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`)sign? digit digit*
  float ::= sign? digit digit* exp |
  sign? 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.

$$
\begin{array}{lcl}
\textit{stmt} & ::= & \texttt{;} \mid \texttt{\{} \; \textit{stmt}^* \; \texttt{\}} \mid \\
& & \textit{name} \; \texttt{=} \; \textit{expr} \texttt{;} \mid \textit{name} \; \texttt{=} \; \texttt{read();} \mid \texttt{write(} \; \textit{expr} \; \texttt{);} \mid \\
& & \texttt{if (} \; \textit{cond} \; \texttt{)} \; \textit{stmt} \mid \\
& & \texttt{if (} \; \textit{cond} \; \texttt{)} \; \textit{stmt} \; \texttt{else} \; \textit{stmt} \mid \\
& & \texttt{while (} \; \textit{cond} \; \texttt{)} \; \textit{stmt}
\end{array}
$$

- 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.

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

- 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.

$$
\begin{array}{lll}
\text{cond} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{(}\ \text{cond}\ \texttt{)} \mid \\
 & & \text{expr comp expr} \mid \\
 & & \text{bunop cond} \mid \text{cond bbinop cond} \\
\text{comp} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>} \\
\text{bunop} & ::= & \texttt{!} \\
\text{bbinop} & ::= & \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

- Bedingungen unterscheiden sich 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.

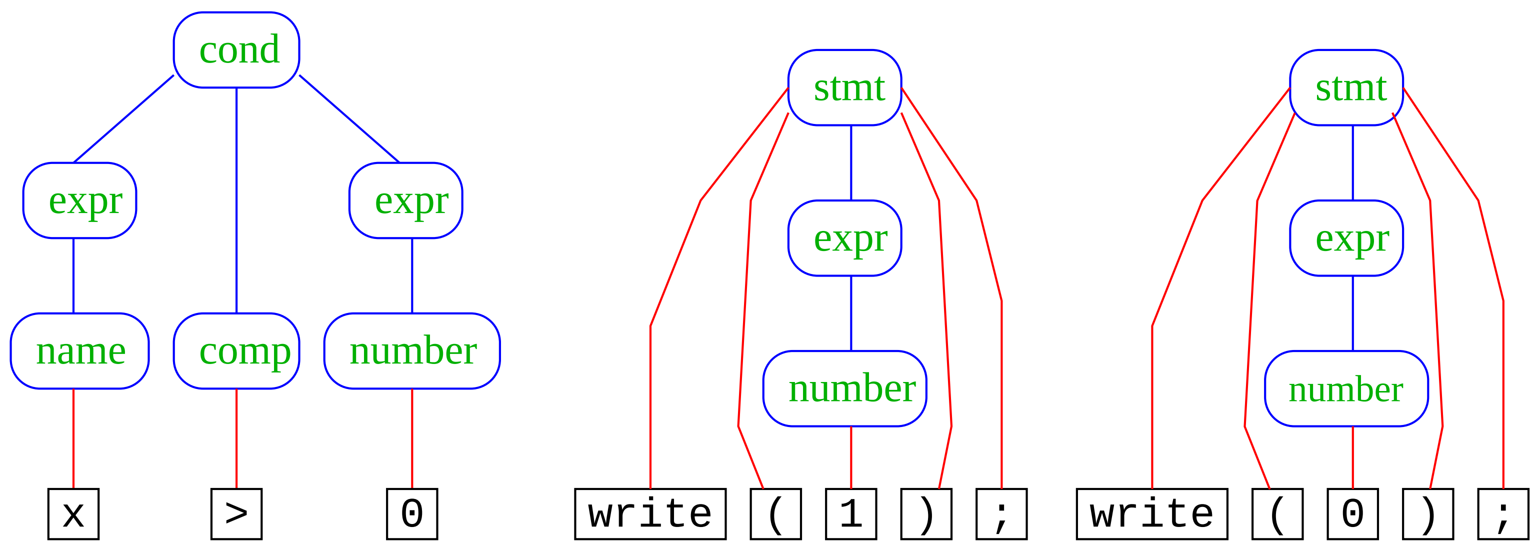
Puh!!! Geschafft ...

Beispiel:

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

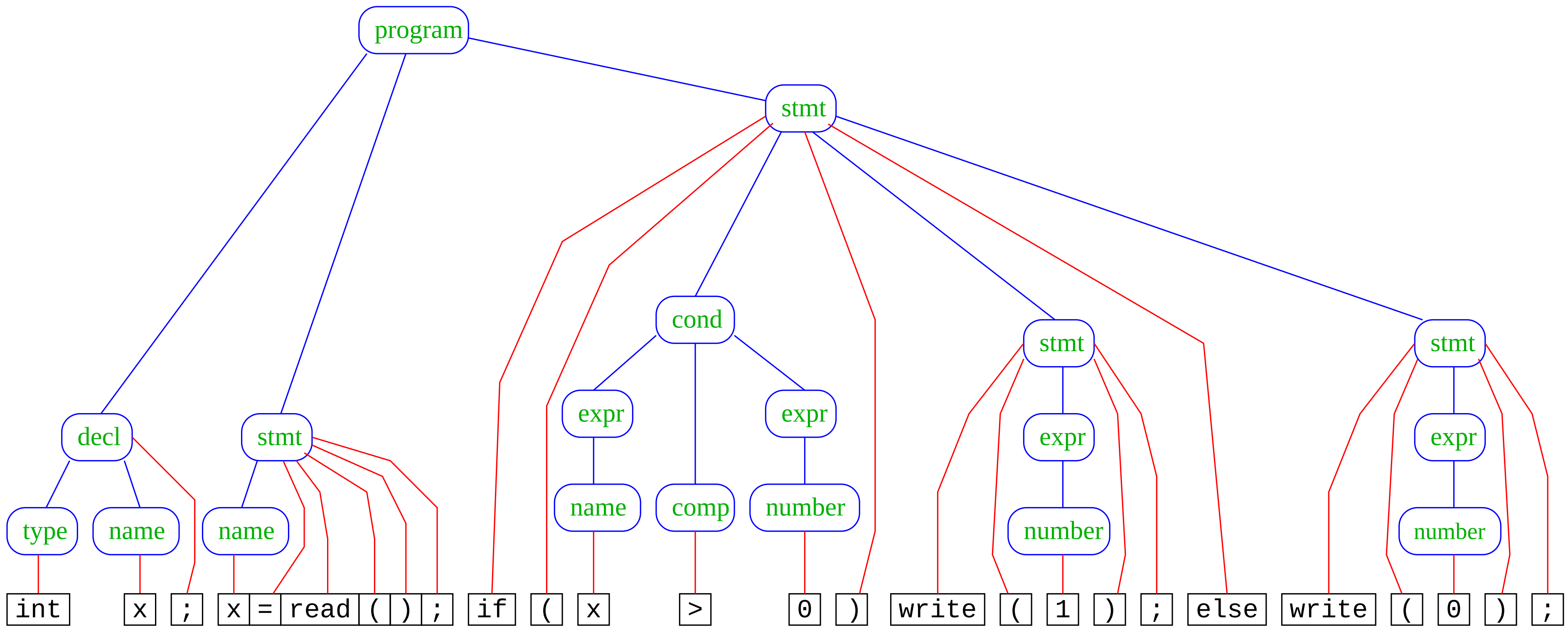
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

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 (Erfinder von Fortran)

Achtung:

