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.

```
expr    ::=  number | name | ( expr ) |
             unop expr | expr binop expr
unop    ::=  -
binop   ::=  - | + | * | / | %
```

- 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.

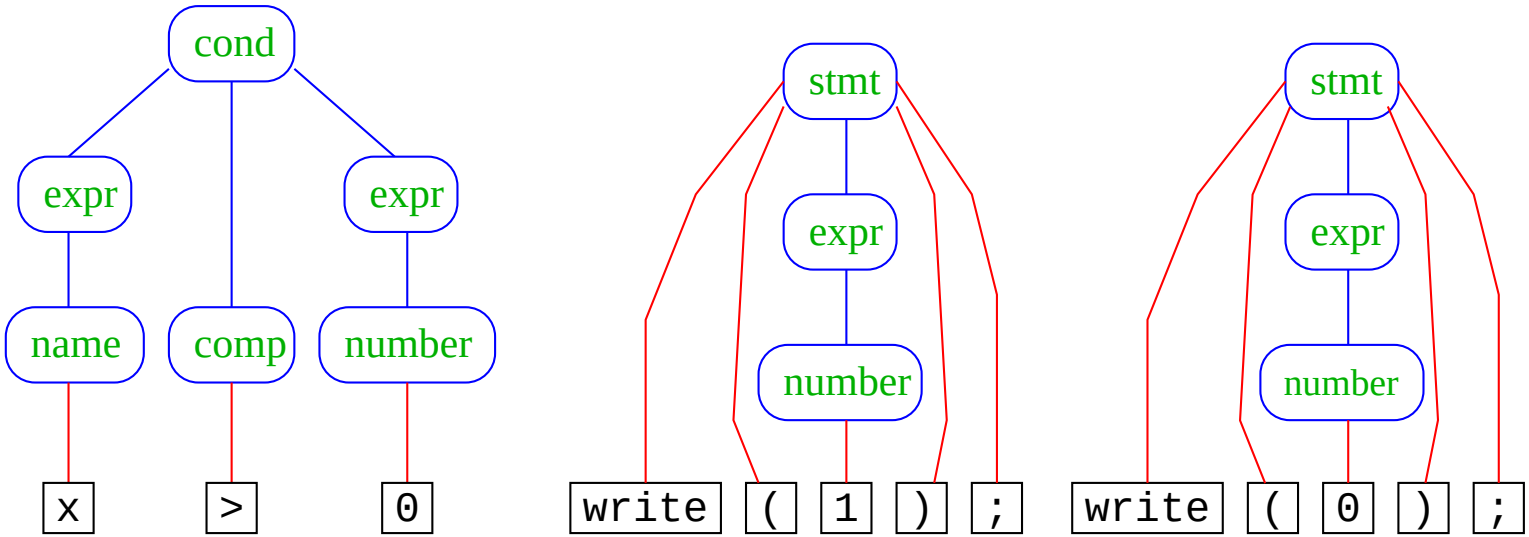
Puh!!! Geschafft ...

Beispiel:

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

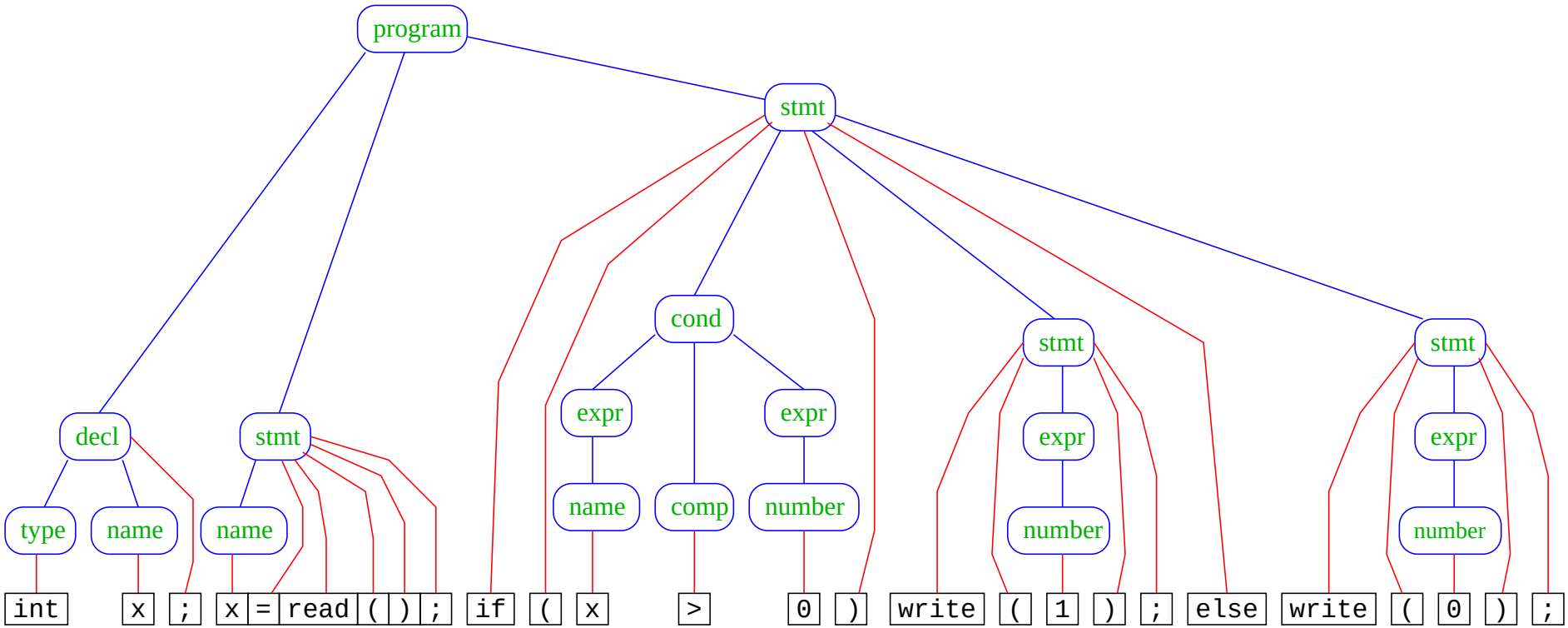
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

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
type
name
name
expr
expr
expr
expr
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.

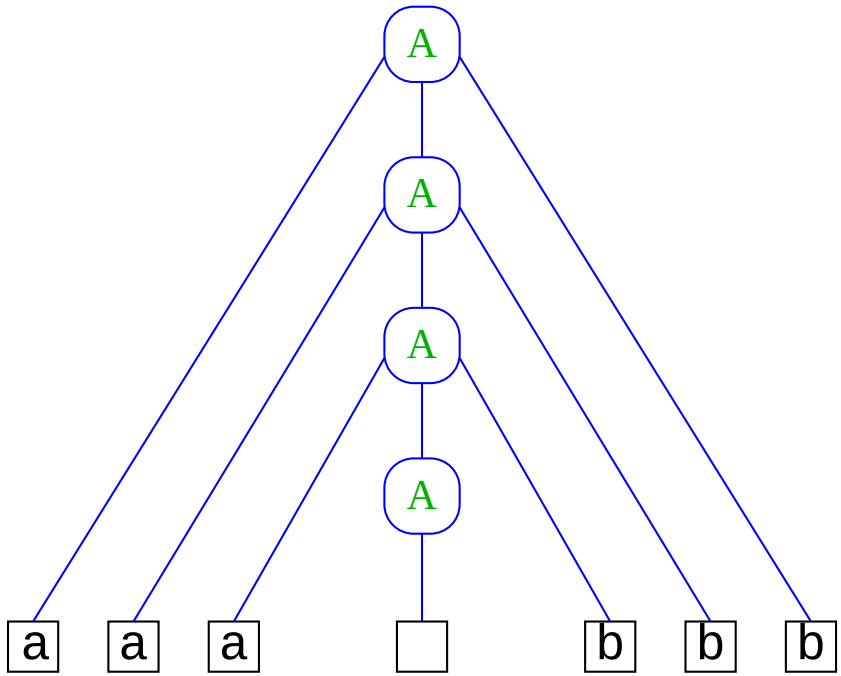
Beispiel:

