaration gibt den Typ an, hier: `int`, gefolgt von einer Komma-separierten Liste von Variablen-Namen.

```
stmt  ::=  ; | { stmt* } |
           name = expr; | name = read(); | write( expr ); |
           if ( cond ) stmt |
           if ( cond ) stmt else stmt |
           while ( cond ) stmt
```

- Ein Statement ist entweder “leer” (d.h. gleich ;) oder eine geklammerte Folge von Statements;
- oder eine Zuweisung, eine Lese- oder Schreib-Operation;
- eine (einseitige oder zweiseitige) bedingte Verzweigung;
- oder eine Schleife.

$$
\begin{array}{lll}
\text{expr} & ::= & \text{number} \mid \text{name} \mid \texttt{(} \text{ expr } \texttt{)} \mid \\
 & & \text{unop expr} \mid \text{expr binop expr} \\
\text{unop} & ::= & \texttt{-} \\
\text{binop} & ::= & \texttt{-} \mid \texttt{+} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}
\end{array}
$$

- Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- oder ein unärer Operator, angewandt auf einen Ausdruck,
- oder ein binärer Operator, angewand auf zwei Argument-Ausdrücke.
- Einziger unärer Operator ist (bisher) die Negation.
- Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganz-zahlige) Division und Modulo.

```
cond    ::=  true | false | ( cond ) |
             expr comp expr |
             bunop cond | cond bbinop cond
comp    ::=  == | != | <= | < | >= | >
bunop   ::=  !
bbinop  ::=  && | ||
```

- Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein Wahrheitswert – vom Typ `boolean`).
- Bedingungen sind darum Konstanten, Vergleiche
- oder logische Verknüpfungen anderer Bedingungen.

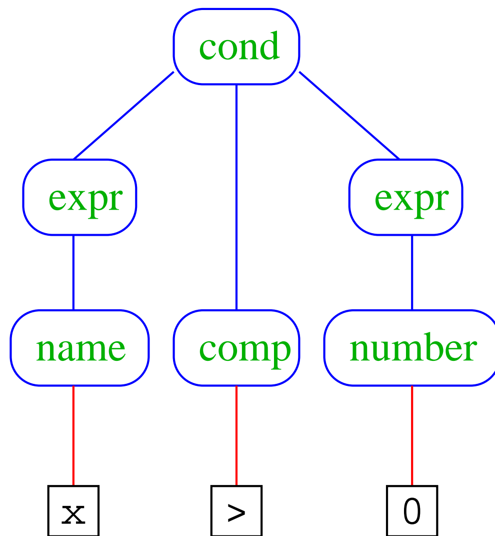
Puh!!! Geschafft ...

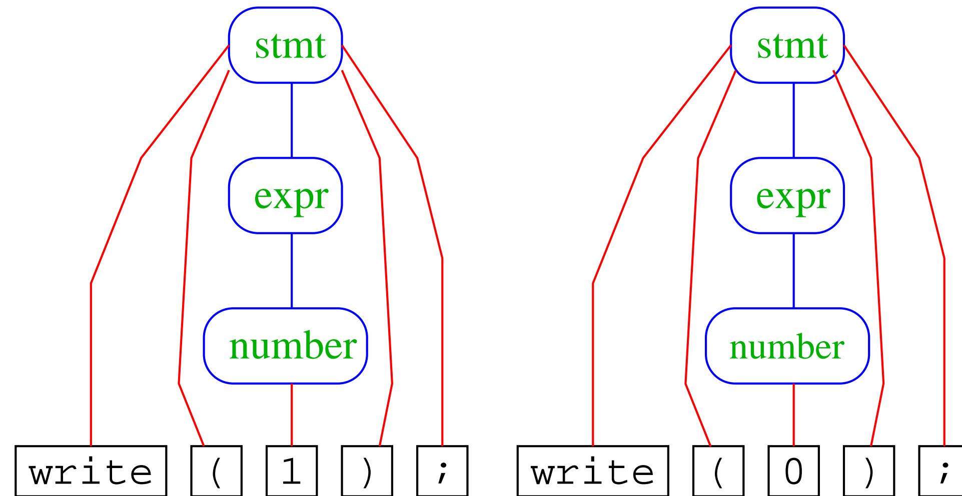
Beispiel:

```
int x;
x = read();
if (x > 0)
    write(1);
else
    write(0);
```

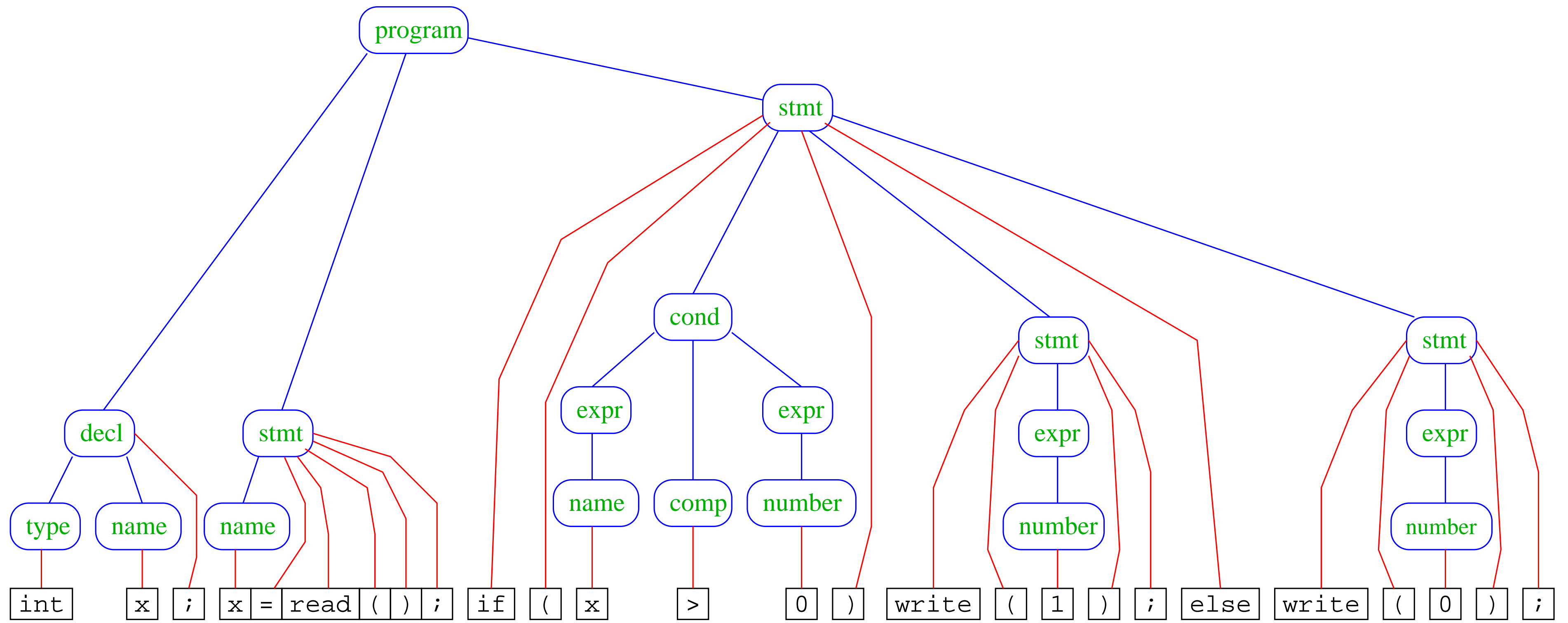
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

Syntax-Bäume für `x > 0` sowie `write(0);` und `write(1);`





Blätter:	Wörter/Tokens
innere Knoten:	Namen von Programm-Bestandteilen


program
stmt
decl
stmt
cond
stmt
stmt
expr
expr
expr
expr
type
name
name
name
comp
number
number
number
int
x
;
x
=
read
(
)
;
if
(
x
>
0
)
write
(
1
)
;
else
write
(
0
)
;

Bemerkungen:

- Die vorgestellte Methode der Beschreibung von Syntax heißt EBNF-Notation (Extended Backus Naur Form Notation).
- Ein anderer Name dafür ist erweiterte kontextfreie Grammatik (↑Linguistik, Automatentheorie).
- Linke Seiten von Regeln heißen auch Nicht-Terminale.
- Tokens heißen auch Terminale.

Noam Chomsky,
MIT

John Backus, IBM
Turing Award
(Erfinder von Fortran)

Peter Naur,
Turing Award
(Erfinder von Algol60)

Achtung:

- Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nicht-Terminale enthalten.
- Deshalb sind kontextfreie Grammatiken mächtiger als reguläre Ausdrücke.

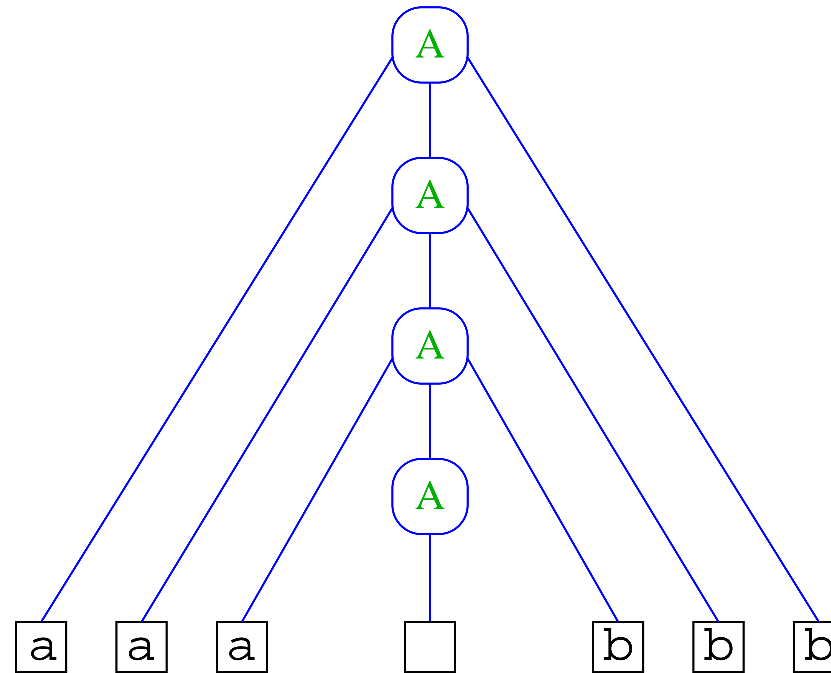
Beispiel:

$$\mathcal{L} = \{\epsilon, \texttt{ab}, \texttt{aabb}, \texttt{aaabbb}, \ldots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$A \quad ::= \quad (\texttt{ a } A \texttt{ b })?$$

Syntax-Baum für das Wort `aaabbb` :

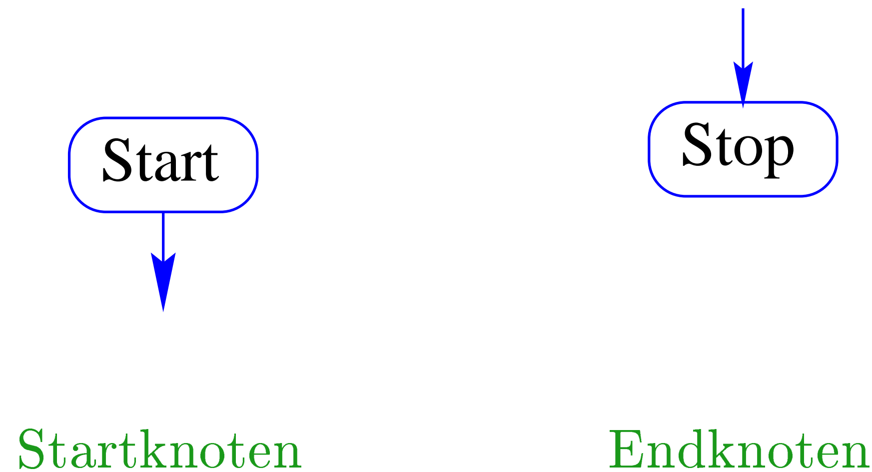


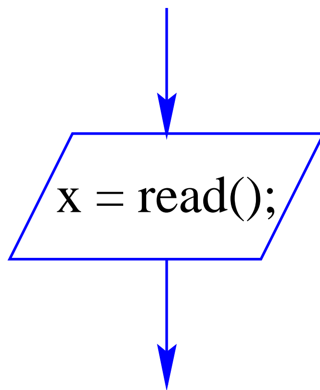
Für $\mathcal{L}$ gibt es aber keinen regulären Ausdruck!!! (↑Automatentheorie)

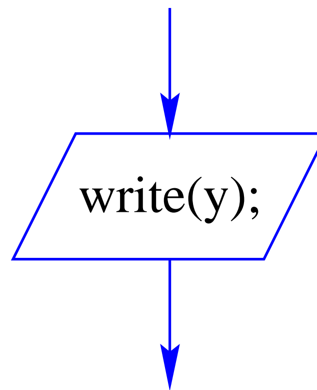
4 Kontrollfluss-Diagramme

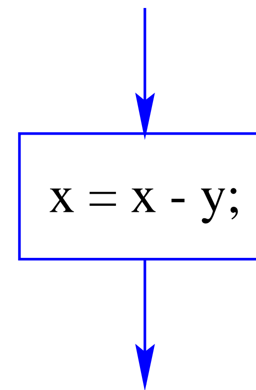
In welcher Weise die Operationen eines Programms nacheinander ausgeführt werden, lässt sich anschaulich mithilfe von Kontrollfluss-Diagrammen darstellen.