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

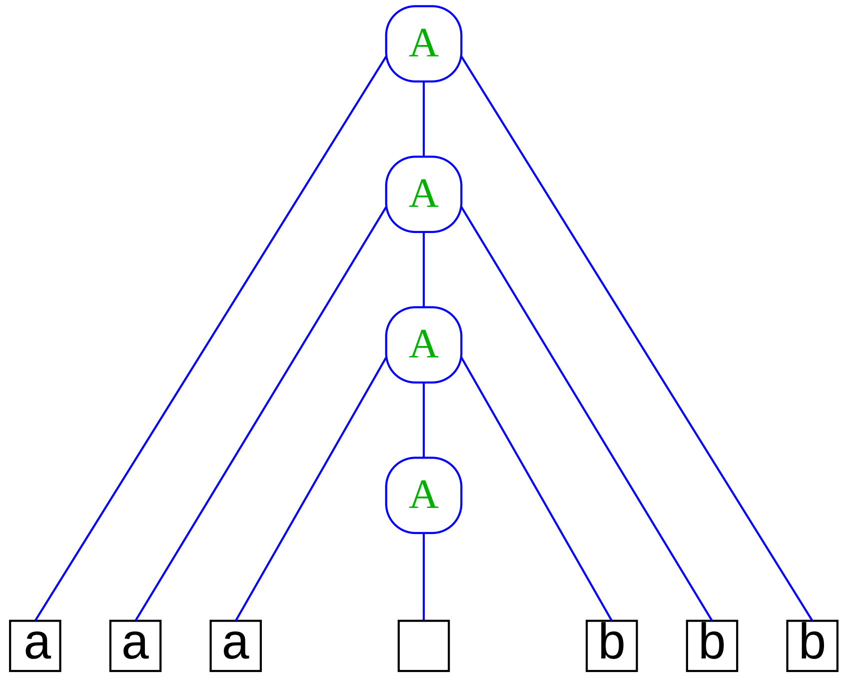
Beispiel:

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

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!!!
($\uparrow$Automatentheorie)

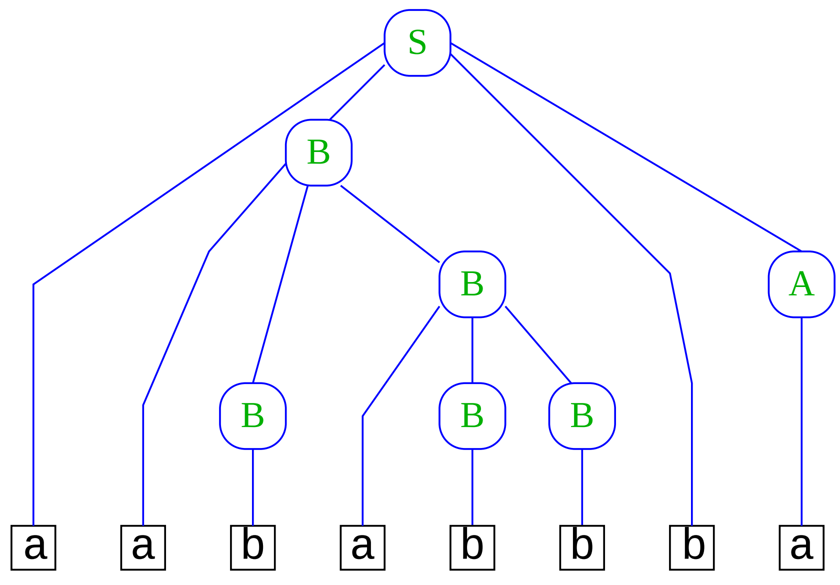
Weiteres Beispiel:

$$\mathcal{L} = \text{alle Worte mit gleich vielen } \texttt{a}\text{'s und } \texttt{b}\text{'s}$$

Zugehörige Grammatik:

$$
\begin{array}{lcl}
S & ::= & (\ \texttt{b}\ A \mid \texttt{a}\ B\ )^* \\
A & ::= & (\ \texttt{b}\ A\ A \mid \texttt{a}\ ) \\
B & ::= & (\ \texttt{a}\ B\ B \mid \texttt{b}\ )
\end{array}
$$

Syntax-Baum für das Wort `aababbba` :

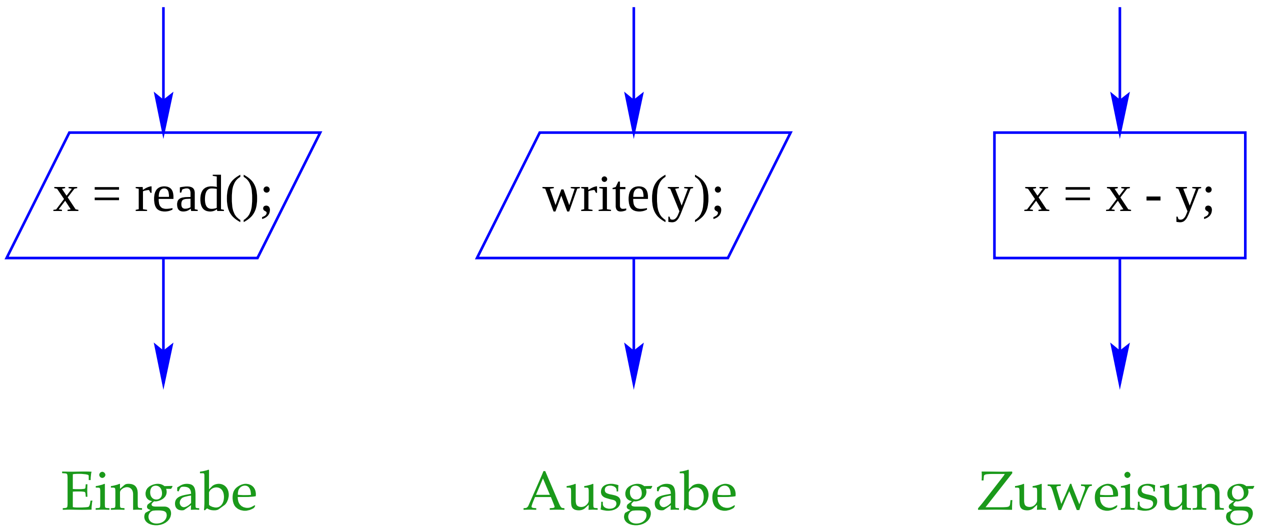


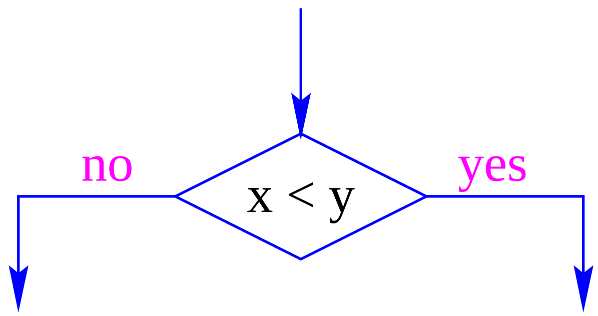
4 Kontrollfluss-Diagramme

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

Ingredienzien:

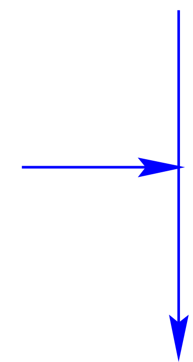
Startknoten Endknoten


x = read();
write(y);
x = x - y;
Eingabe
Ausgabe
Zuweisung



bedingte Verzweigung

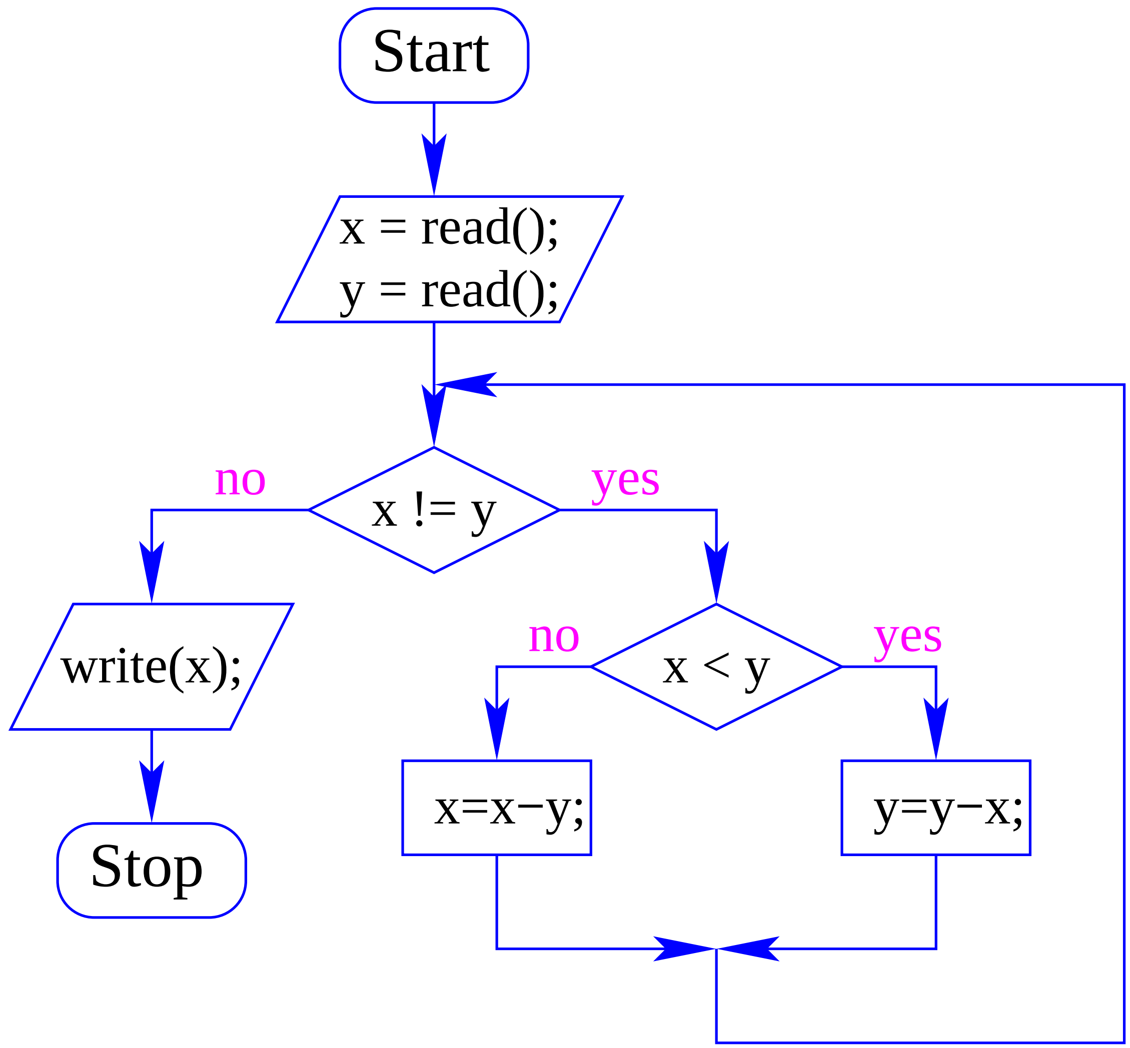
Kante



Zusammenlauf

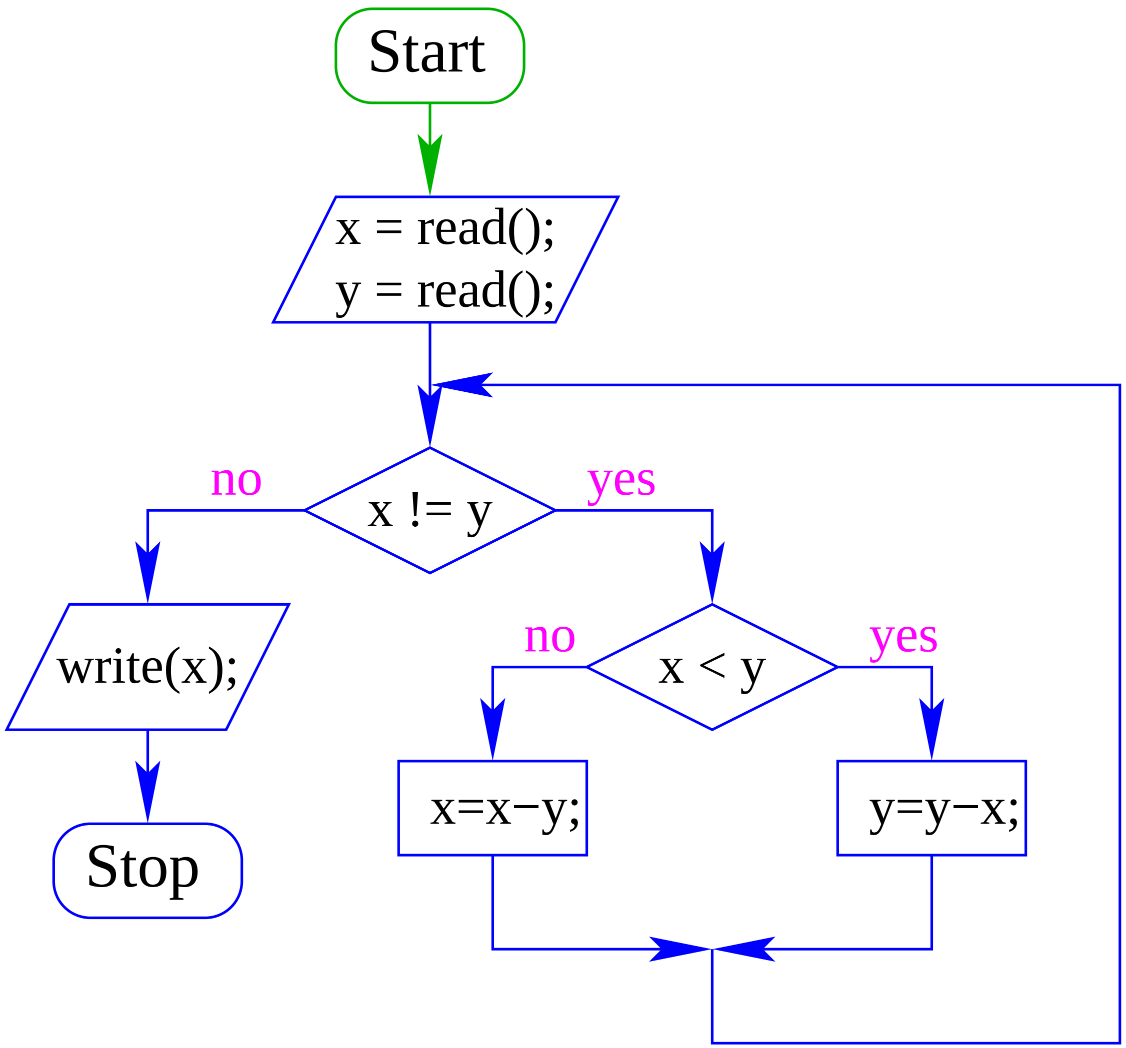
Beispiel:

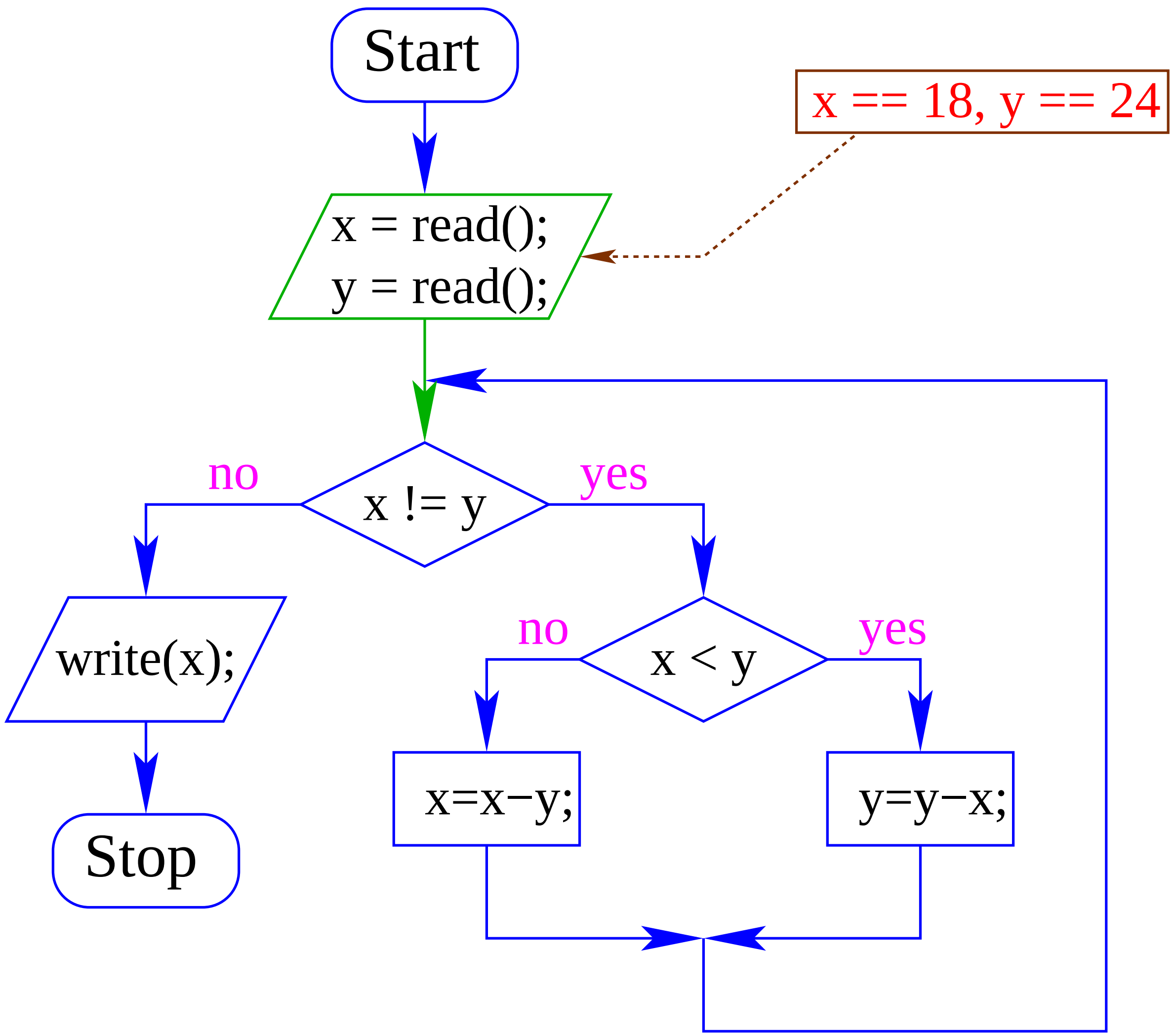
```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

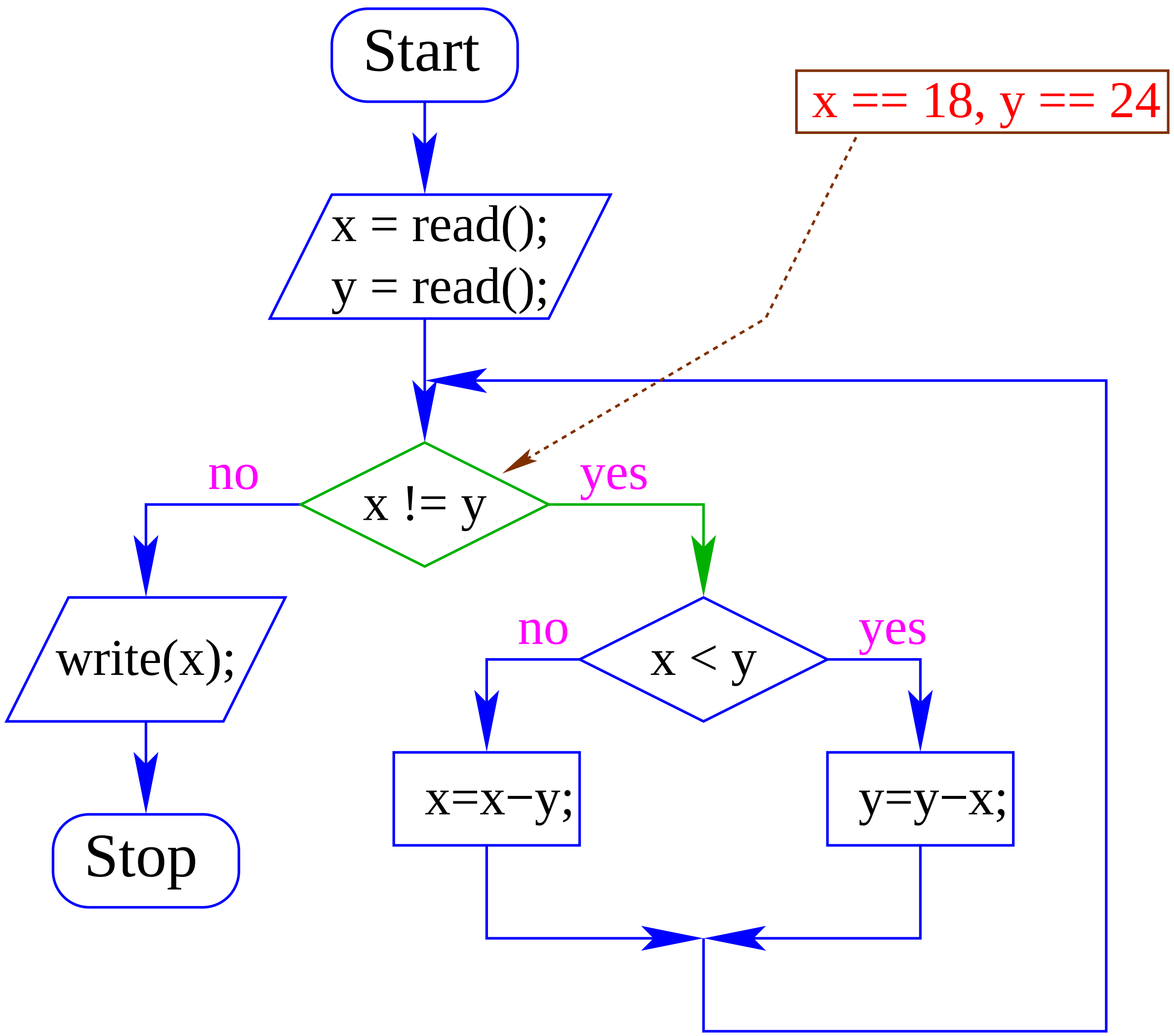


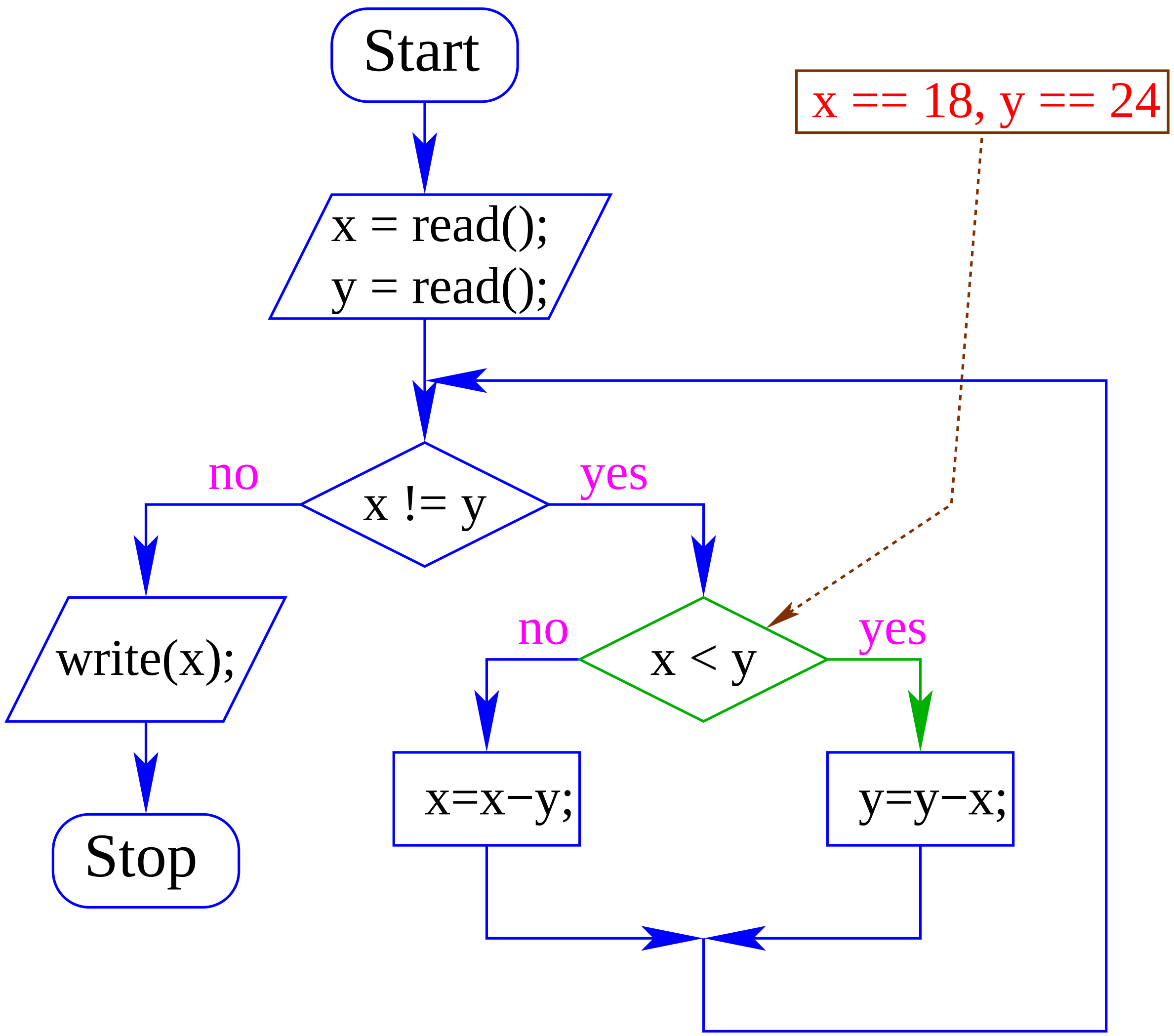
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollfluss-Diagramm vom Startknoten zum Endknoten.