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

lässt sich mithilfe einer Grammatik beschreiben:

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

Syntax-Baum für das Wort `aaabbb` :

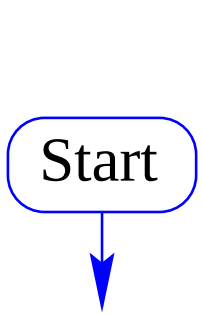


Für $\mathcal{L}$ gibt es aber keinen regulären Ausdruck!!! (↑Automatentheorie)

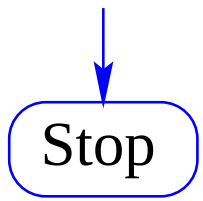
4 Kontrollfluss-Diagramme

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

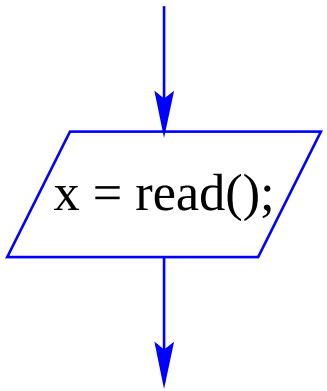
Ingredienzien:



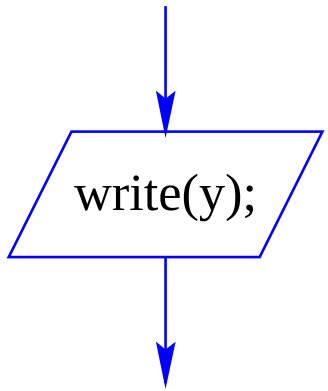
Startknoten



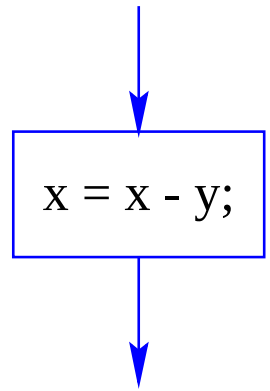
Endknoten



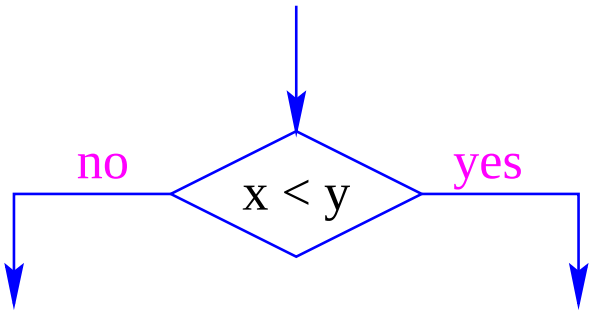
Eingabe



Ausgabe

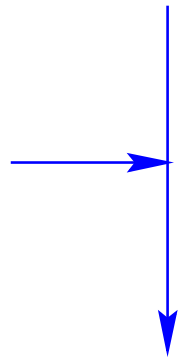


Zuweisung



bedingte Verzweigung

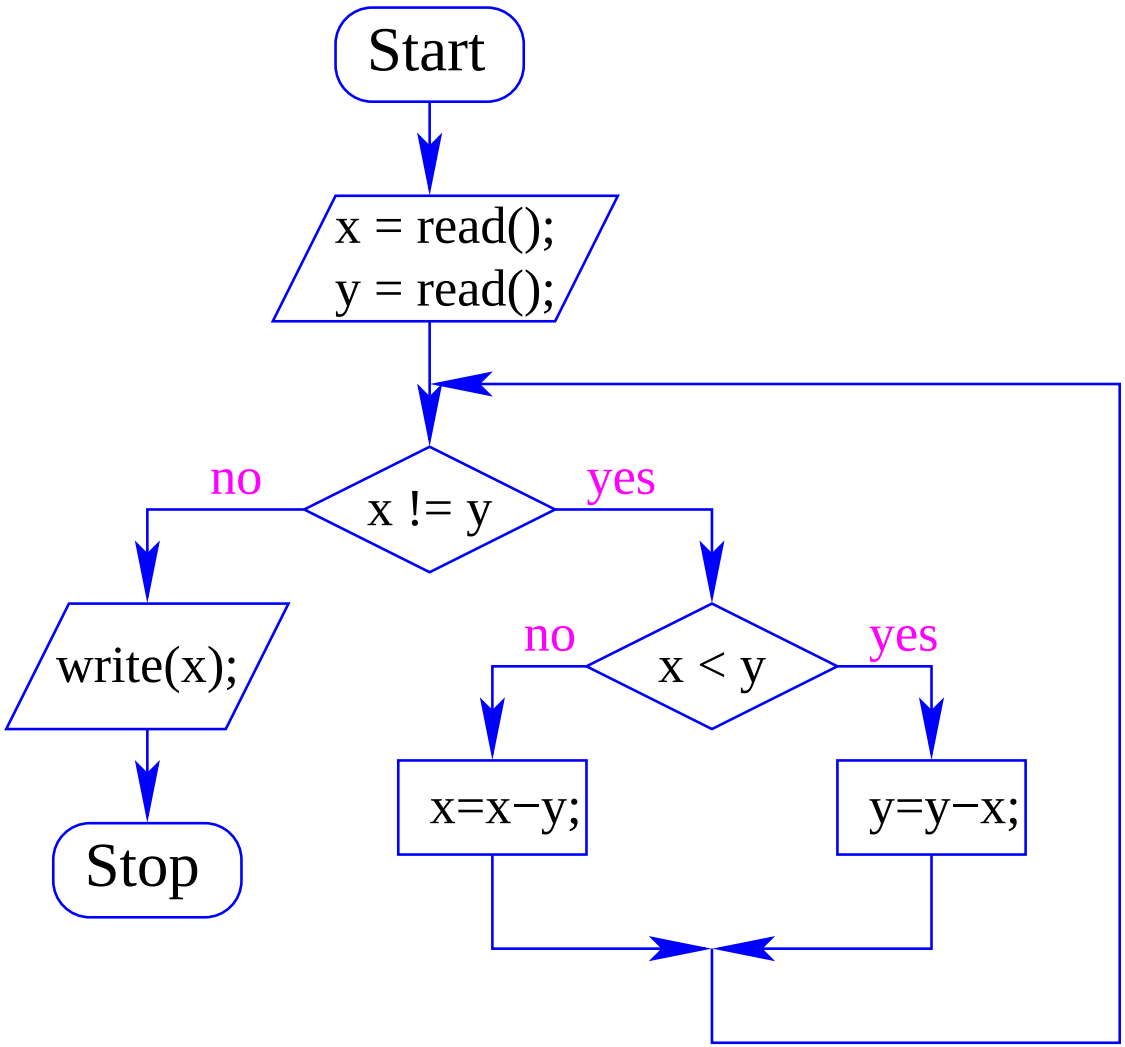
Kante



Zusammenlauf

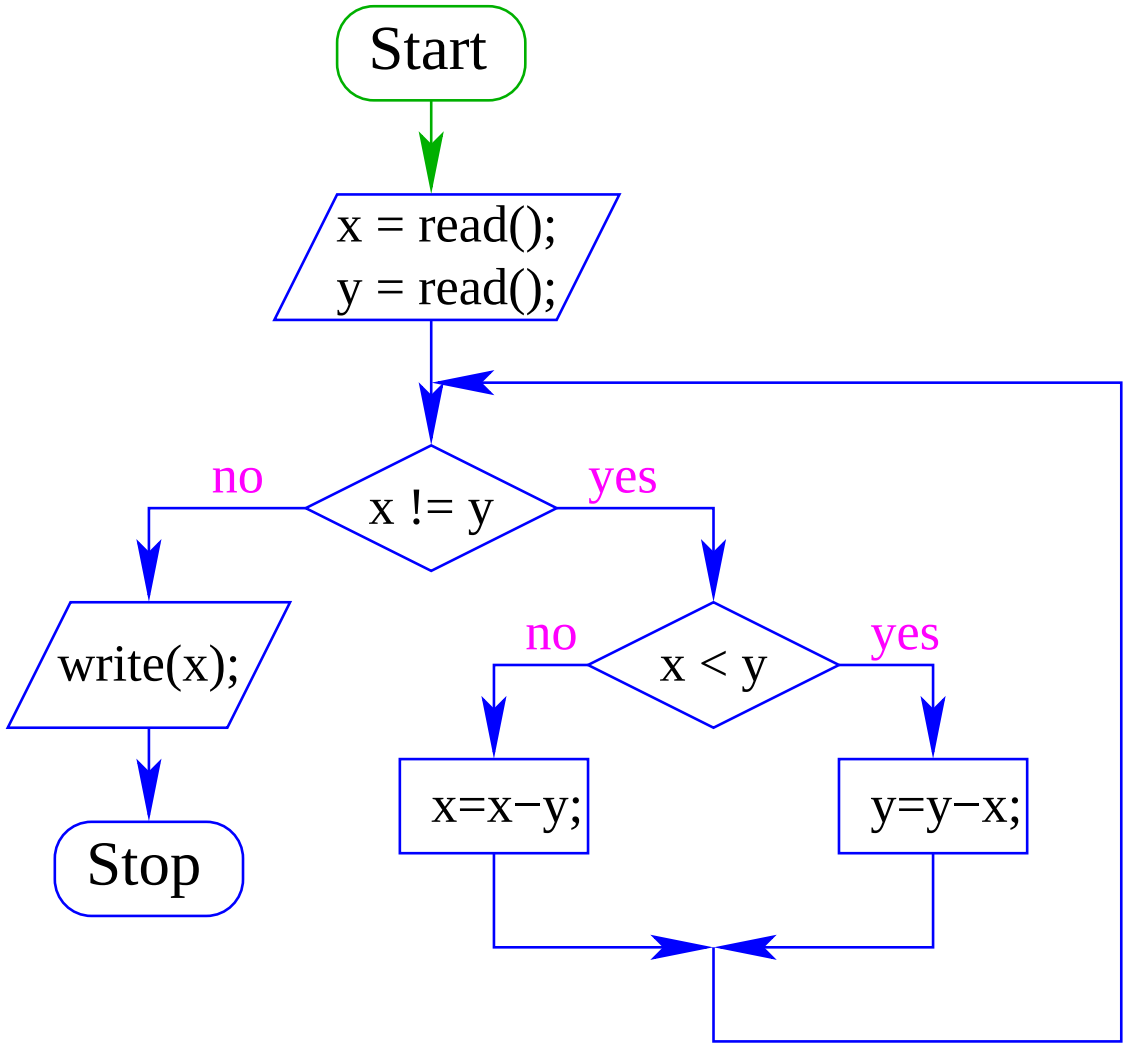
Beispiel:

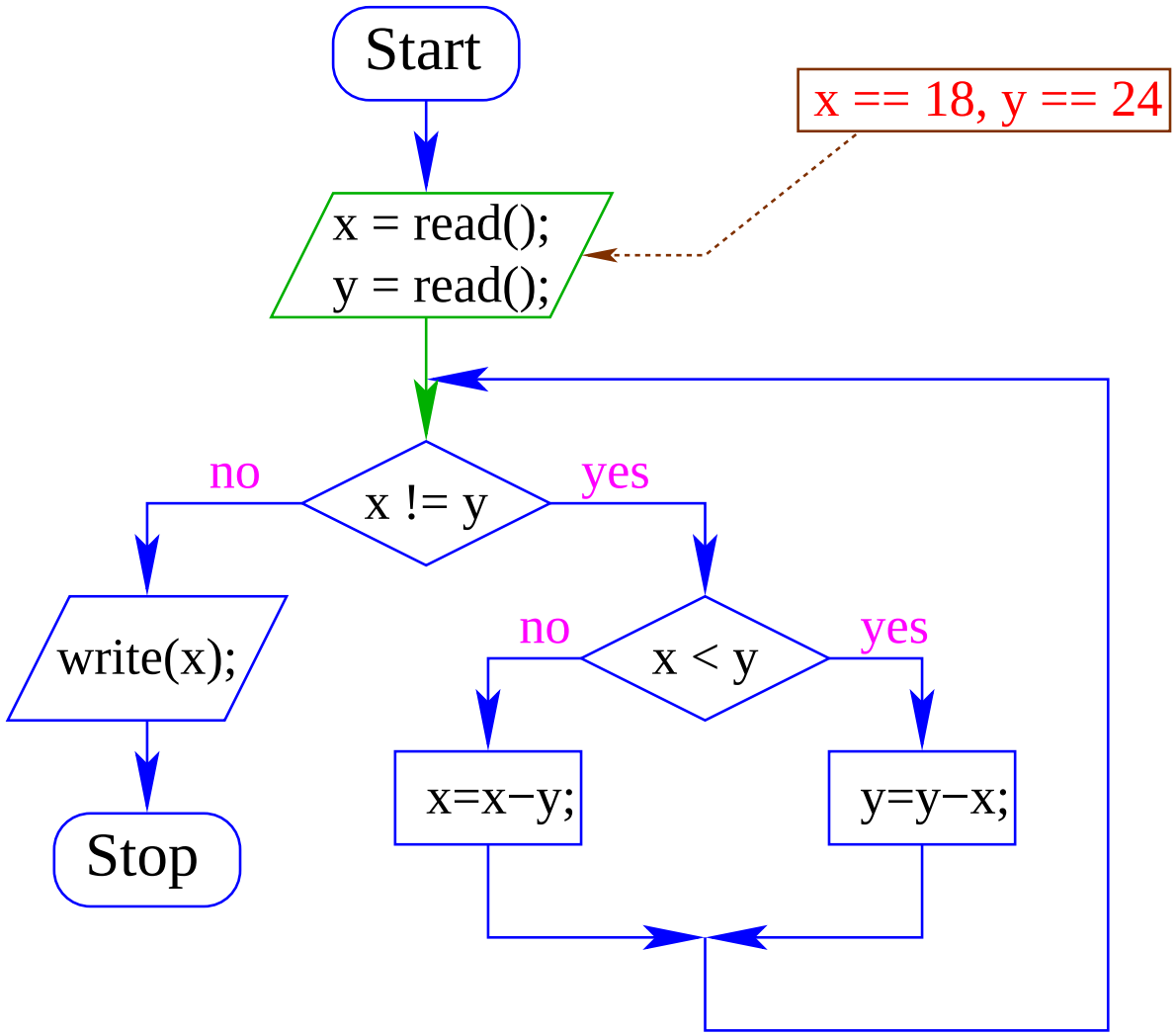
```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

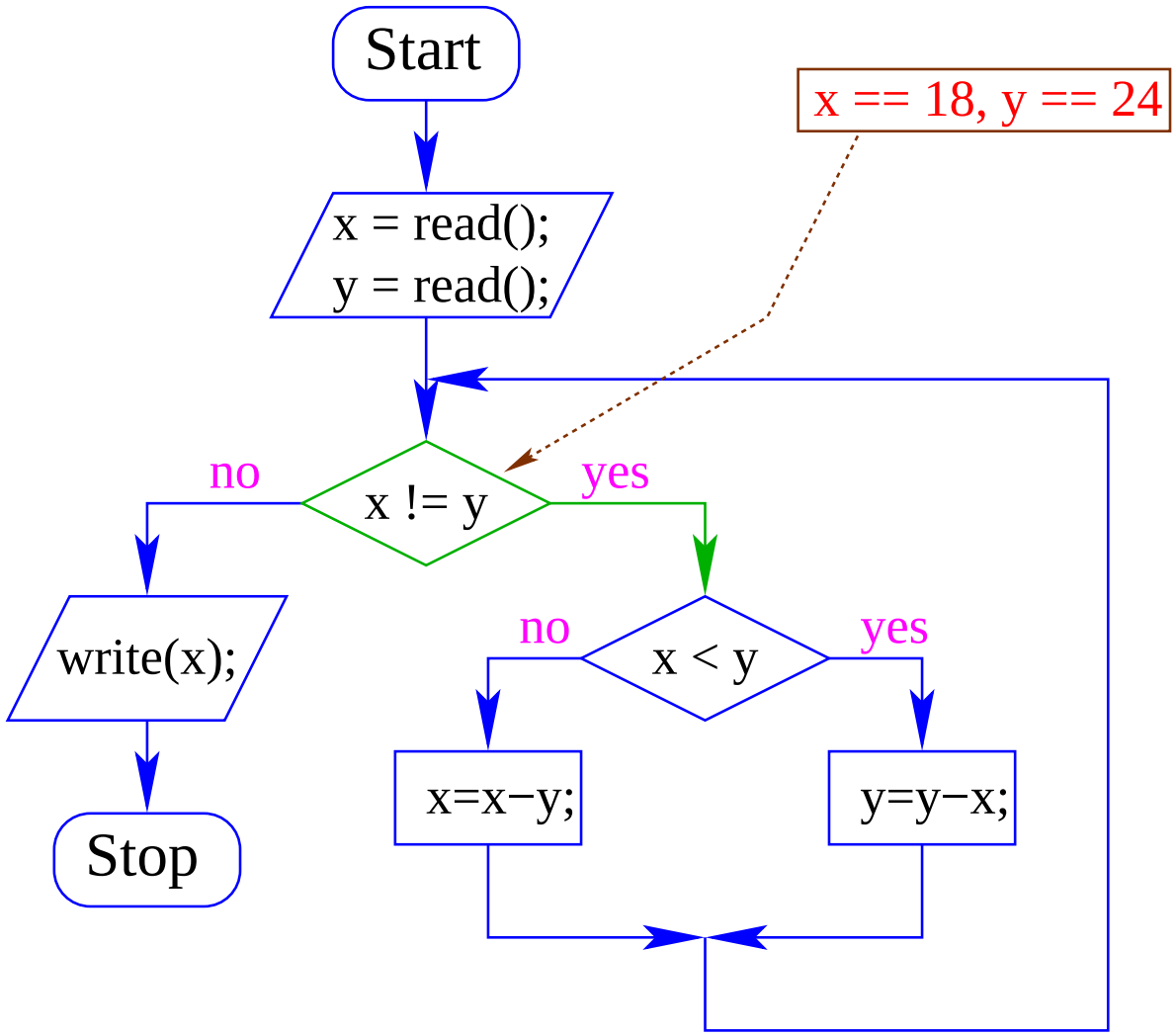


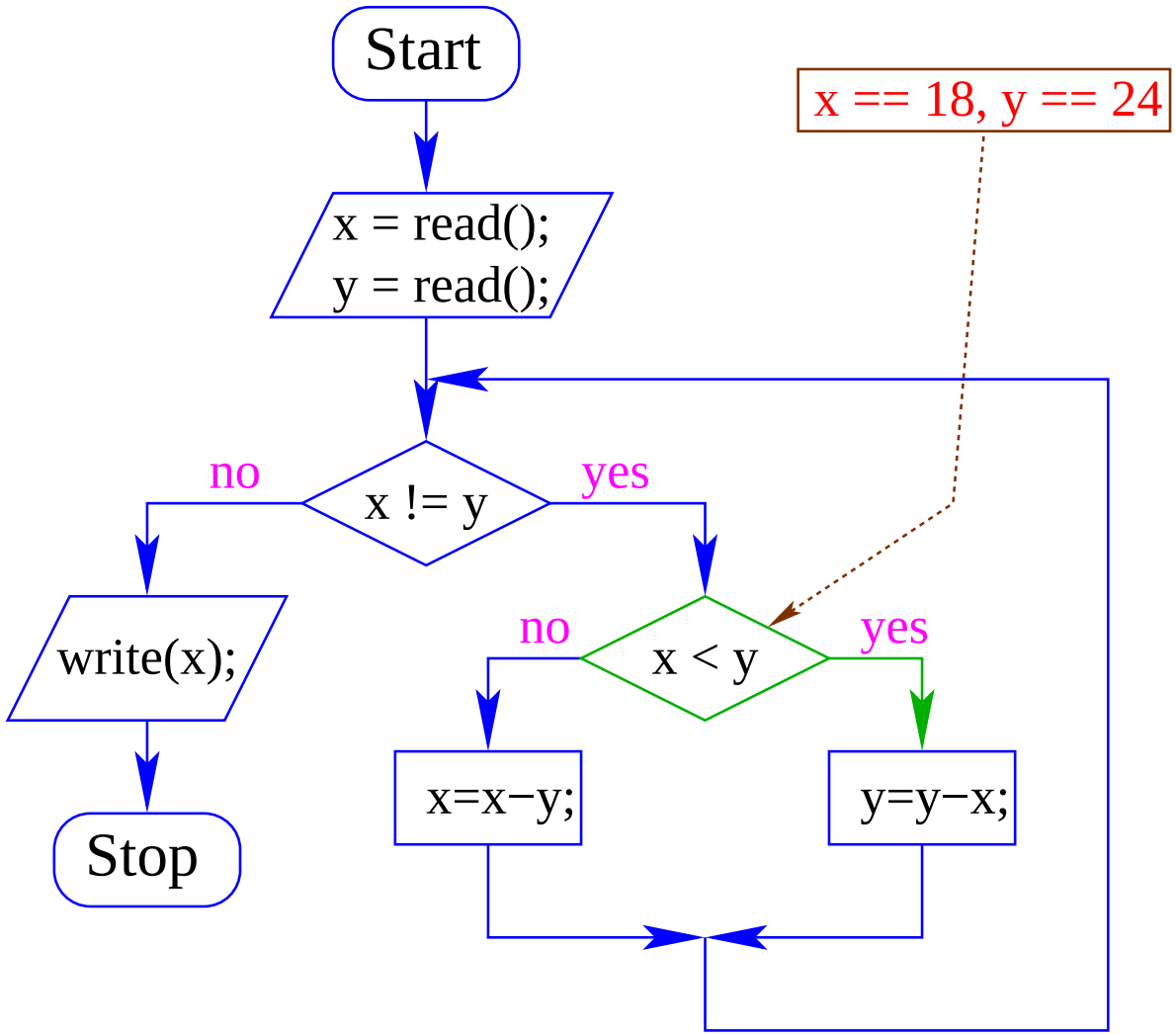
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollfluss-Diagramm vom Startknoten zum Endknoten.