Ingredienzien:

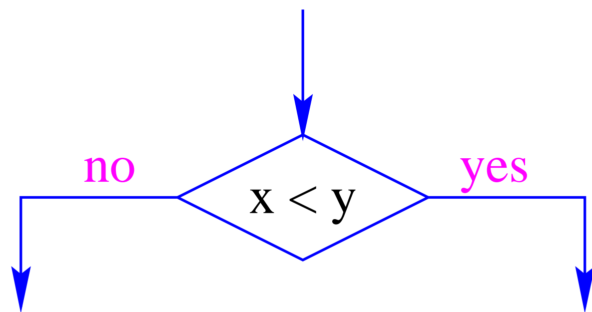






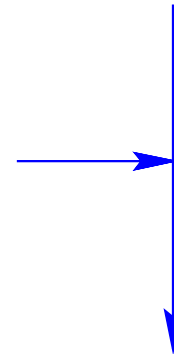


Eingabe Ausgabe Zuweisung



bedingte Verzweigung

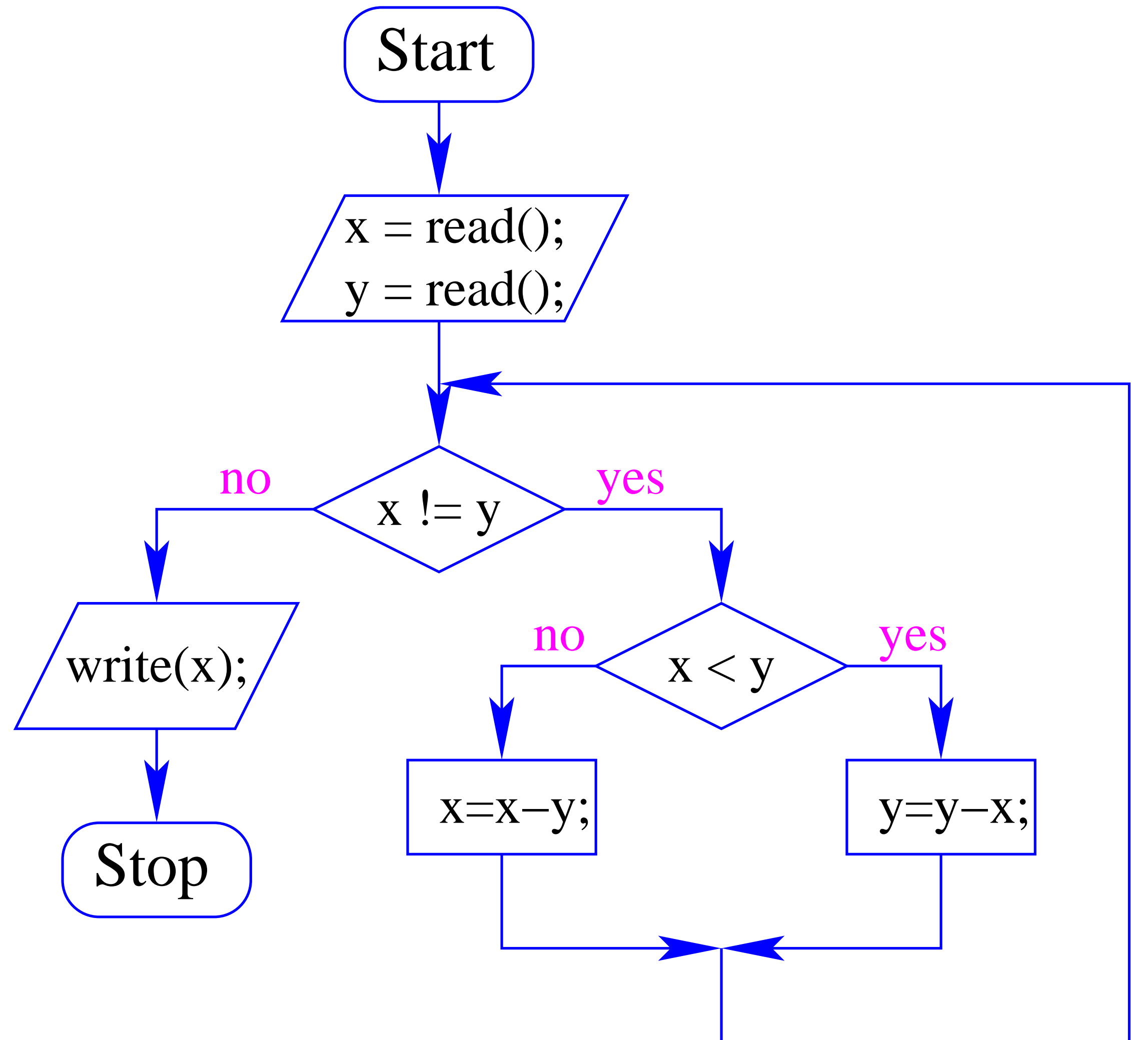
Kante



Zusammenlauf

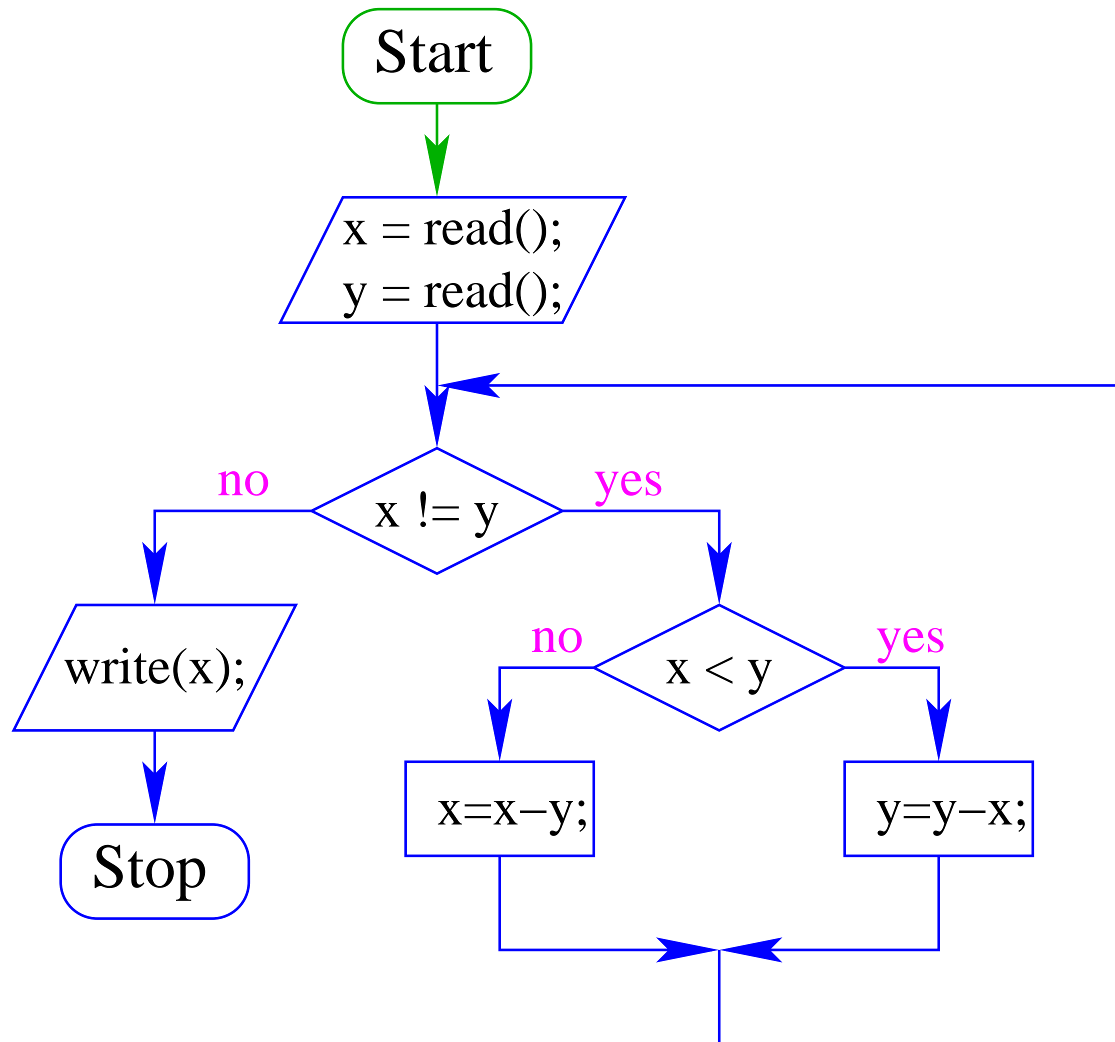
Beispiel:

```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

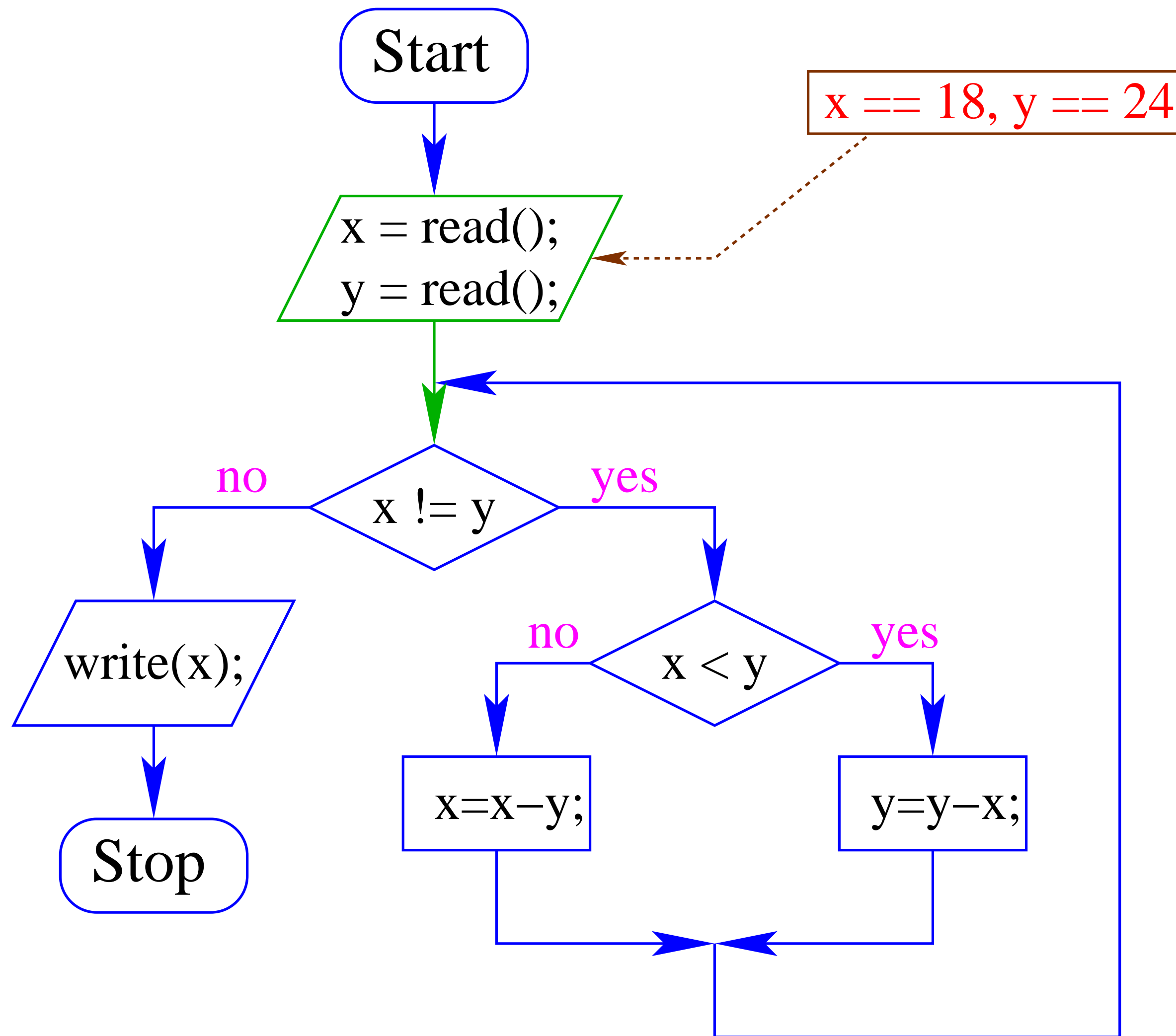


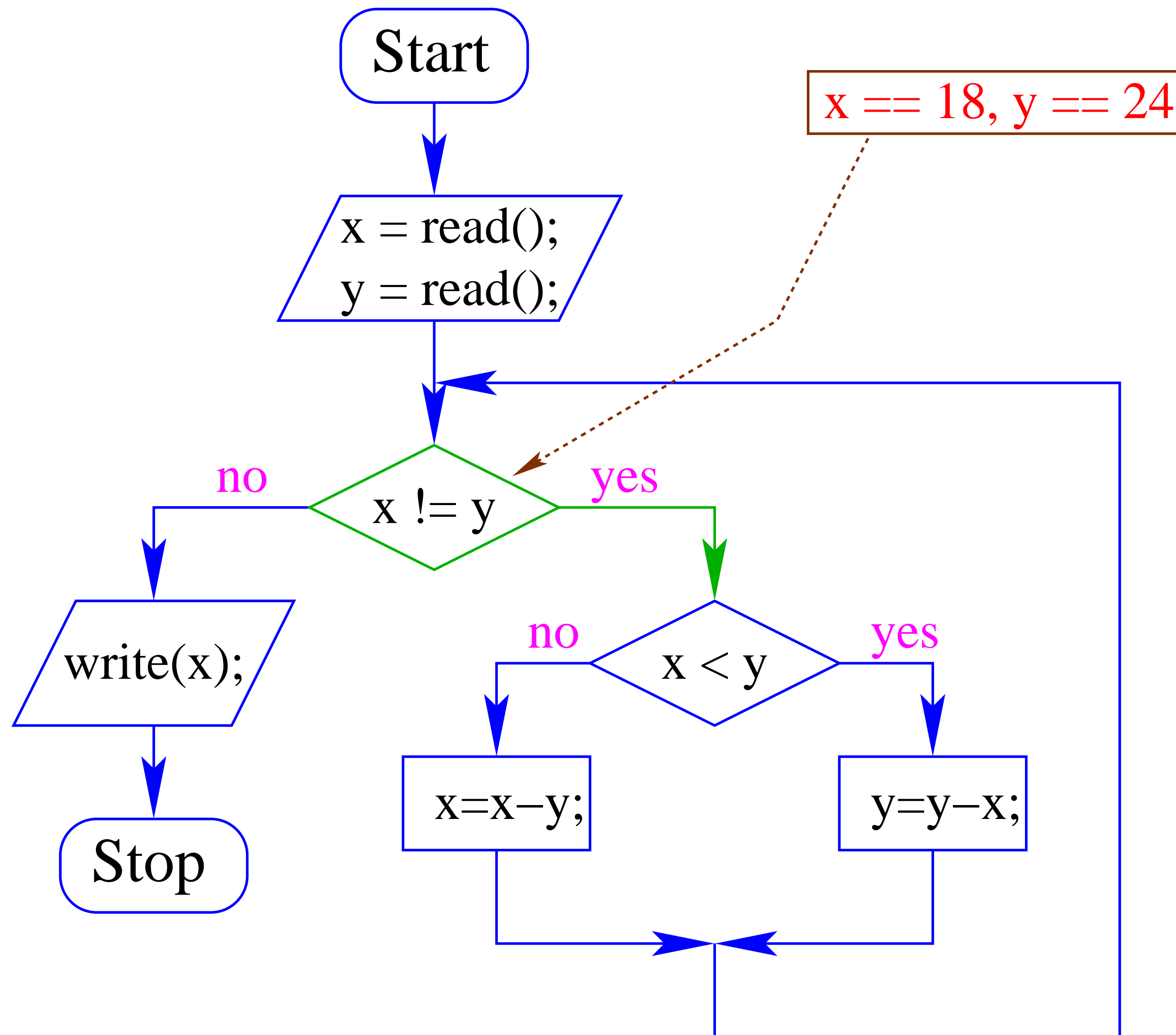
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollfluss-Diagramm vom Startknoten zum Endknoten.
- Die Deklarationen von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer dem Start- und Endknoten) sind die dabei auszuführenden Operationen bzw. auszuwertenden Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung für die aktuellen Werte der Variablen ausgewertet werden.

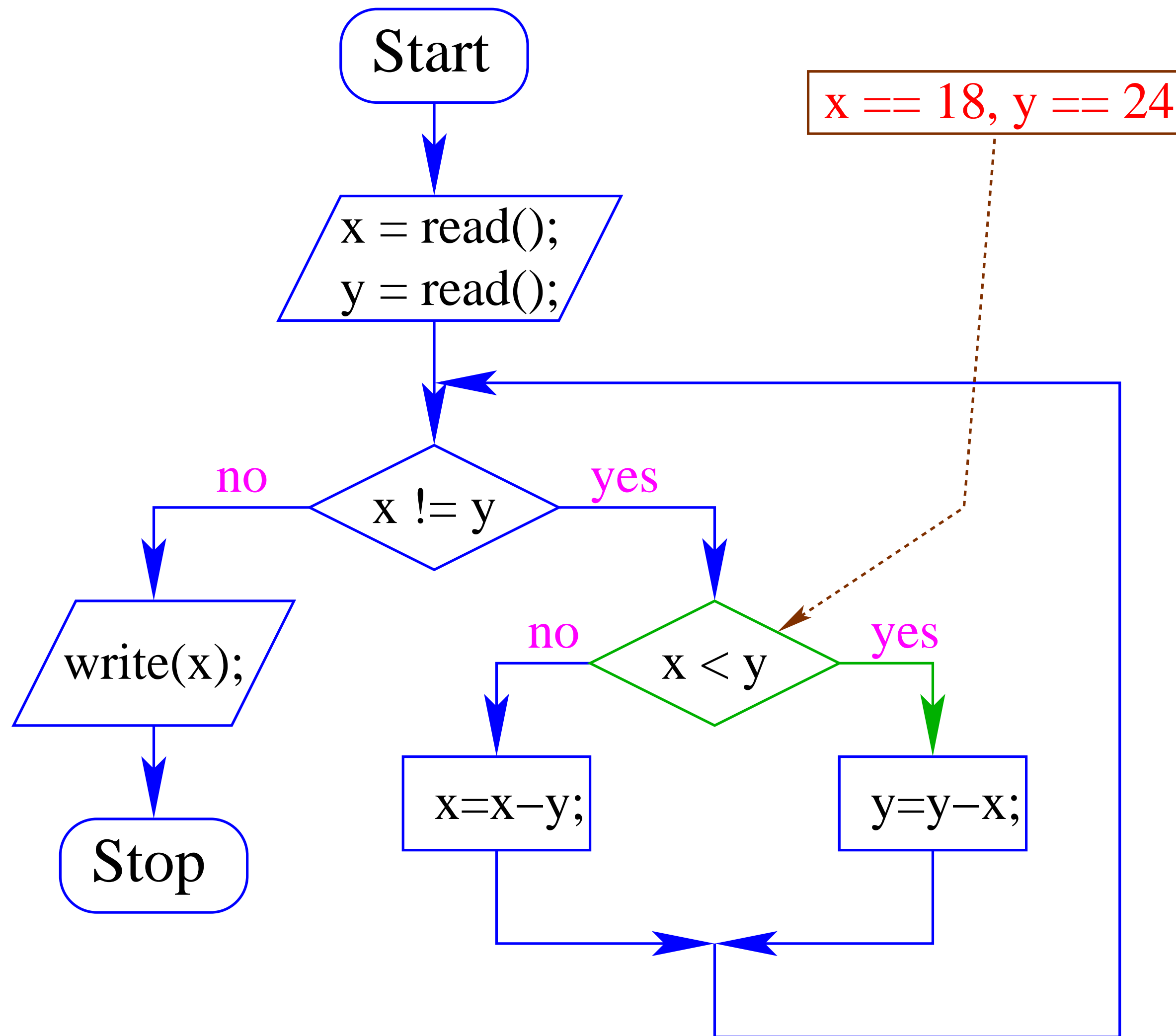
$\Longrightarrow$ operationelle Semantik

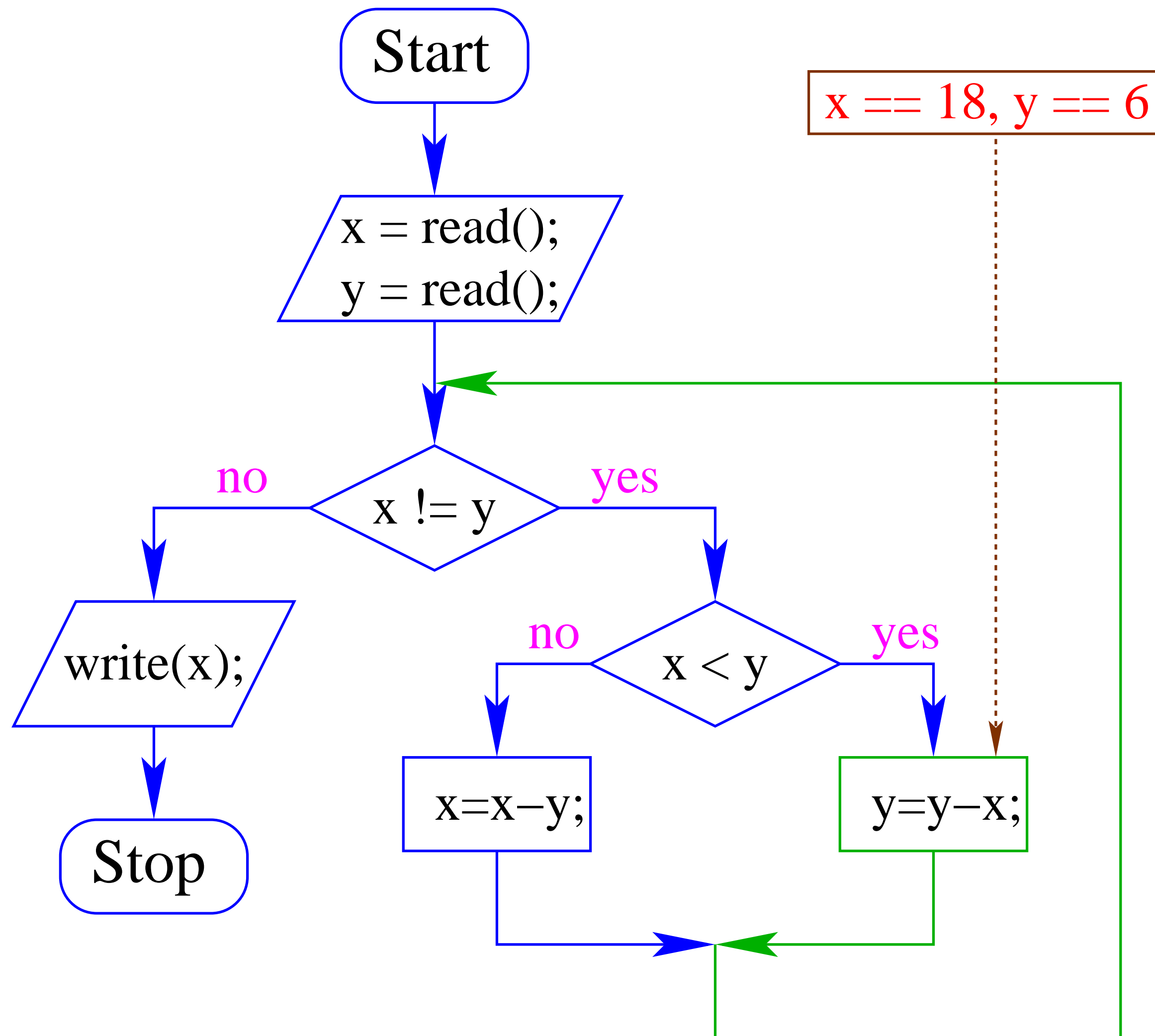


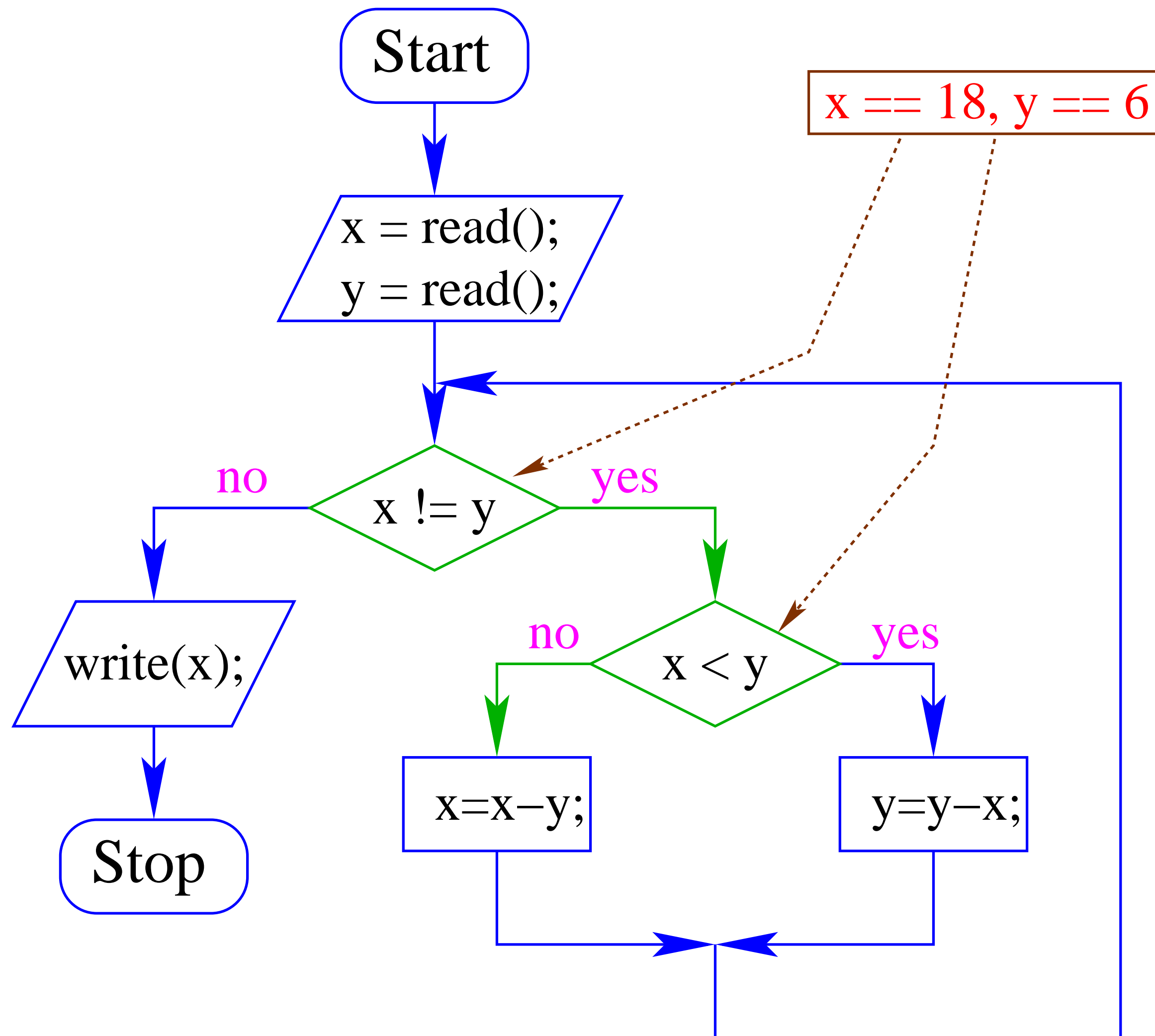
Start
x = read();
y = read();
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;

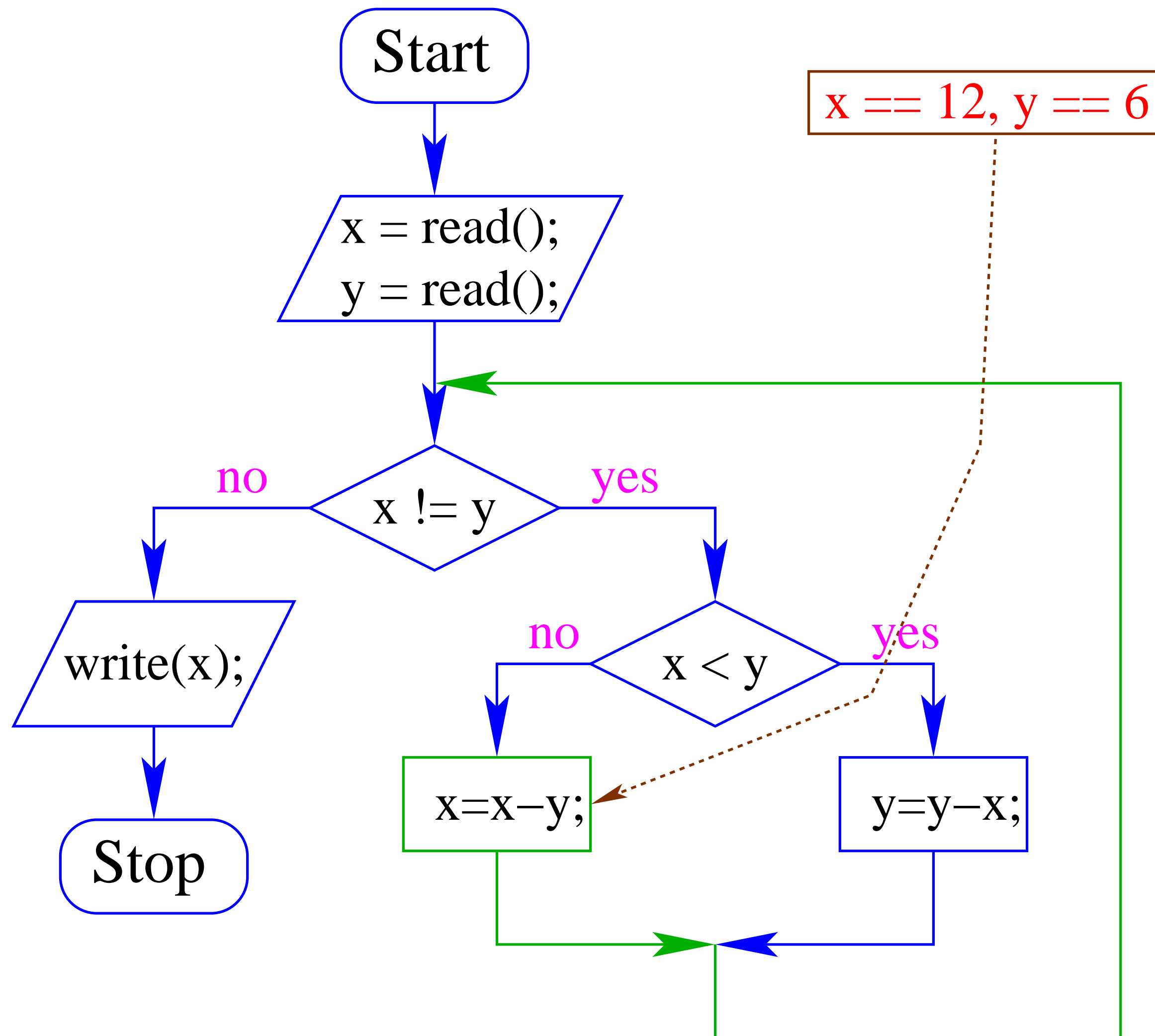

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