- Die Deklarationen von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer dem Start- und Endknoten) sind die dabei auszuführenden Operationen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

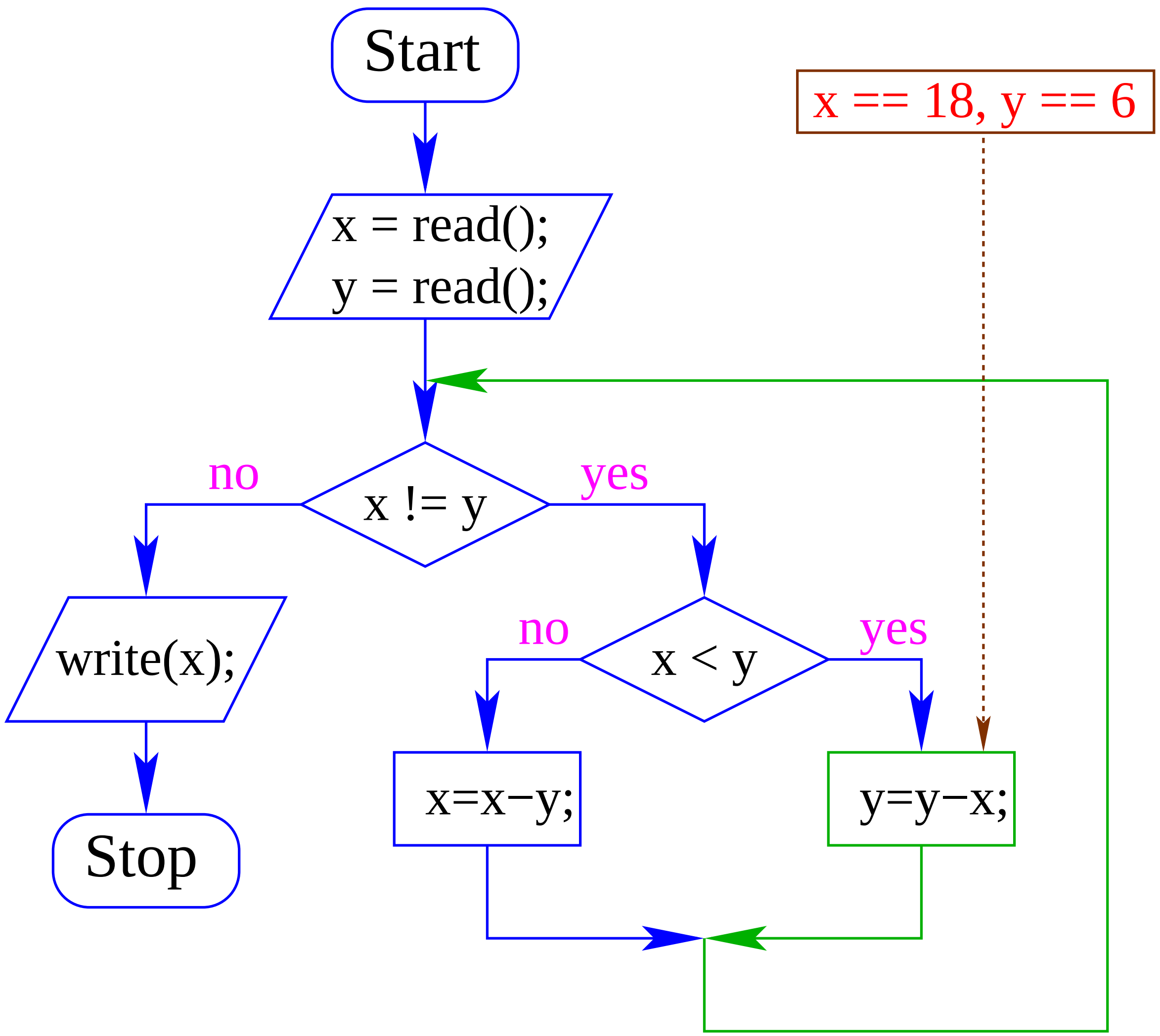
$\Longrightarrow$ operationelle Semantik