- Die Deklarationen von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer dem Start- und Endknoten) sind die dabei auszuführenden Operationen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

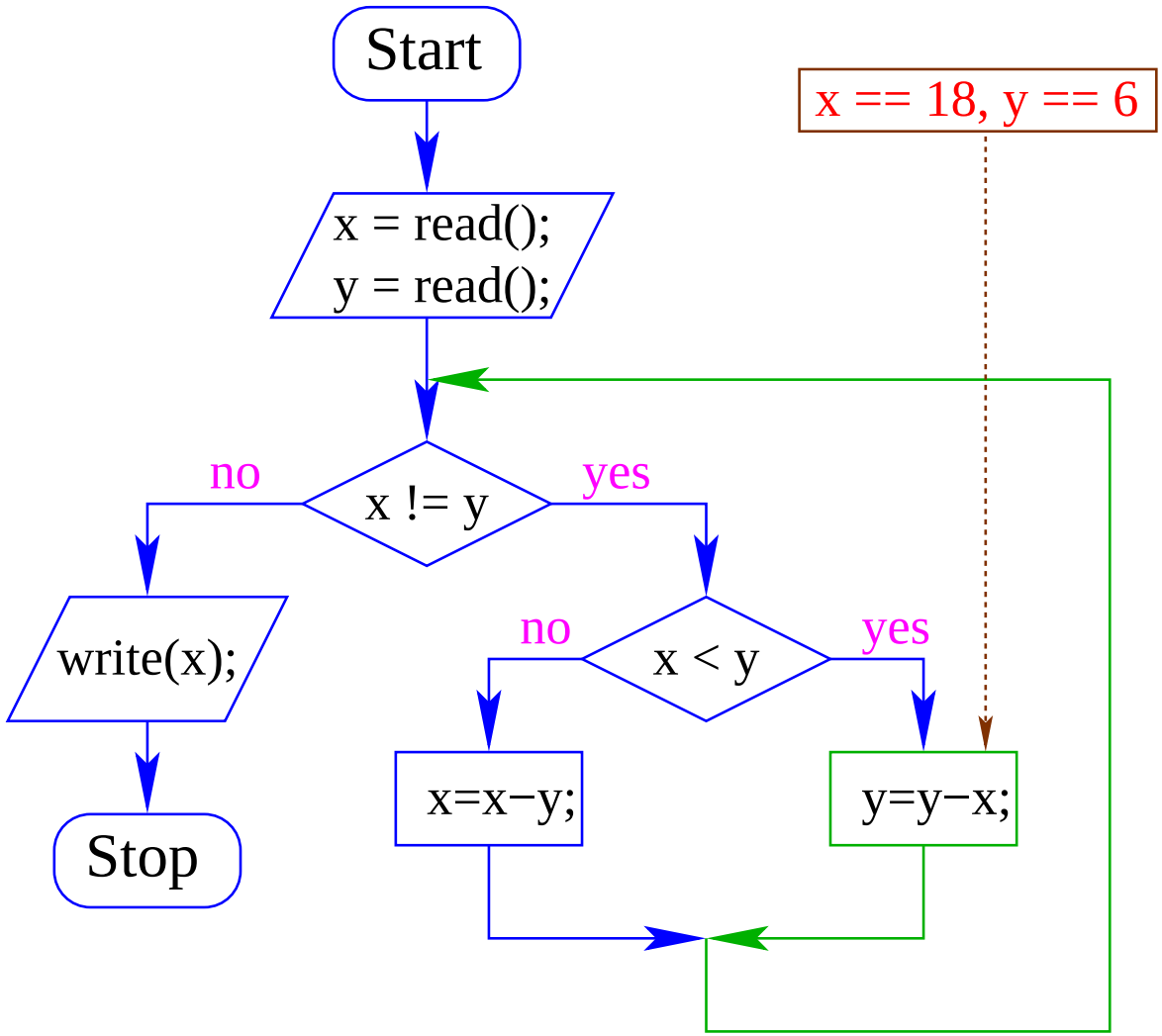
$\Longrightarrow$ operationelle Semantik

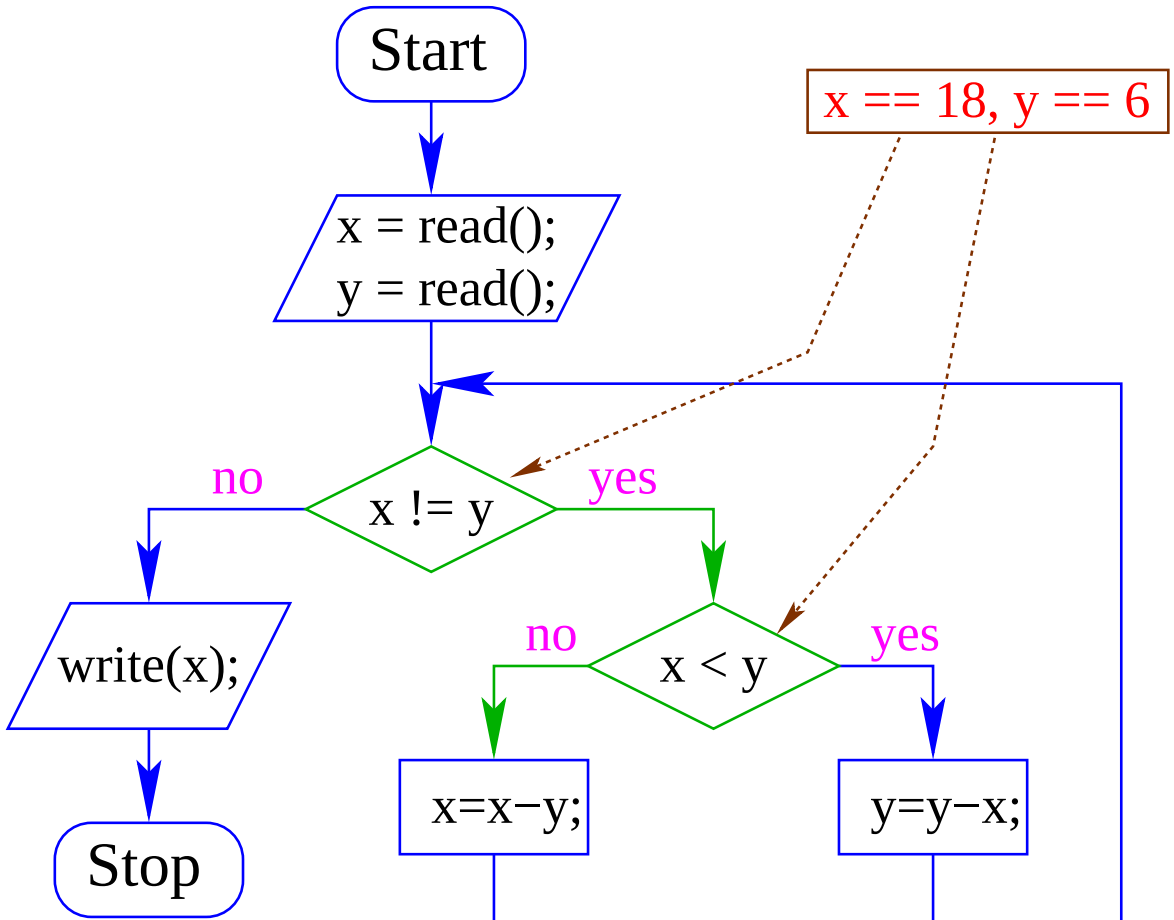

Start
x = read();
y = read();