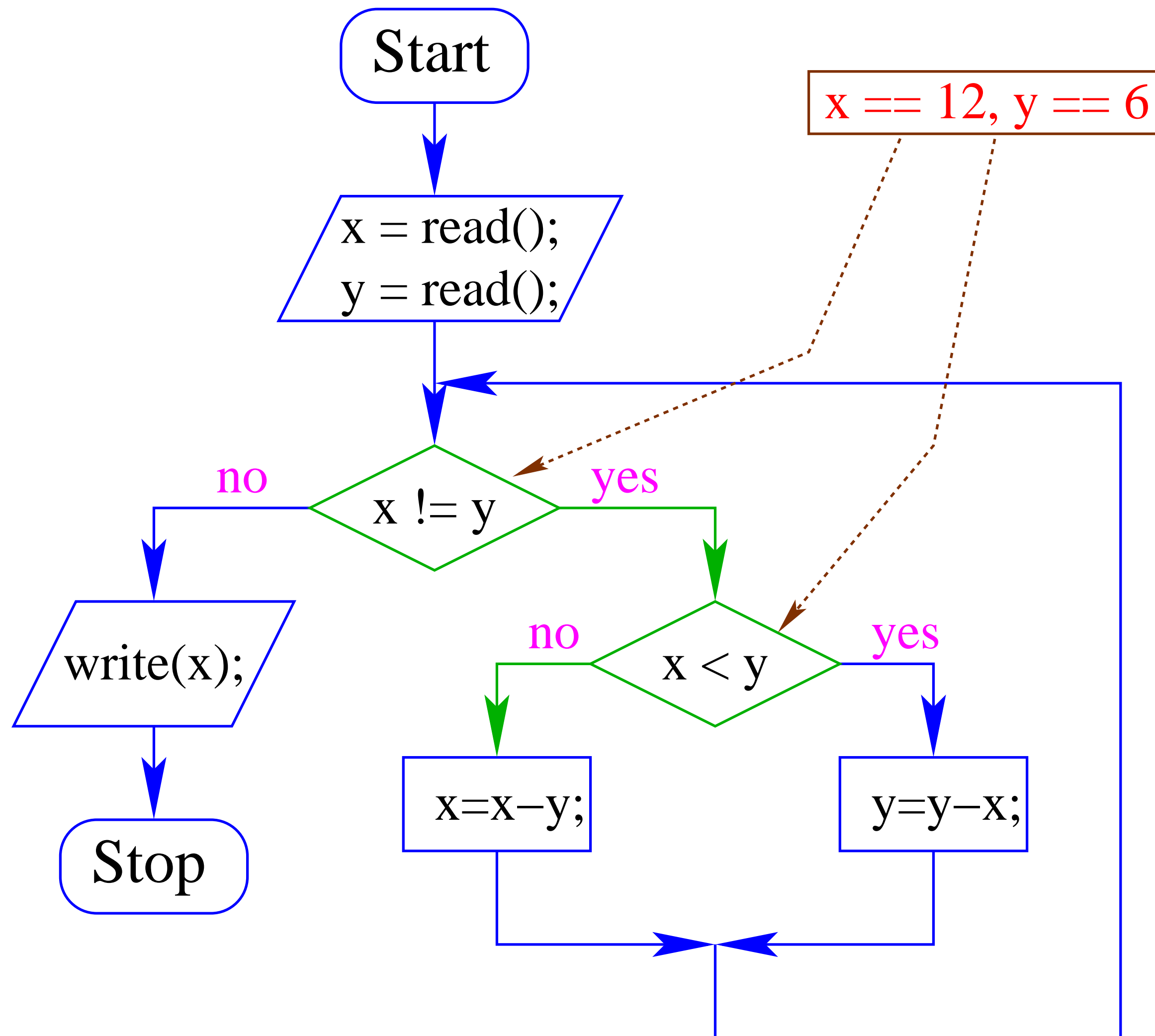

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
x=x−y;
y=y−x;
Stop

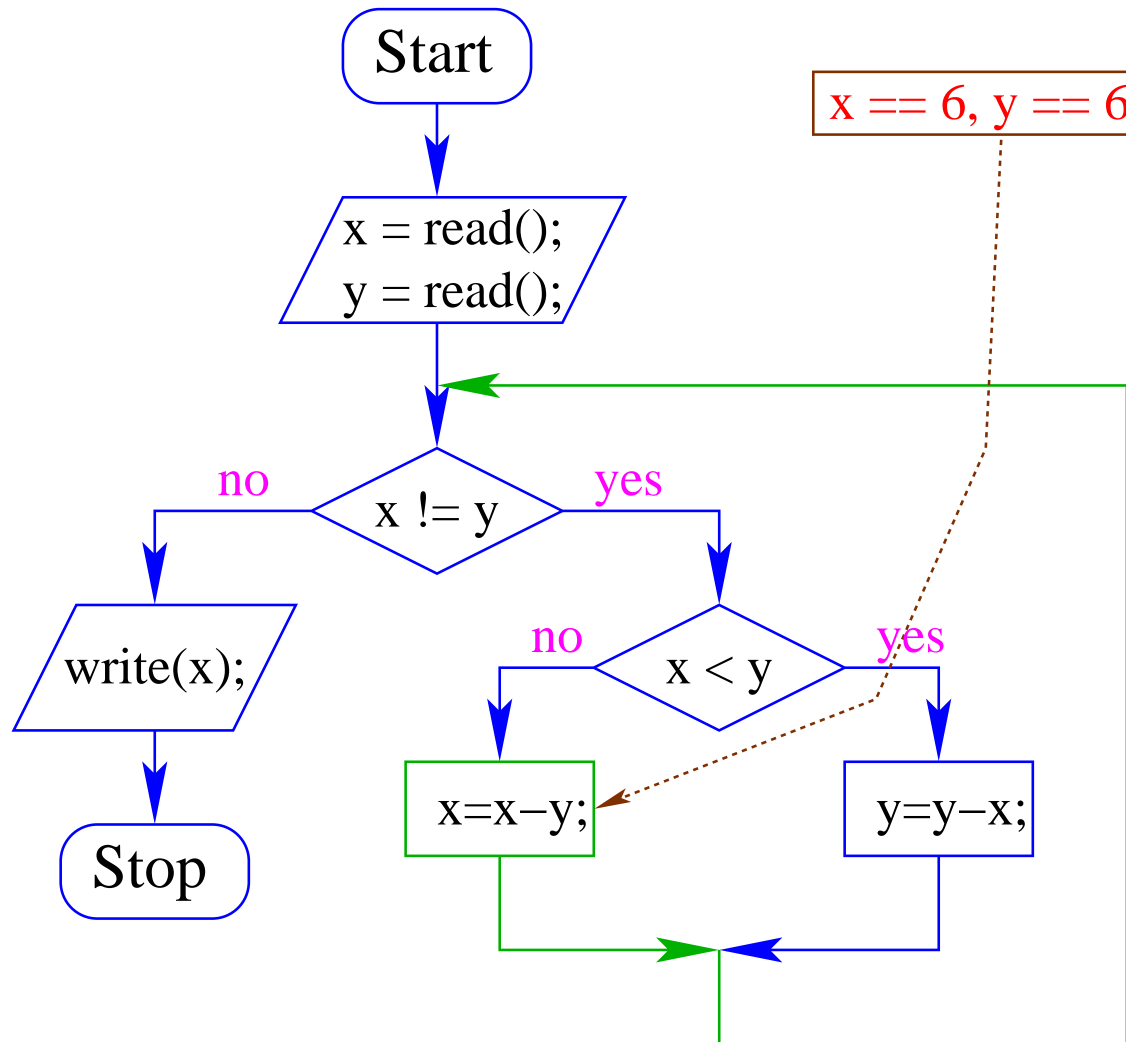

Start
x = read();
y = read();
x == 18, y == 24
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

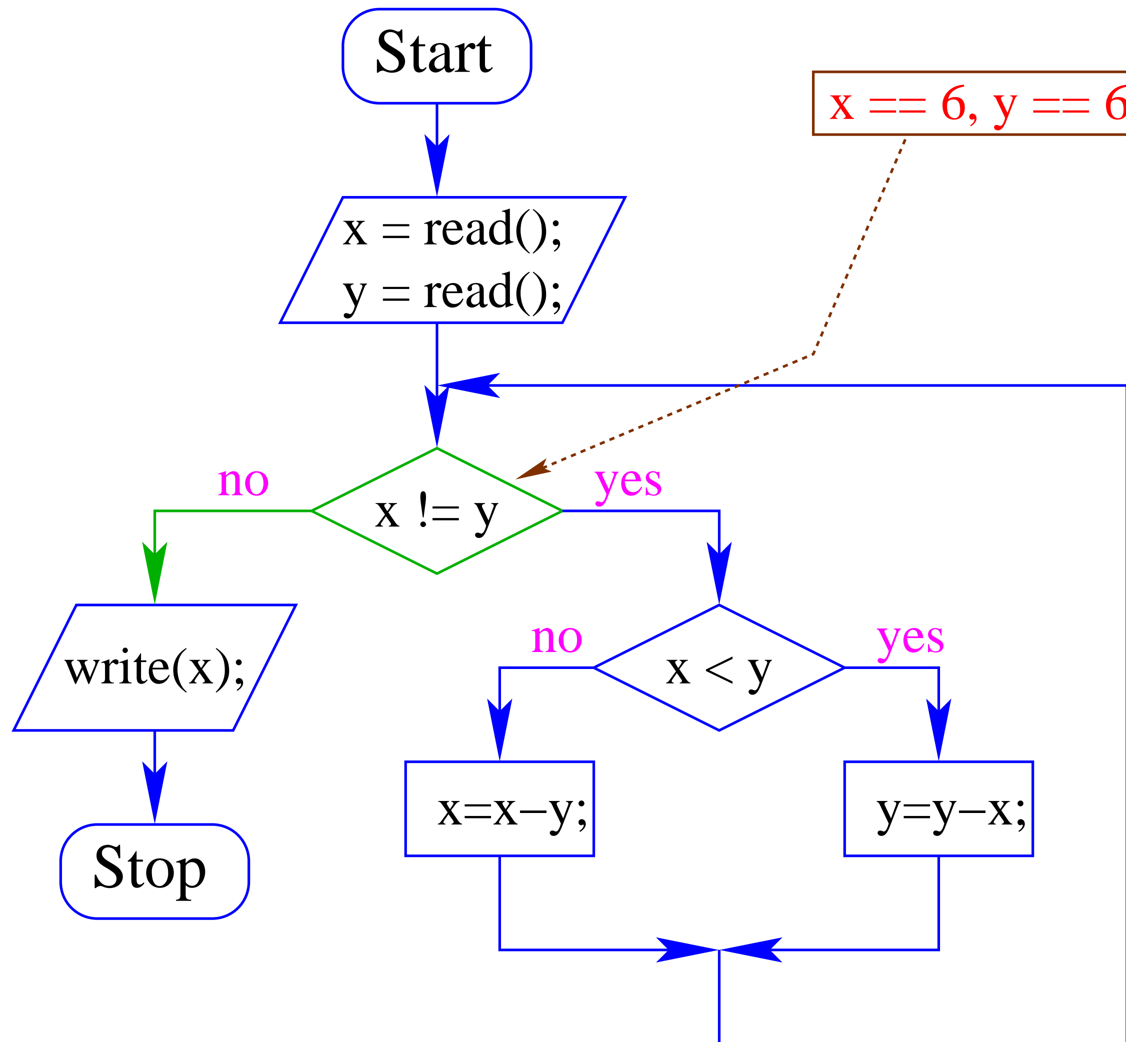

Start
x = read();
y = read();
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;
x == 18, y == 6

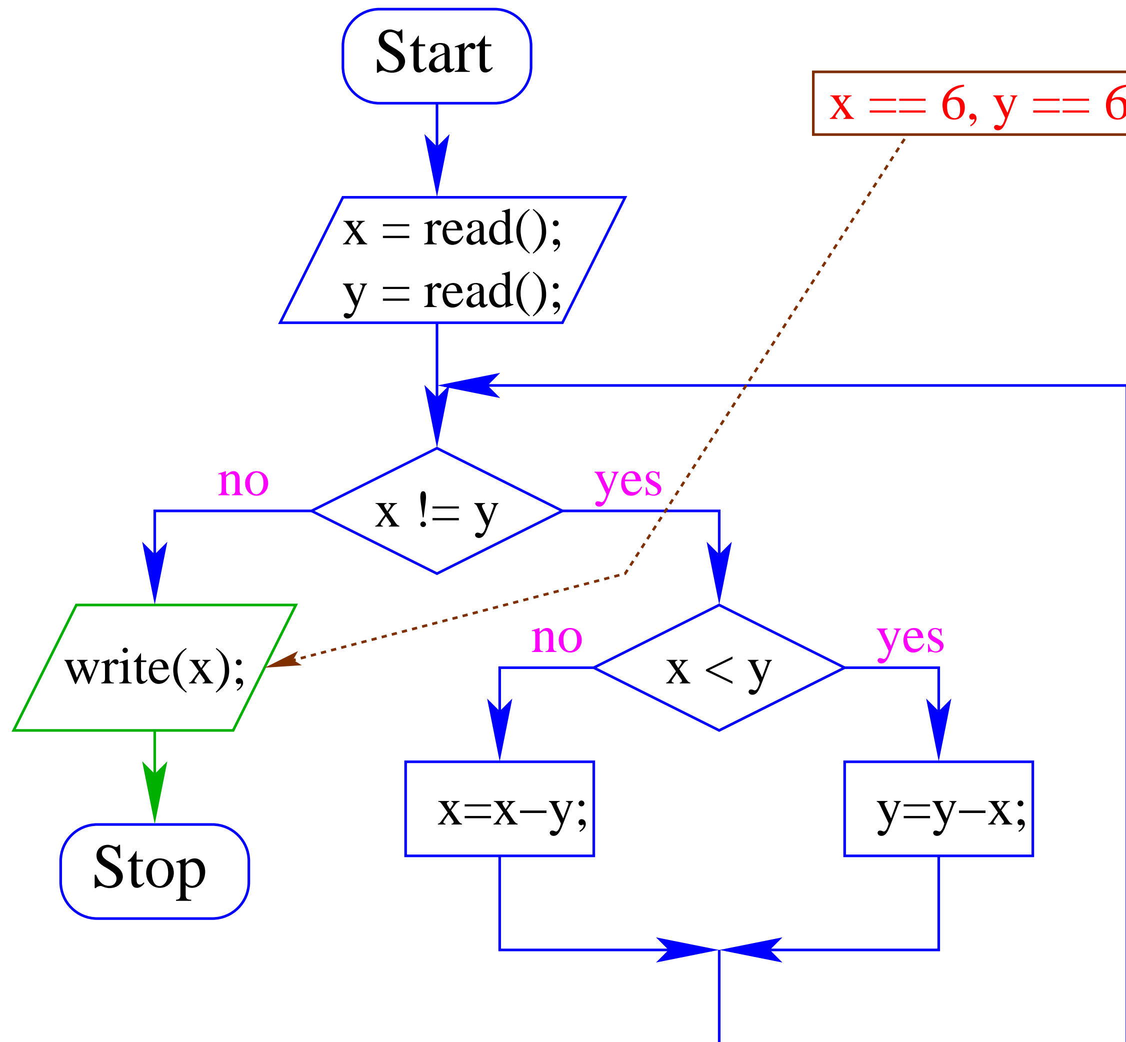

Start
x = read();
y = read();
x == 18, y == 6
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

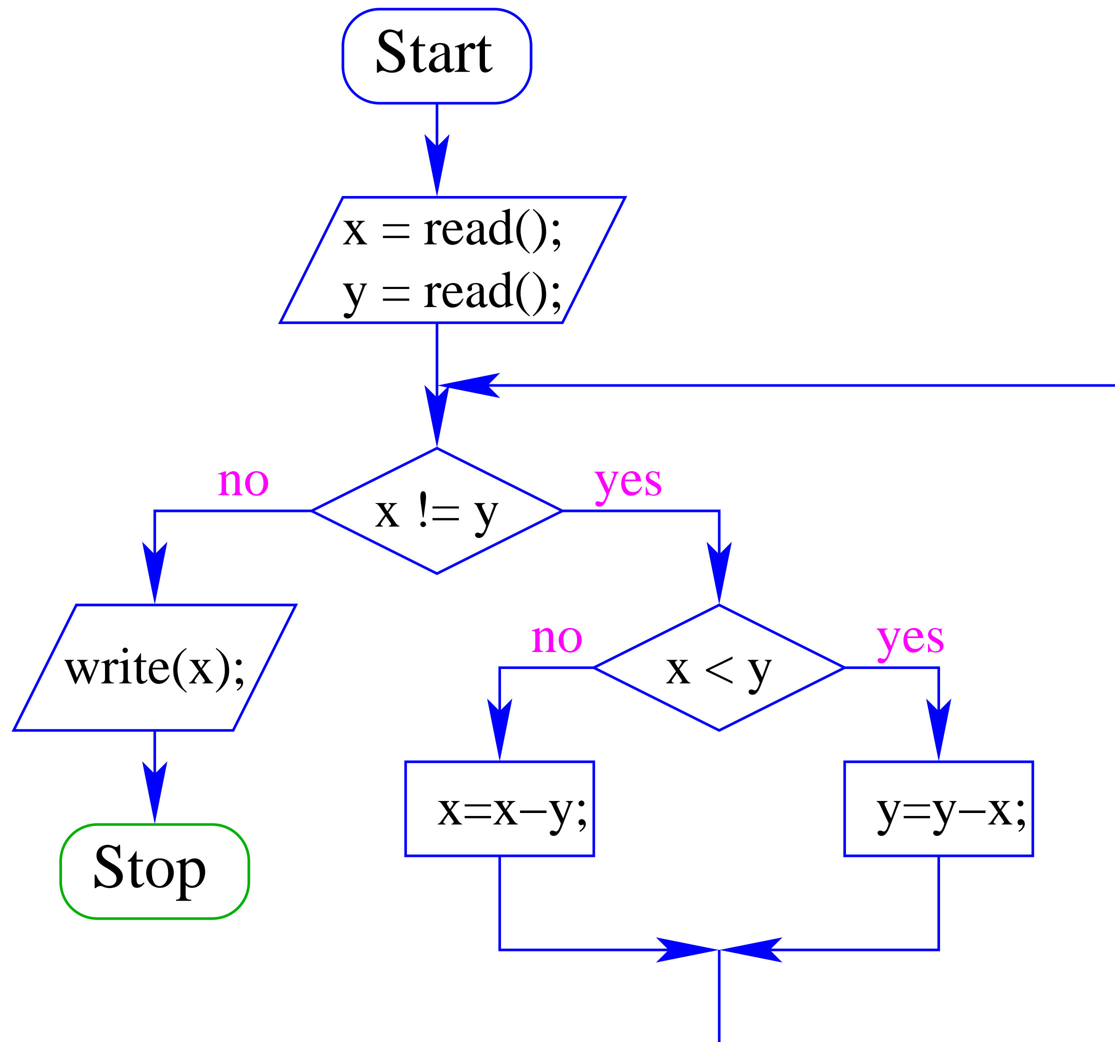

Start
x == 12, y == 6
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;


Start
x == 12, y == 6
x = read();
y = read();
no
yes
x != y
write(x);
no
yes
x < y
x=x−y;
y=y−x;
Stop


Start
x = read();
y = read();
x == 6, y == 6
no
x != y
yes
write(x);
no
x < y
yes
x=x−y;
y=y−x;
Stop


Start
x == 6, y == 6
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

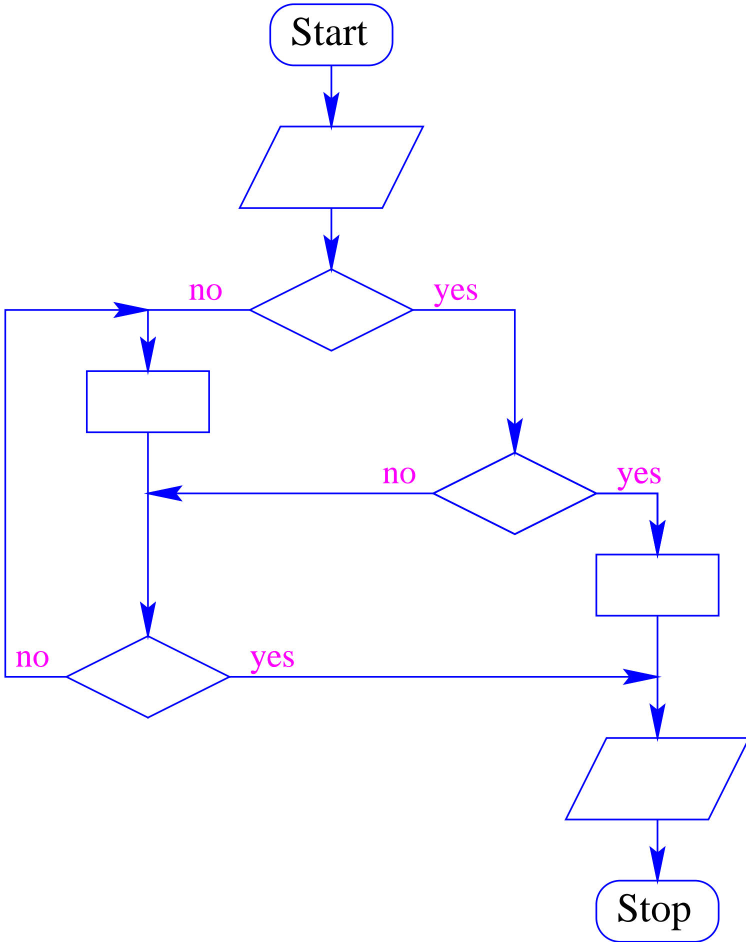

Start
x == 6, y == 6
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
x=x−y;
y=y−x;
Stop


Start
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

Achtung:

- Zu jedem MiniJava-Programm lässt sich ein Kontrollfluss-Diagramm konstruieren;
- die umgekehrte Richtung gilt zwar ebenfalls, liegt aber nicht so auf der Hand.

Beispiel:



5 Mehr Java

Um komfortabel programmieren zu können, brauchen wir

- mehr Datenstrukturen;
- mehr Kontrollstrukturen.

5.1 Mehr Basistypen

- Außer `int`, stellt Java weitere Basistypen zur Verfügung.
- Zu jedem Basistyp gibt es eine Menge möglicher Werte.
- Jeder Wert eines Basistyps benötigt die gleiche Menge Platz, um ihn im Rechner zu repräsentieren.
- Der Platz wird in Bit gemessen.

  (Wie viele Werte kann man mit n Bit darstellen?)

Es gibt vier Sorten ganzer Zahlen:

Typ	Platz	kleinster Wert	größter Wert
`byte`	8	−128	127
`short`	16	−32 768	32 767
`int`	32	−2 147 483 648	2 147 483 647
`long`	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie `byte` oder `short` spart Platz.

Es gibt vier Sorten ganzer Zahlen:

Typ	Platz	kleinster Wert	größter Wert
`byte`	8	−128	127
`short`	16	−32 768	32 767
`int`	32	−2 147 483 648	2 147 483 647
`long`	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie `byte` oder `short` spart Platz.

Achtung: Java warnt nicht vor Überlauf/Unterlauf !!

Beispiel:

```
int x = 2147483647; // grösstes int
x = x+1;
write(x);
```

... liefert `-2147483648` ...

- In realem Java kann man bei der Deklaration einer Variablen ihr direkt einen ersten Wert zuweisen (Initialisierung).
- Man kann sie sogar (statt am Anfang des Programms) erst an der Stelle deklarieren, an der man sie das erste Mal braucht!

Es gibt zwei Sorten von Gleitkomma-Zahlen:

Typ	Platz	kleinster Wert	größter Wert	
`float`	32	ca. -3.4e+38	ca. 3.4e+38	7 signifikante Stellen
`double`	64	ca. -1.7e+308	ca. 1.7e+308	15 signifikante Stellen

- Überlauf/Unterlauf liefert die Werte `Infinity` bzw. `-Infinity`.
- Für die Auswahl des geeigneten Typs sollte die gewünschte Genauigkeit des Ergebnisses berücksichtigt werden.
- Gleitkomma-Konstanten im Programm werden als `double` aufgefasst.
- Zur Unterscheidung kann man an die Zahl `f` (oder `F`) bzw. `d` (oder `D`) anhängen.

... weitere Basistypen:

Typ	Platz	Werte
`boolean`	1	`true`, `false`
`char`	16	alle Unicode-Zeichen

Unicode ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

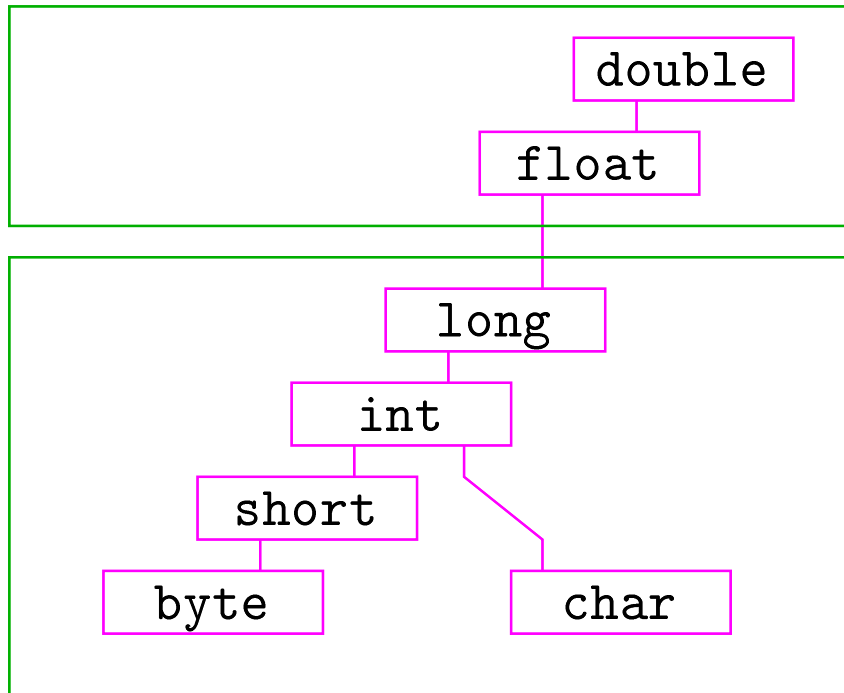
- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

`char`-Konstanten schreibt man mit Hochkommas: `'A'`, `';'`, `'\n'`.

5.2 Mehr über Arithmetik

- Die Operatoren `+`, `-`, `*`, `/` und `%` gibt es für jeden der aufgelisteten Zahltypen.

- Werden sie auf ein Paar von Argumenten verschiedenen Typs angewendet, wird automatisch vorher der speziellere in den allgemeineren umgewandelt (impliziter Type Cast) ...



Gleitkomma-Zahlen

ganze Zahlen

Beispiel:

```
short xs = 1;
int x = 999999999;
write(x + xs);
```

... liefert den `int`-Wert `1000000000` ...

```
float xs = 1.0f;
int x = 999999999;
write(x + xs);
```

... liefert den `float`-Wert `1.0E9` ...

Beispiel:

```
short xs = 1;
int x = 999999999;
write(x + xs);
```

... liefert den `int`-Wert 1000000000 ...

```
float xs = 1.0f;
int x = 999999999;
write(x + xs);
```

... liefert den `float`-Wert 1.0E9 ...

... vorausgesetzt, `write()` kann Gleitkomma-Zahlen ausgeben.

Achtung:

- Das Ergebnis einer Operation auf `float` kann aus dem Bereich von `float` herausführen. Dann ergibt sich der Wert `Infinity` oder `-Infinity`.

  Das gleiche gilt für `double`.

- Das Ergebnis einer Operation auf Basistypen, die in `int` enthalten sind (außer `char`), liefern ein `int`.
- Wird das Ergebnis einer Variablen zugewiesen, sollte deren Typ dies zulassen.

5.3 Strings

Der Datentyp `String` für Wörter ist kein Basistyp, sondern eine Klasse (dazu kommen wir später)

Hier behandeln wir nur drei Eigenschaften:

- Werte vom Typ `String` haben die Form `"Hello World!";`
- Man kann Wörter in Variablen vom Typ `String` abspeichern.
- Man kann Wörter mithilfe des Operators “`+`” konkatenieren.

Beispiel:

```
String s0 = "";
String s1 = "Hel";
String s2 = "lo Wo";
String s3 = "rld!";
write(s0 + s1 + s2 + s3);
```

... schreibt `Hello World!` auf die Ausgabe.

Beachte:

- Jeder Wert in Java hat eine Darstellung als `String`.
- Wird der Operator "+" auf einen Wert vom Typ `String` und einen anderen Wert x angewendet, wird x automatisch in seine `String`-Darstellung konvertiert ...

⟹ ... liefert einfache Methode, um `float` oder `double` auszugeben !!!

Beispiel:

```
double x = -0.55e13;
write("Eine Gleitkomma-Zahl: "+x);
```

... schreibt `Eine Gleitkomma-Zahl: -0.55E13` auf die Ausgabe.

5.4 Felder

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!

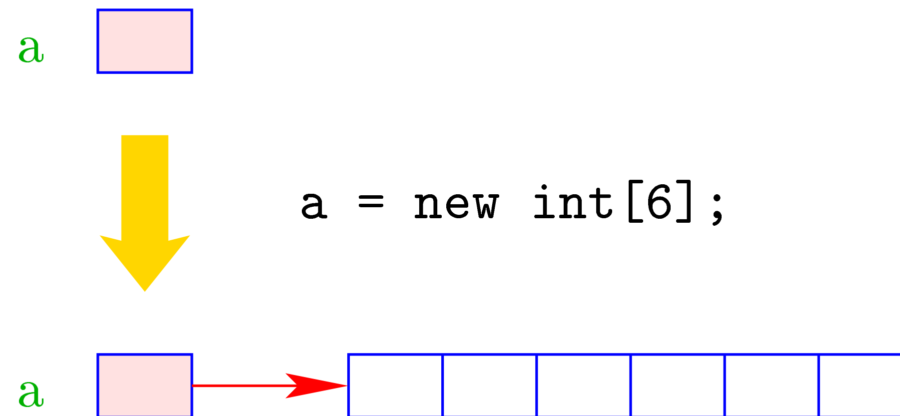
Feld:	17	3	-2	9	0	1
Index:	0	1	2	3	4	5

Beispiel: Einlesen eines Felds

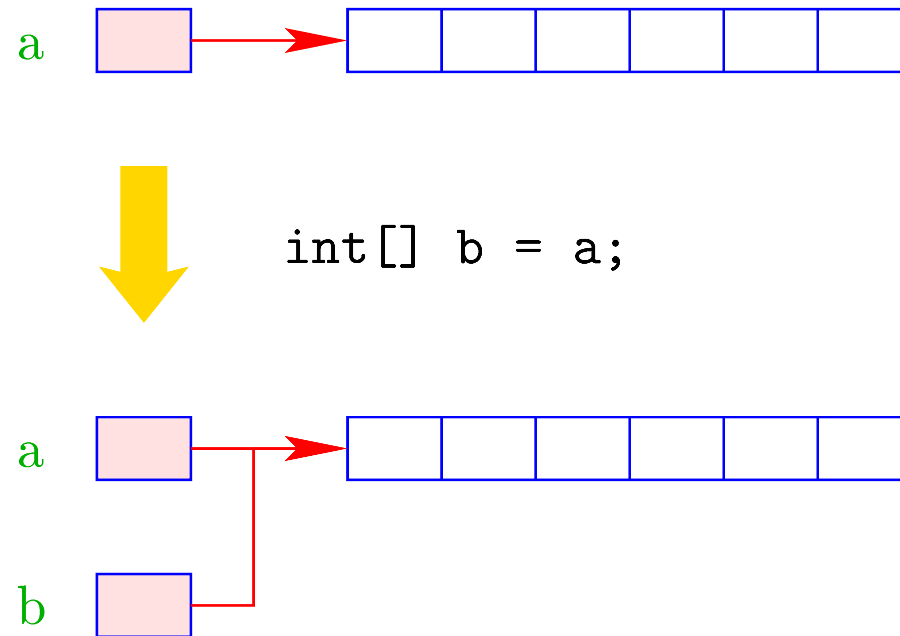
```
int[] a; // Deklaration
int n = read();