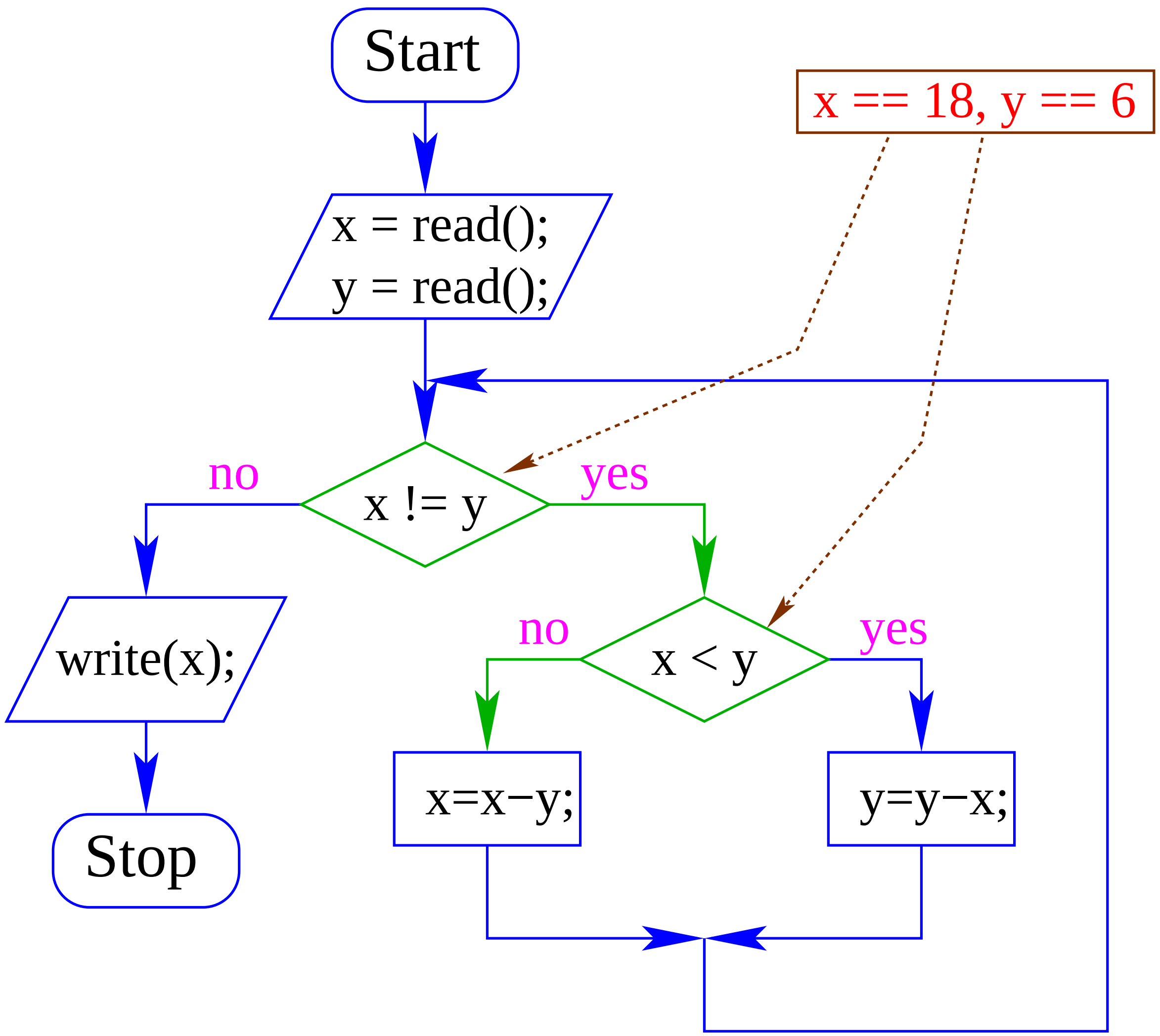

Start
x = read();
y = read();
no
x != y
yes
write(x);
Stop
no
x < y
yes
x=x−y;
y=y−x;

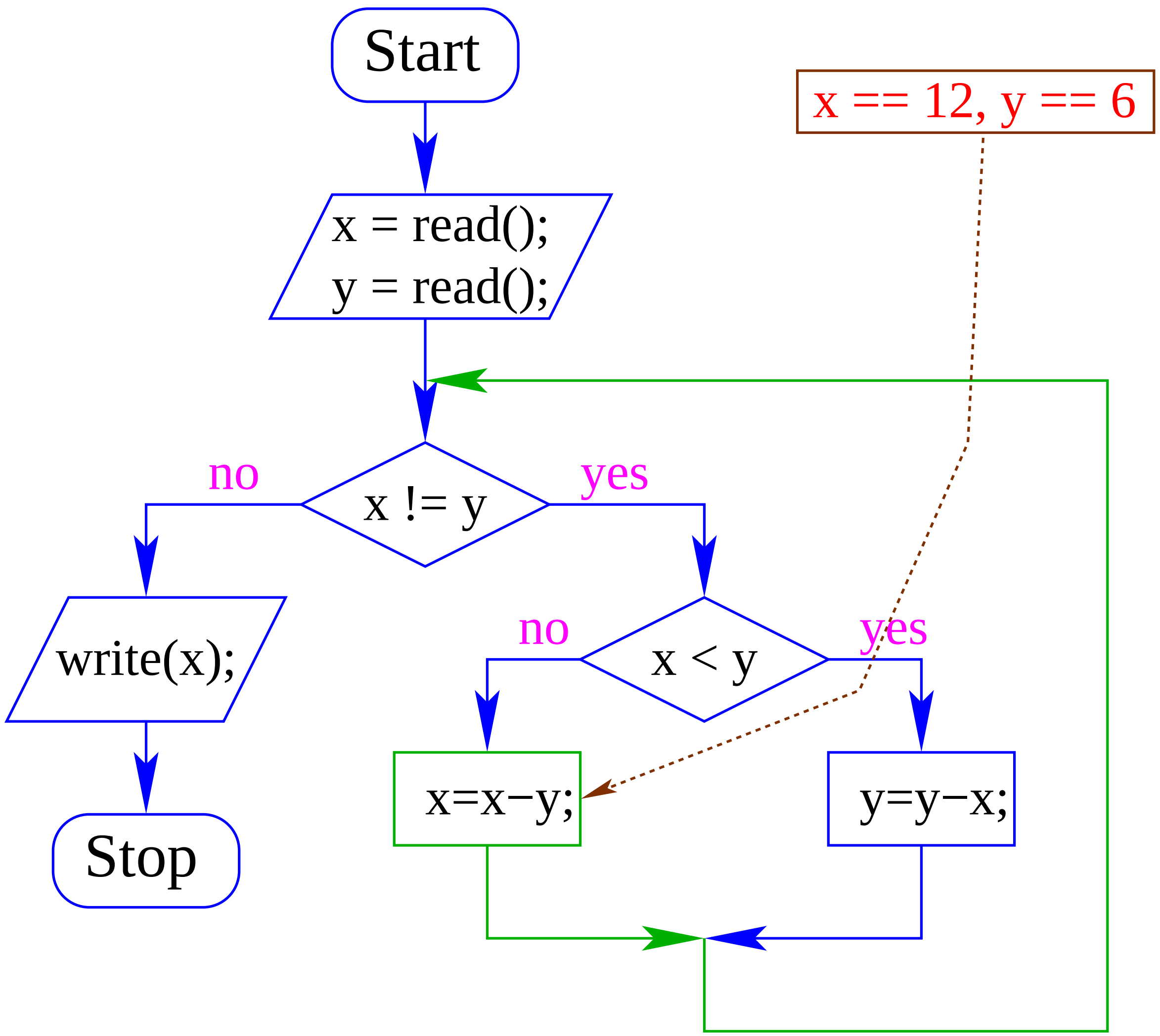

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

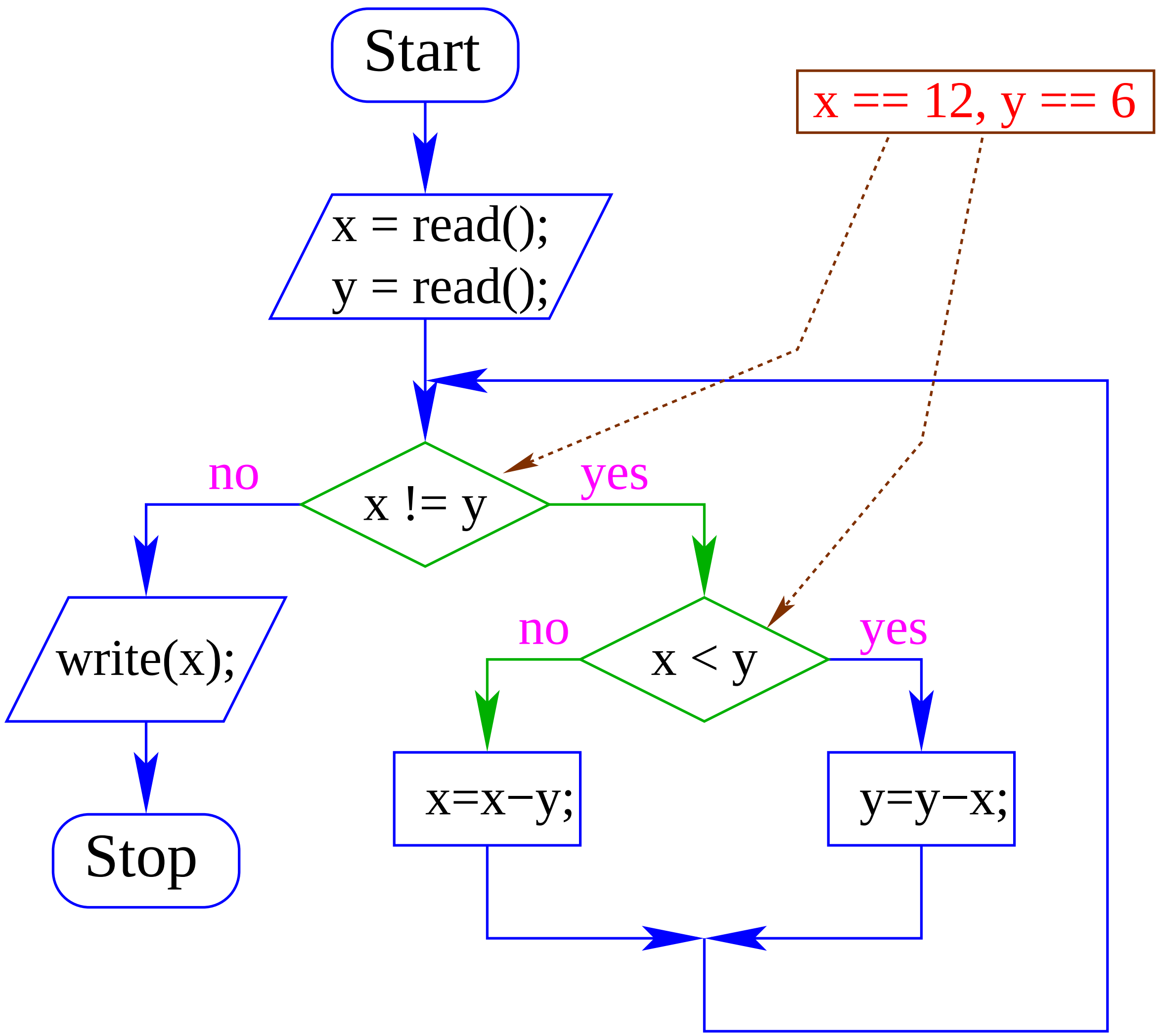