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

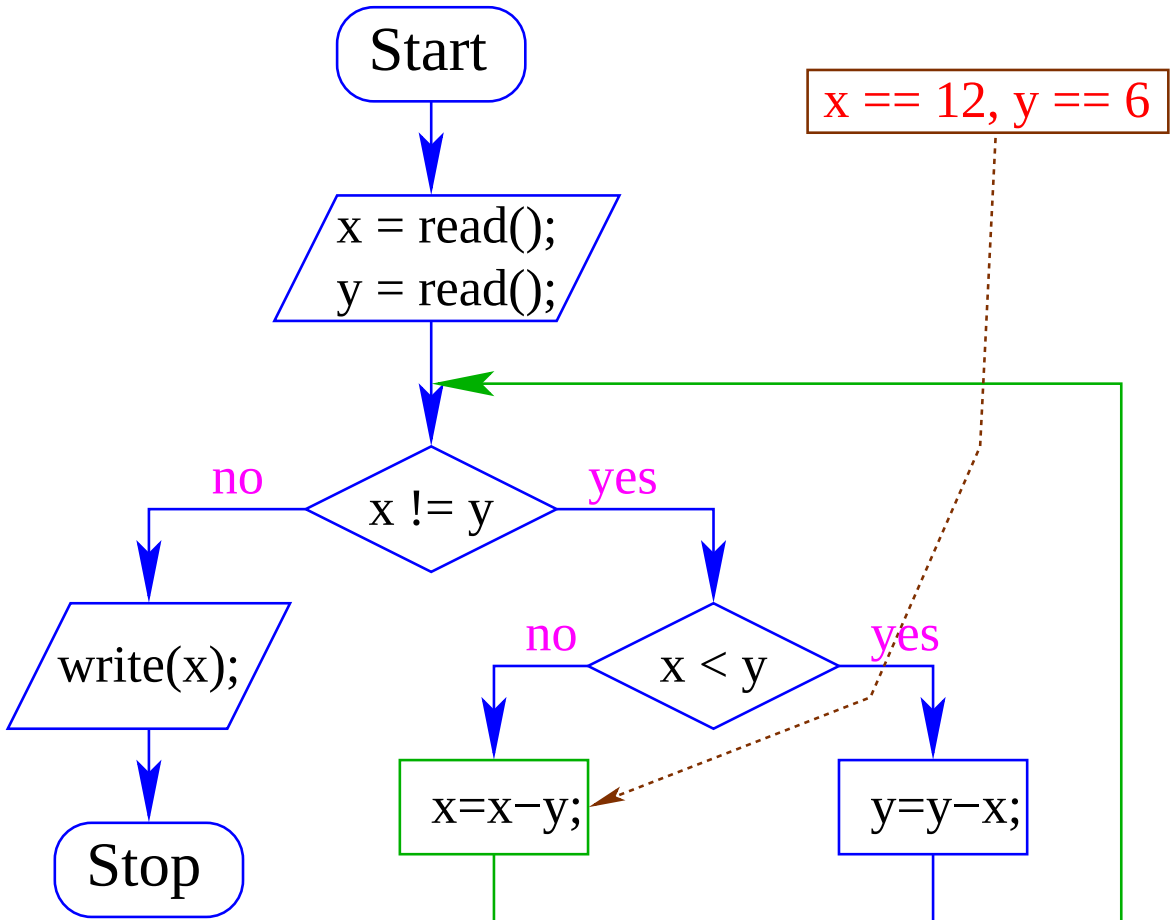

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

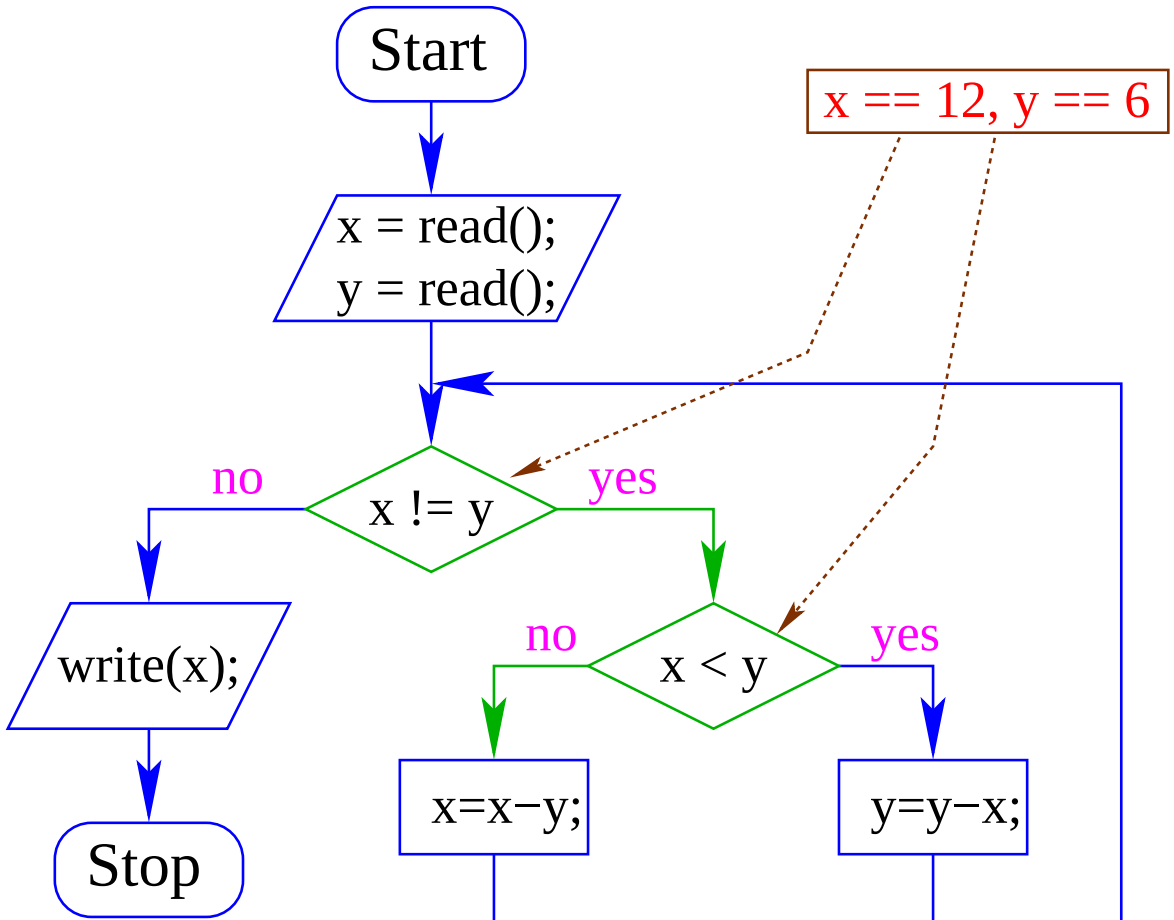

Start
x = read();
y = read();
x == 18, y == 24
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

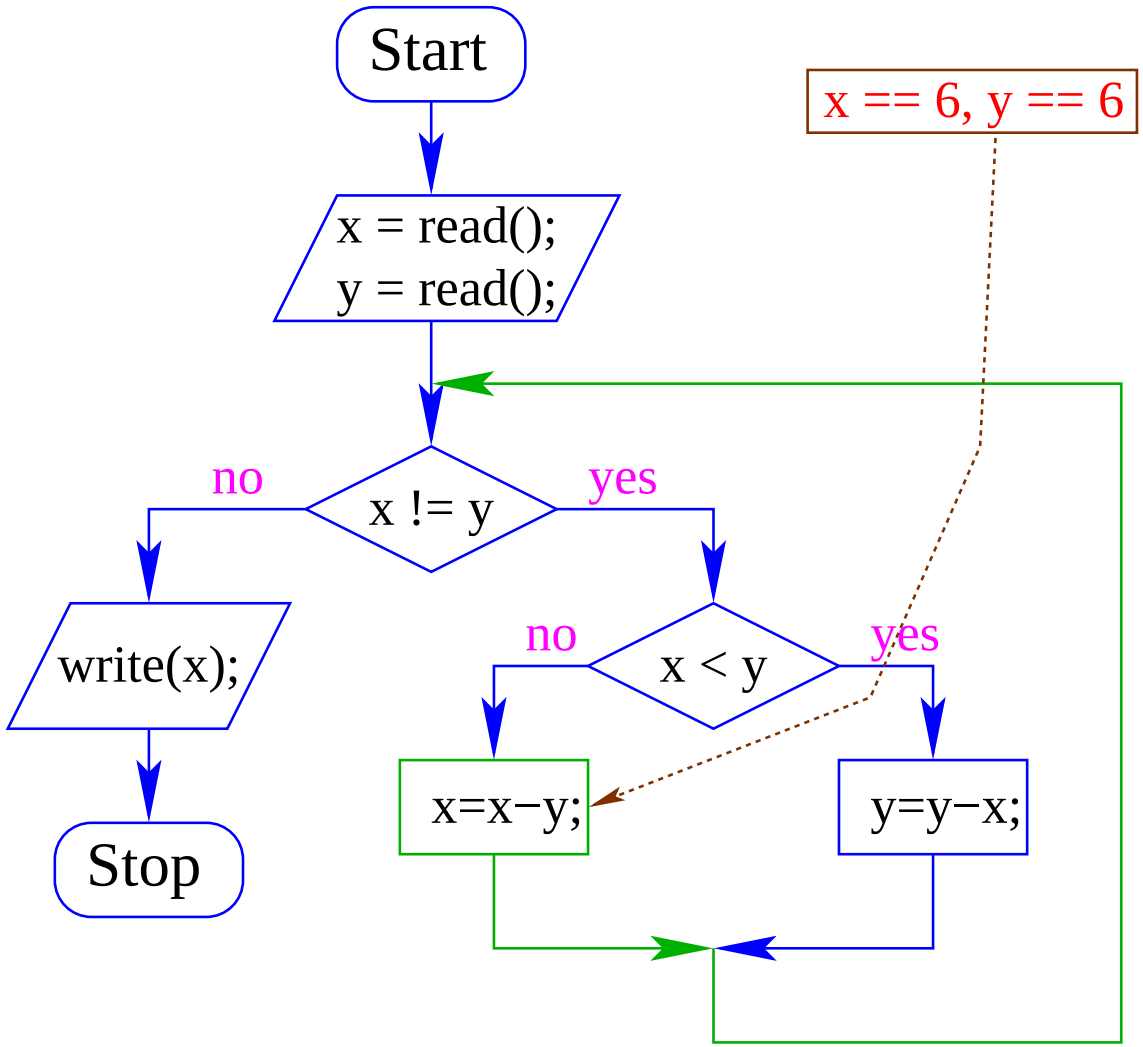