a = new int[n];
          // Anlegen des Felds
int i = 0;
while (i < n) {
    a[i] = read();
    i = i+1;
    }
```

- type [] name ; deklariert eine Variable für ein Feld (array), dessen Elemente vom Typ type sind.
- Alternative Schreibweise:
  type name [];
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen Verweis darauf zurück:



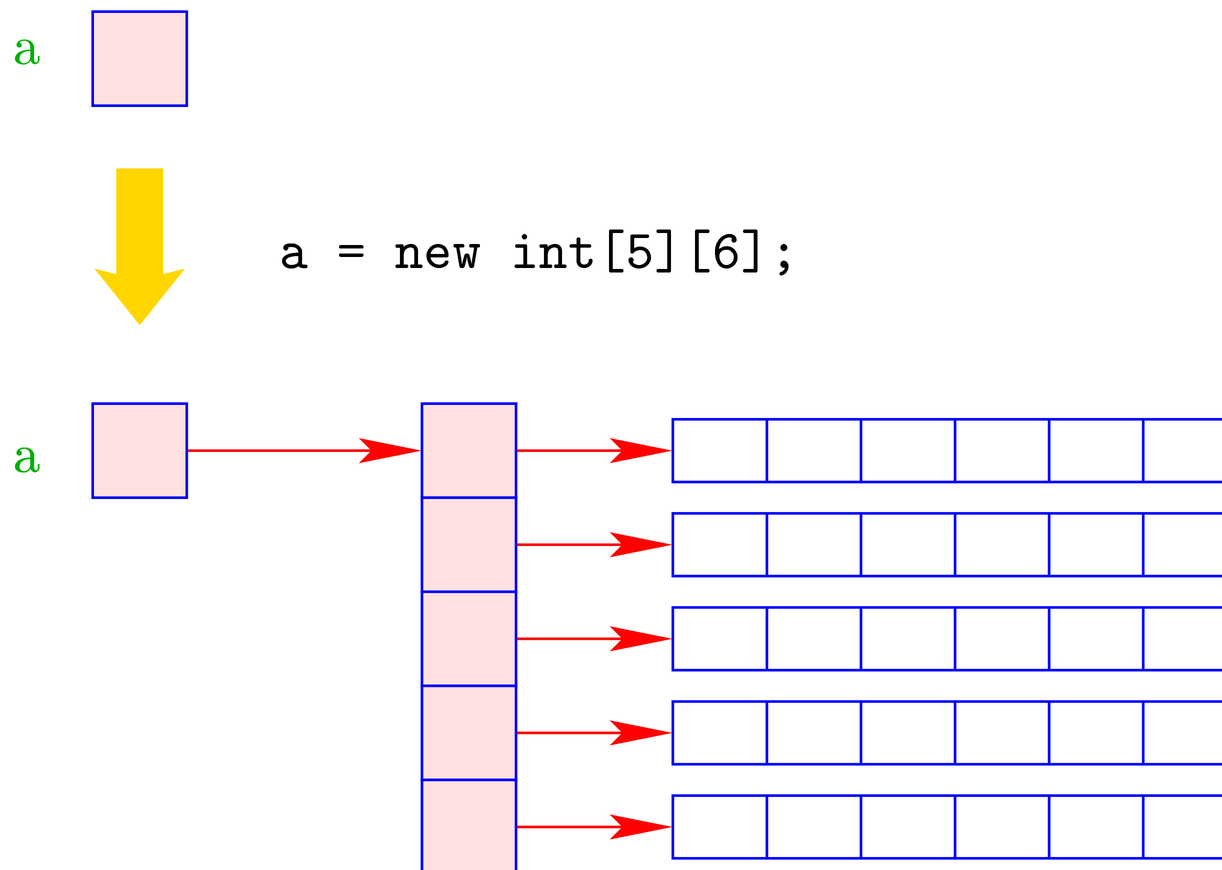
- Der Wert einer Feld-Variable ist also ein Verweis.
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



- Die Elemente eines Felds sind von 0 an durchnumeriert.
- Die Anzahl der Elemente des Felds name ist name`.length`.
- Auf das i-te Element des Felds name greift man mittels name`[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0, \ldots,$ name`.length`-1$\}$ liegt.
- Liegt der Index außerhalb des Intervalls, wird die `ArrayIndexOutofBoundsException` ausgelöst (↑Exceptions).

Mehrdimensionale Felder

- Java unterstützt direkt nur ein-dimensionale Felder.
- Ein zwei-dimensionales Feld ist ein Feld von Feldern ...



5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel (Forts.): Bestimmung des Minimums

```
int result = a[0];
int i = 1;        // Initialisierung
while (i < a.length) {
    if (a[i] < result)
        result = a[i];
    i = i+1;   // Modifizierung
    }
write(result);
```

Mithilfe des `for`-Statements:

```
int result = a[0];
for (int i = 1; i < a.length; ++i)
    if (a[i] < result)
        result = a[i];
write(result);
```

Allgemein:

for (init; cond; modify) stmt

... entspricht:

{ init ; while (cond) { stmt modify ;} }

... wobei `++i` äquivalent ist zu `i = i+1` .

Warnung:

- Die Zuweisung `x = x-1` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variable `x` erfolgt als Seiteneffekt.
- Der Semikolon “;” hinter einem Ausdruck wirft nur den Wert weg.

$\Longrightarrow$ ... fatal für Fehler in Bedingungen ...

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

- Die Operatoranwendungen `++x` und `x++` inkrementieren beide den Wert der Variablen `x`.
- `++x` tut das, bevor der Wert des Ausdrucks ermittelt wird (Pre-Increment).
- `x++` tut das, nachdem der Wert ermittelt wurde (Post-Increment).
- `a[x++] = 7;` entspricht:

```
a[x] = 7;
x = x+1;
```

- `a[++x] = 7;` entspricht:

```
x = x+1;
a[x] = 7;
```

5.6 Funktionen und Prozeduren

Oft möchte man

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel: Einlesen eines Felds

```
public static int[] readArray(int number) {
        // number = Anzahl der zu lesenden Elemente
    int[] result = new int[number]; // Anlegen des Felds
    for (int i = 0; i < number; ++i) {
        result[i] = read();
        }
    return result;
}
```

- Die erste Zeile ist der Header der Funktion.
- `public` sagt, wo die Funktion verwendet werden darf (↑kommt später)
- `static` kommt ebenfalls später.
- `int[]` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der formalen Parameter, hier: `(int number)`.
- Der Rumpf der Funktion steht in geschwungenen Klammern.
- `return` expr beendet die Ausführung der Funktion und liefert den Wert von expr zurück.

- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von “`{`” und “`}`”, sind nur innerhalb dieses Blocks sichtbar, d.h. benutzbar (lokale Variablen).
- Der Rumpf einer Funktion ist ein Block.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert `7`.

Weiteres Beispiel: Bestimmung des Minimums

```
public static int min (int[] b) {
    int result = b[0];
    for (int i = 1; i < b.length; ++i) {
        if (b[i] < result)
            result = b[i];
        }
    return result;
    }
```

... daraus basteln wir das Java-Programm `Min` :

```
public class Min extends MiniJava {
   public static int[] readArray (int number) { ... }
   public static int min (int[] b) { ... }
      // Jetzt kommt das Hauptprogramm
   public static void main (String[] args) {
      int n = read();
      int[] a = readArray(n);
      int result =  min(a);
      write(result);
      }    // end of main()
}          // end of class Min
```

- Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen Prozeduren.
- Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

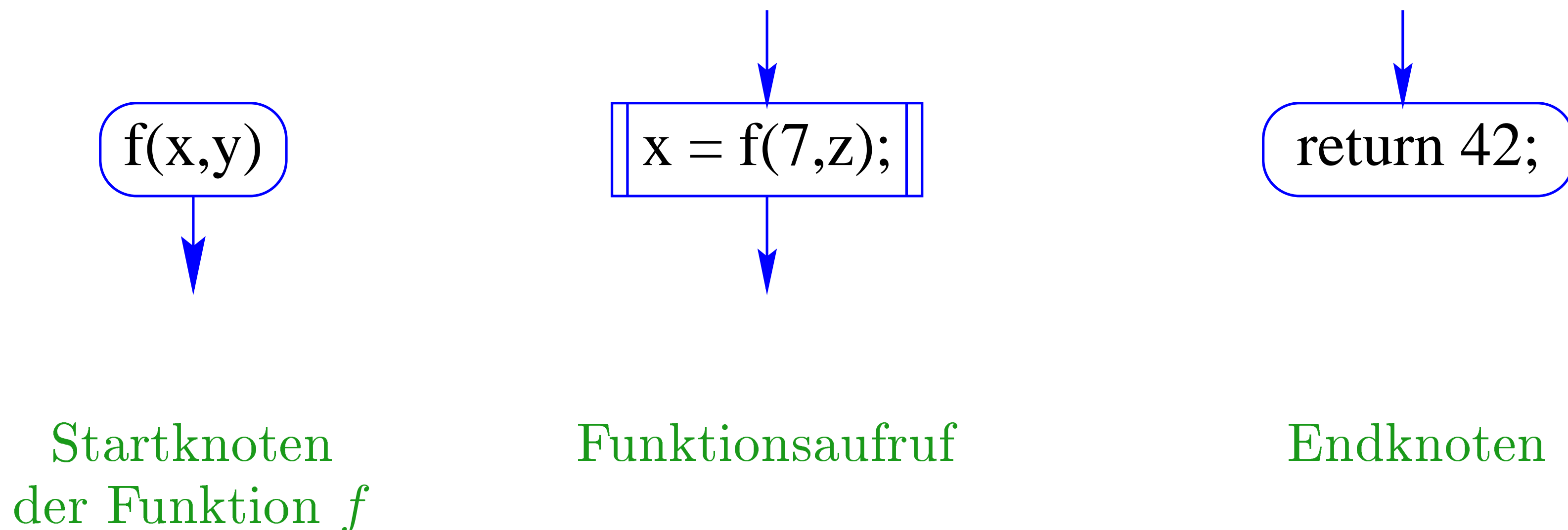
```
public class Test extends MiniJava {
   public static void main (String [] args) {
      write(args[0]+args[1]);
      }
}  // end of class Test
```

Dann liefert der Aufruf:

```
java Test "Hel" "lo World!"
```

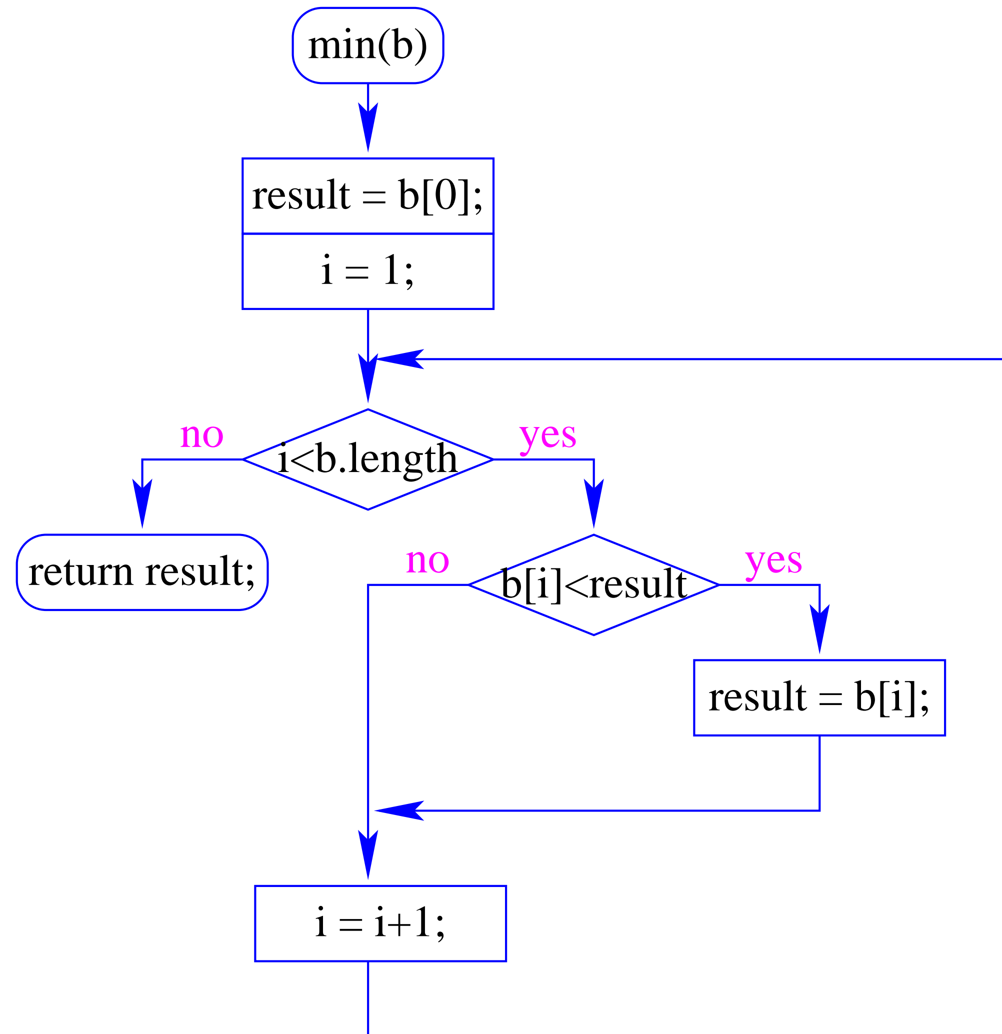
... die Ausgabe: Hello World!

Um die Arbeitsweise von Funktionen zu veranschaulichen, erweitern/modifizieren wir die Kontrollfluss-Diagramme:

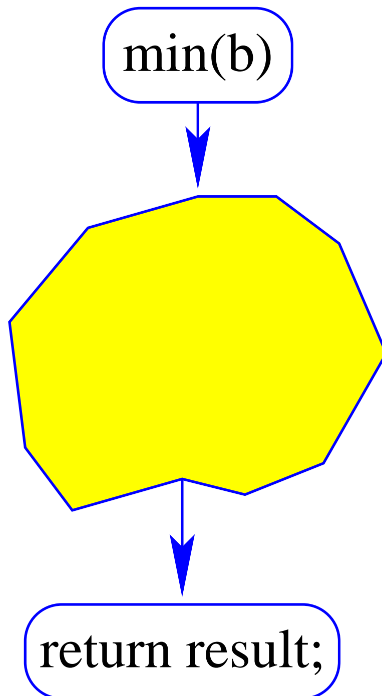


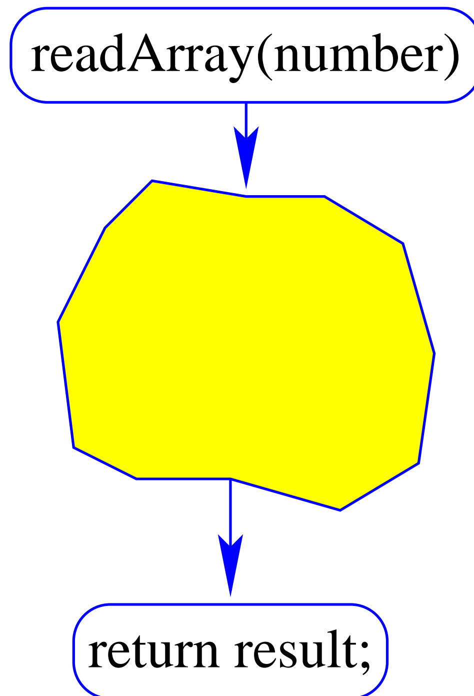
- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.

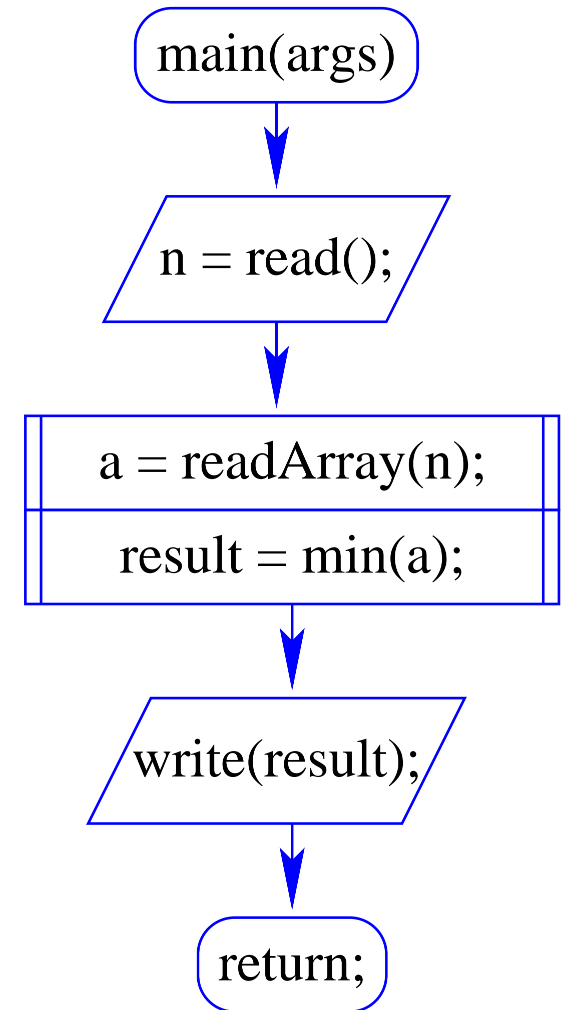
Teildiagramm für die Funktion `min()`:

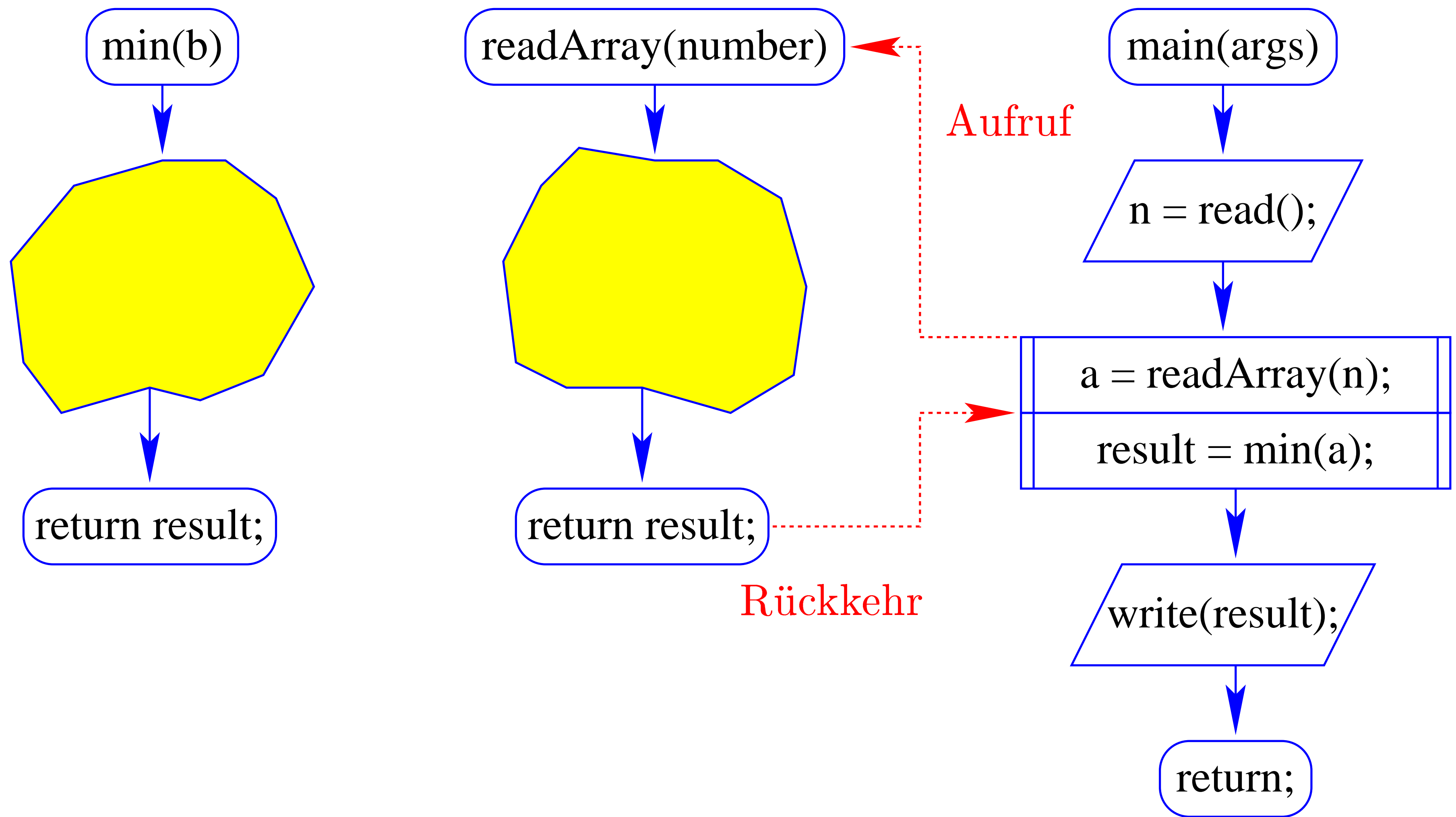


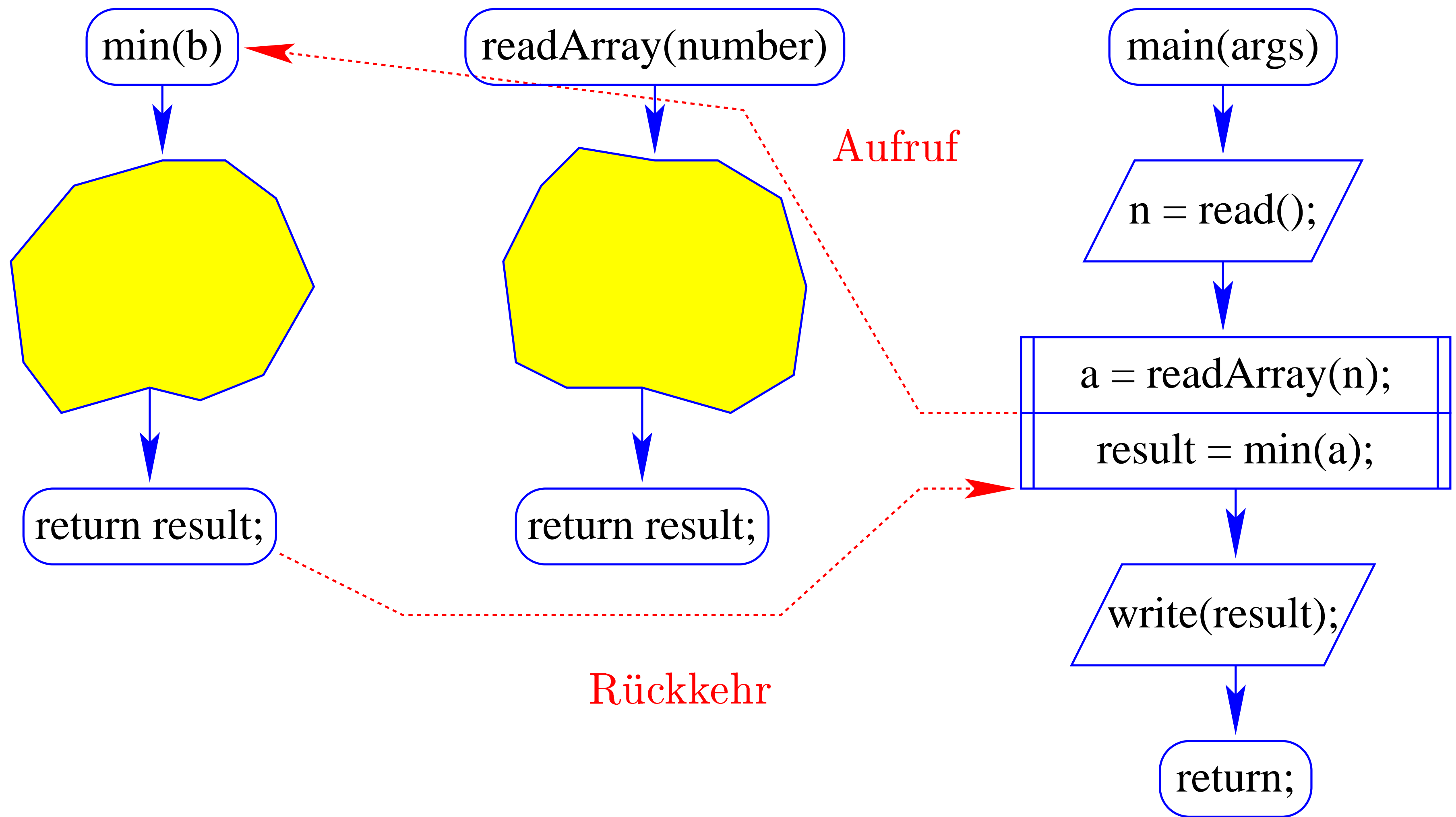
Insgesamt erhalten wir:








min(b)
return result;
readArray(number)
return result;
Aufruf
Rückkehr
main(args)
n = read();
a = readArray(n);
result = min(a);
write(result);
return;


min(b)
readArray(number)
main(args)
Aufruf
n = read();
a = readArray(n);
result = min(a);
return result;
return result;
write(result);
Rückkehr
return;