Start
x == 18, y == 24
x = read();
y = read();
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

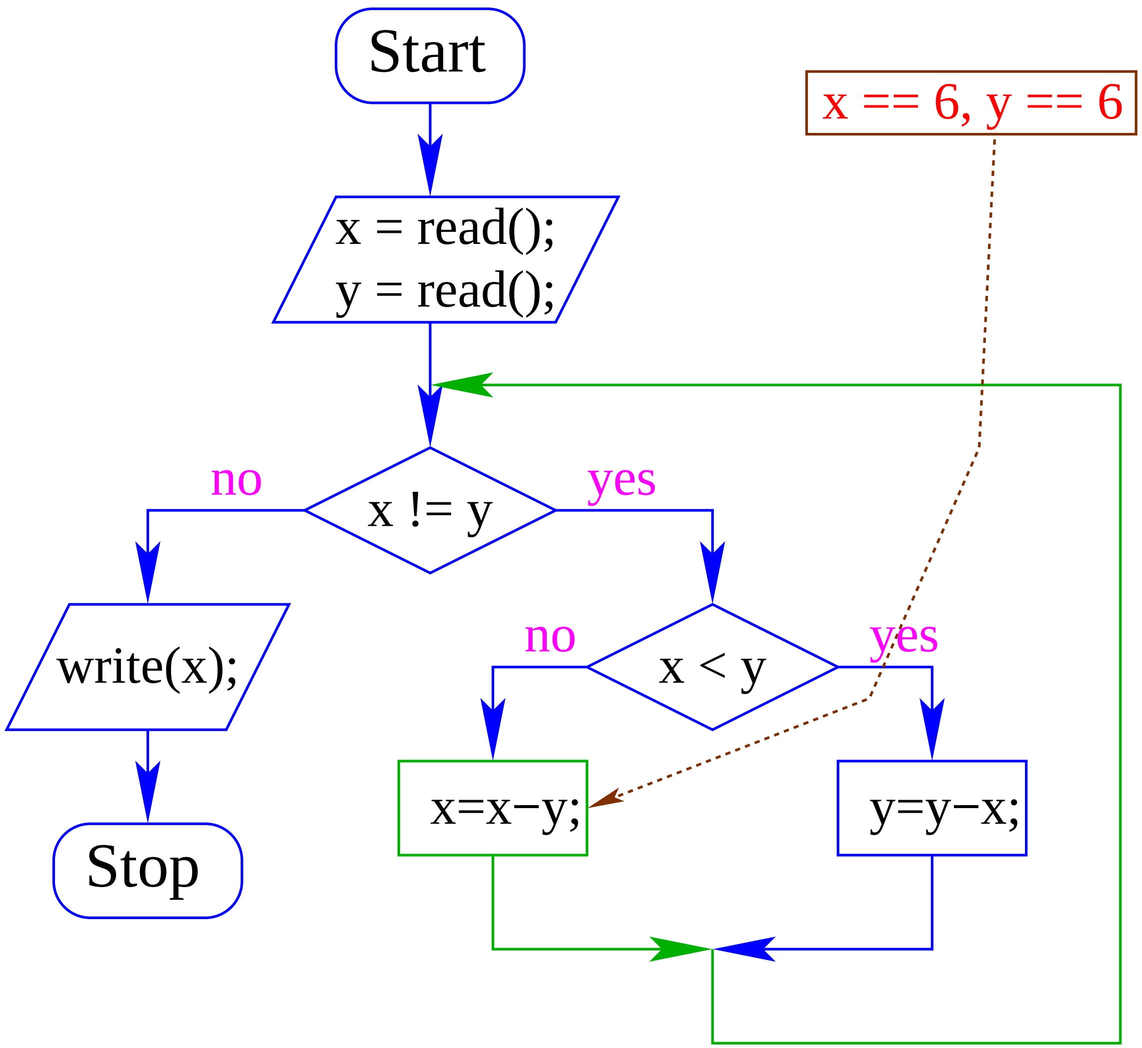

Start
x = read();
y = read();
x == 18, y == 24
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