Start
x = read();
y = read();
x == 18, y == 24
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

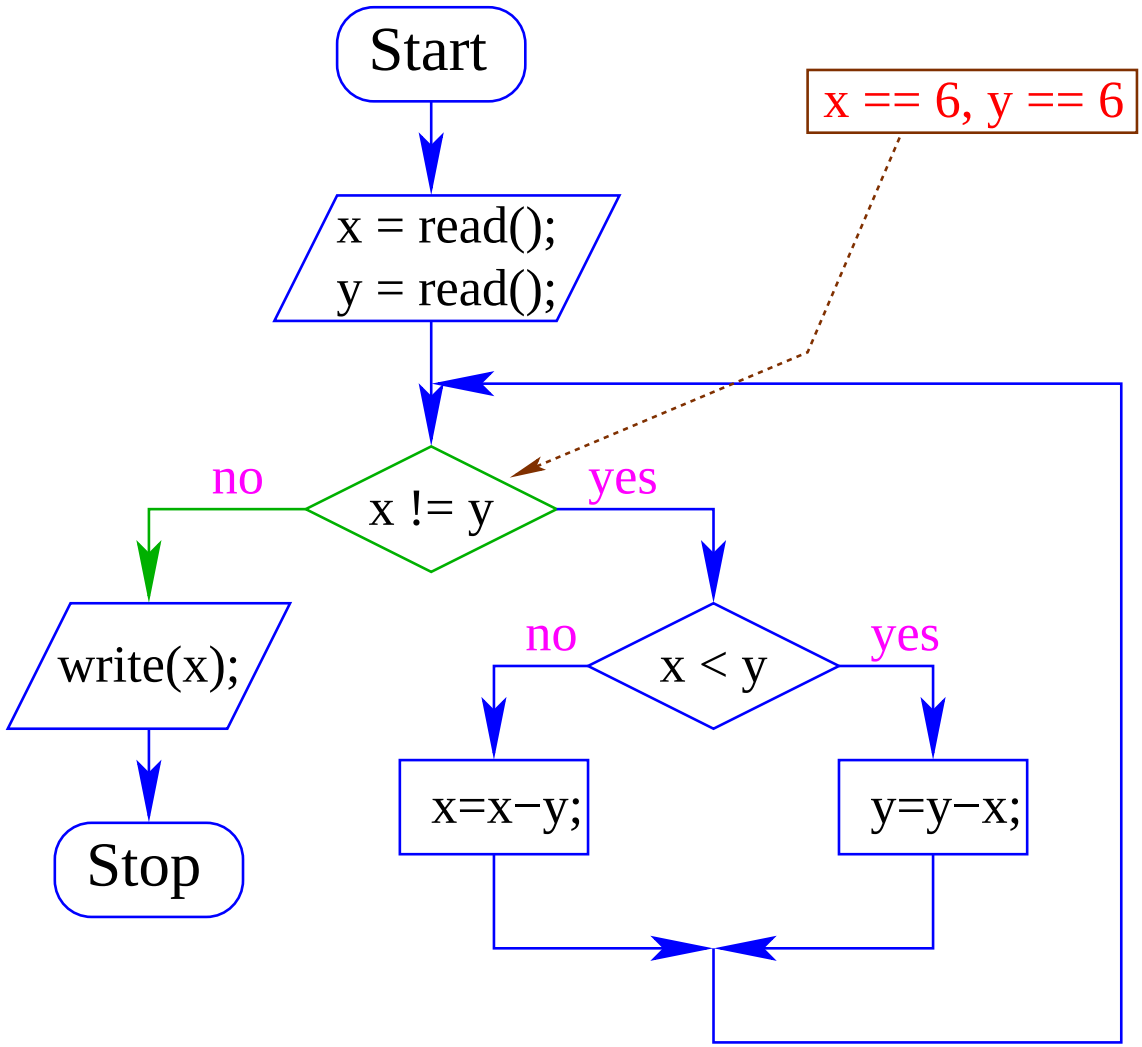

Start
x = read();
y = read();
x == 18, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

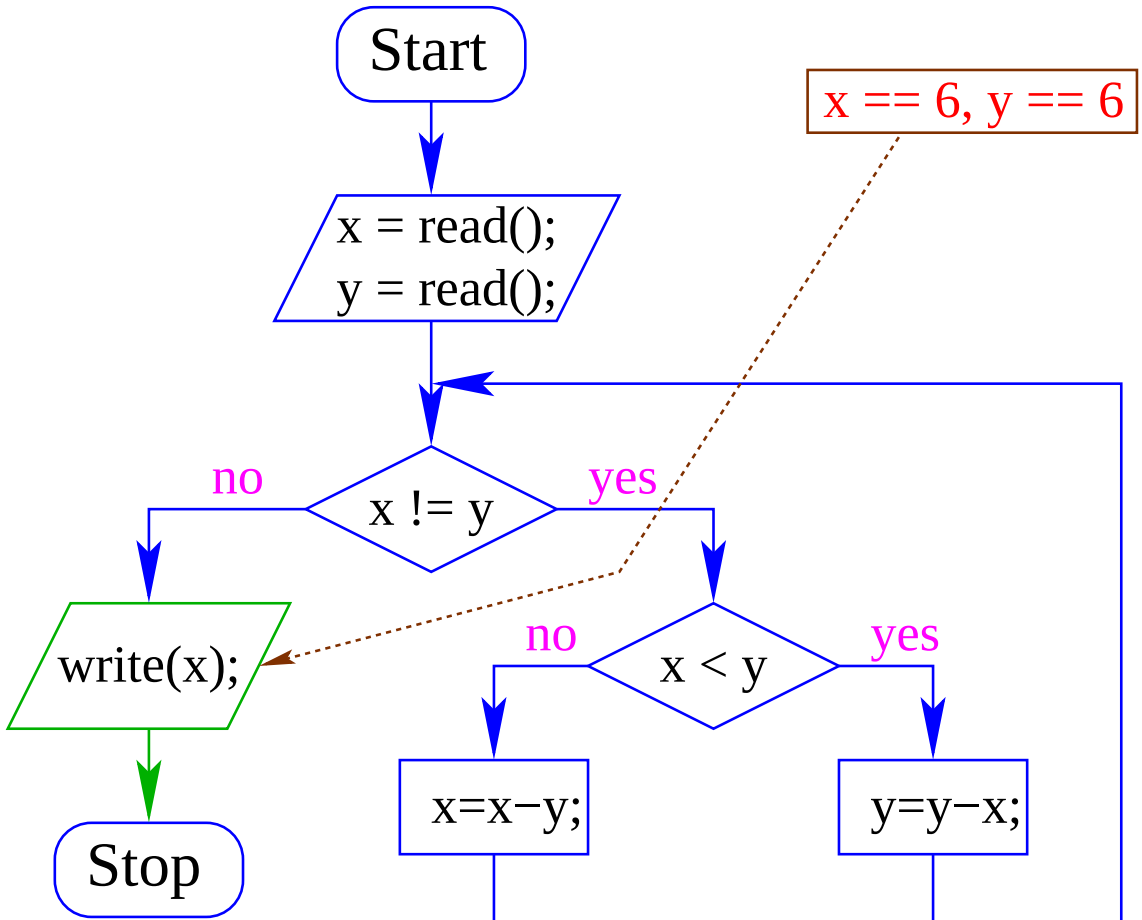

Start
x = read();
y = read();
x == 18, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;

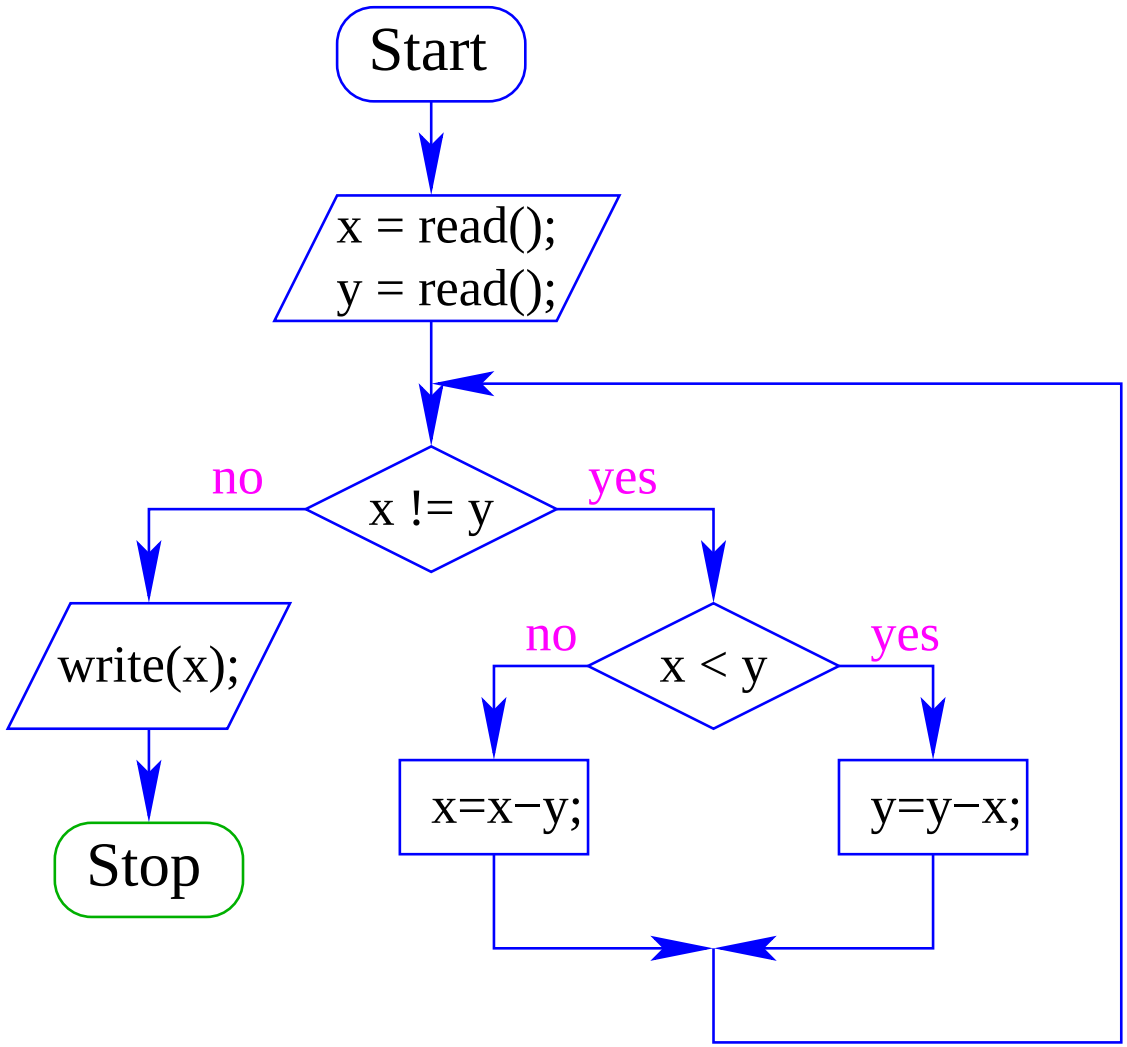

Start
x = read();
y = read();
x == 12, y == 6
x != y
no
yes
write(x);
Stop
x < y
no
yes
x=x−y;
y=y−x;


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


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


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

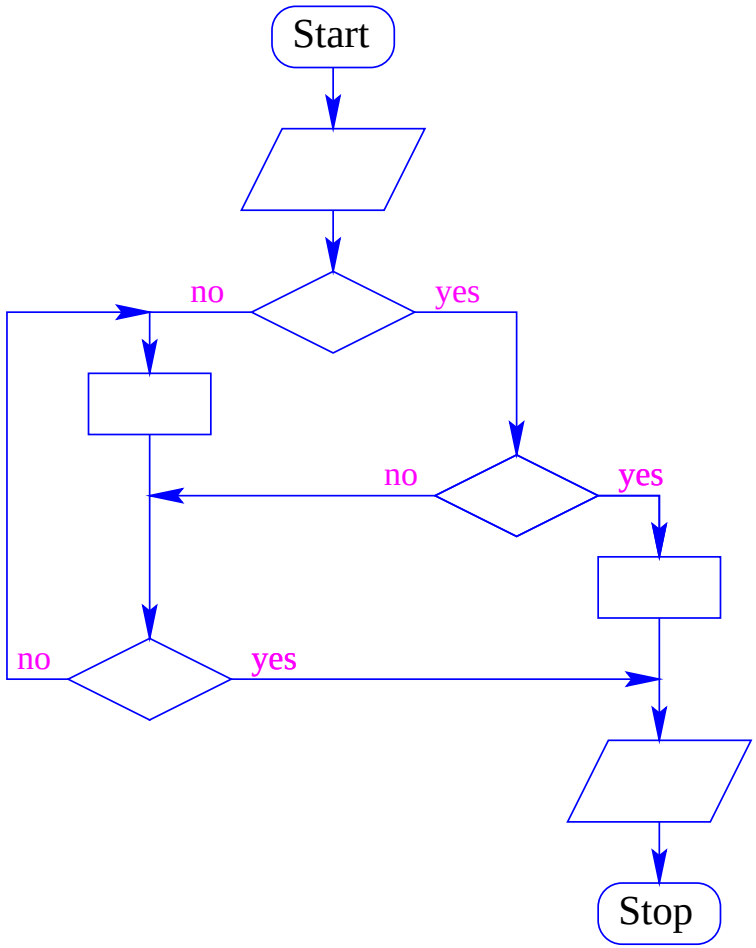

Start
x = read();
y = read();
x == 6, y == 6
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;


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

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.