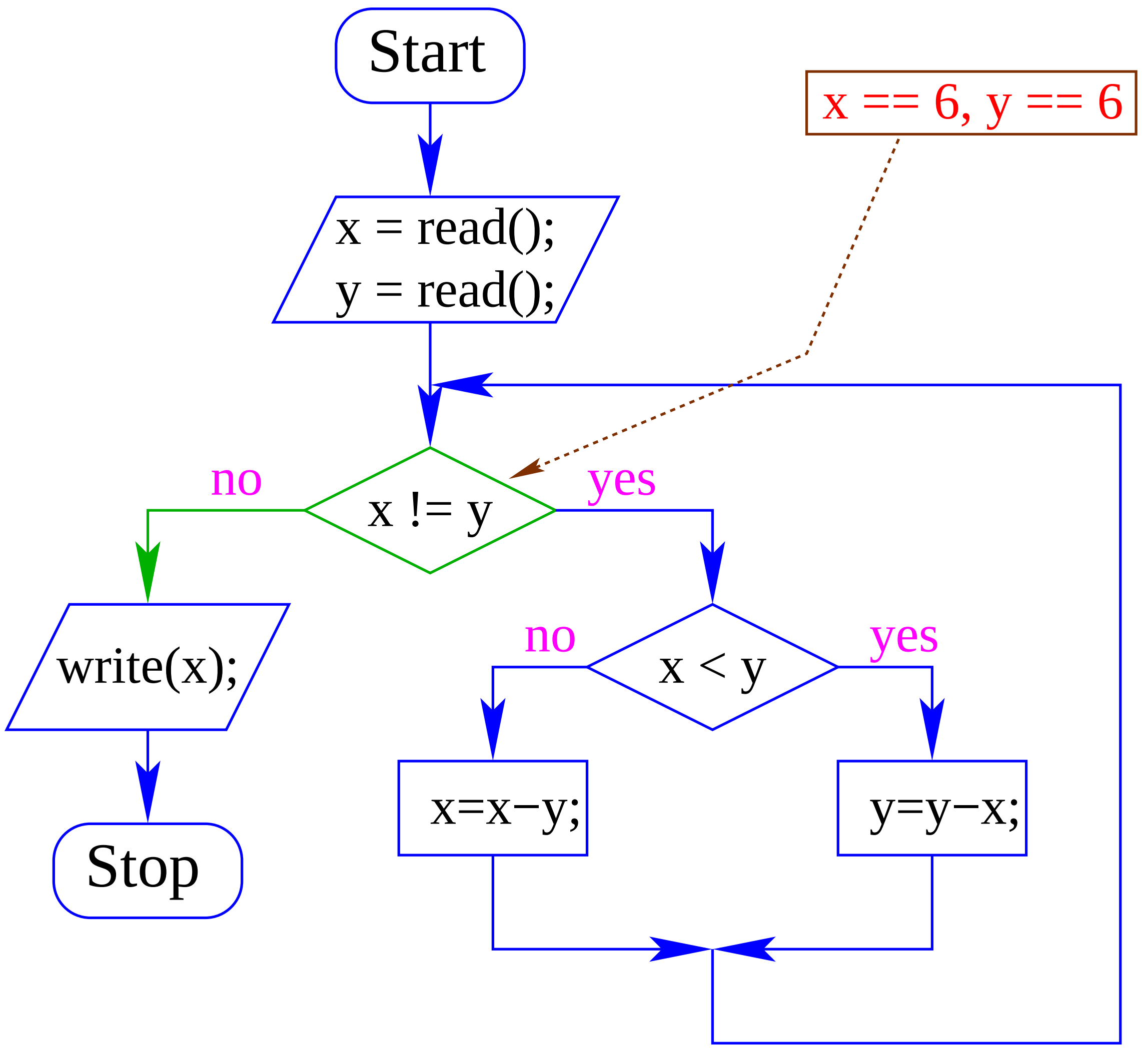

Start
x == 18, y == 6
x = read();
y = read();
no
yes
x != y
write(x);
no
yes
x < y
x=x−y;
y=y−x;
Stop

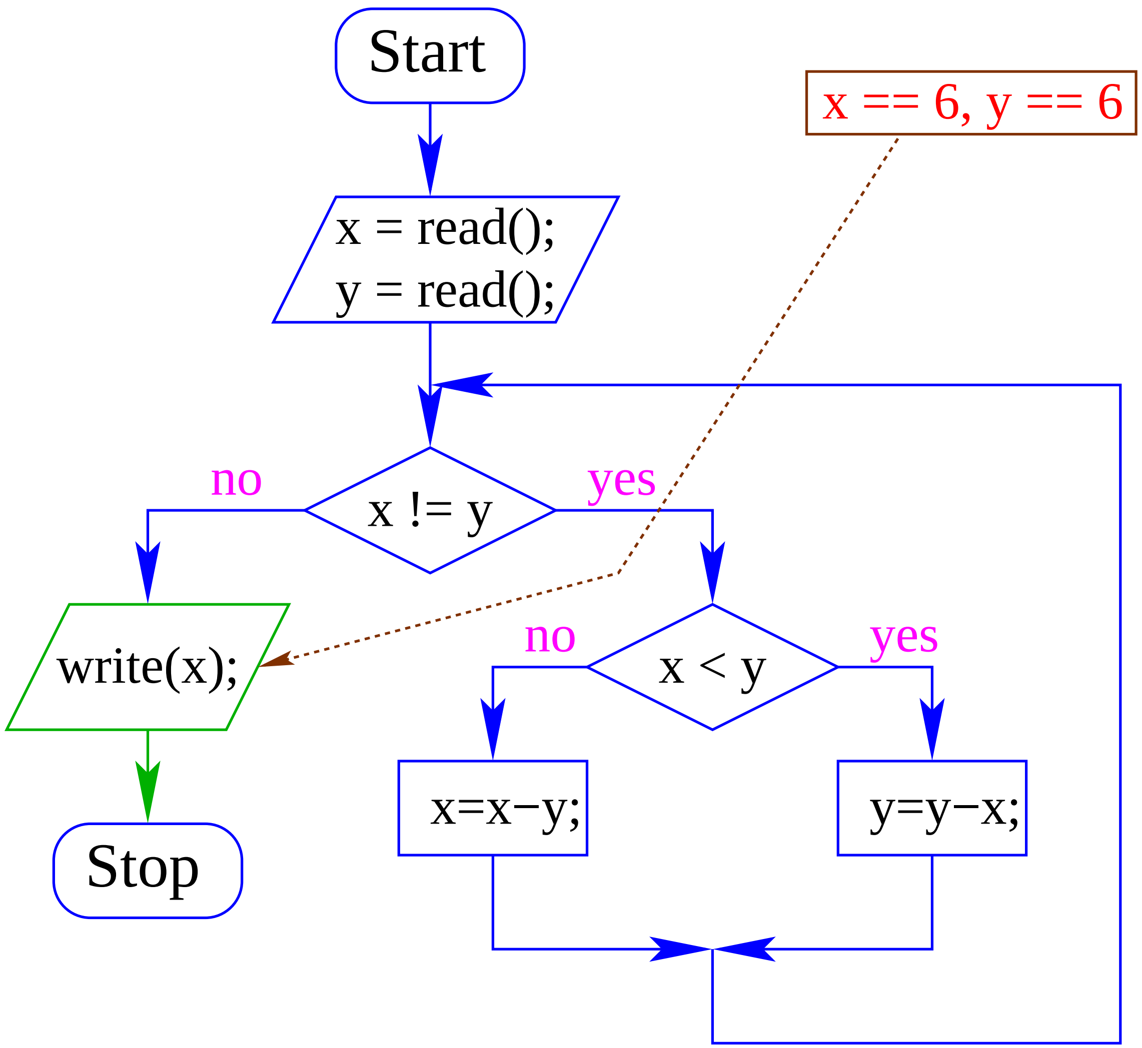

Start
x = read();
y = read();
x == 18, y == 6
no
x != y
yes
write(x);
no
x < y
yes
Stop
x=x−y;
y=y−x;

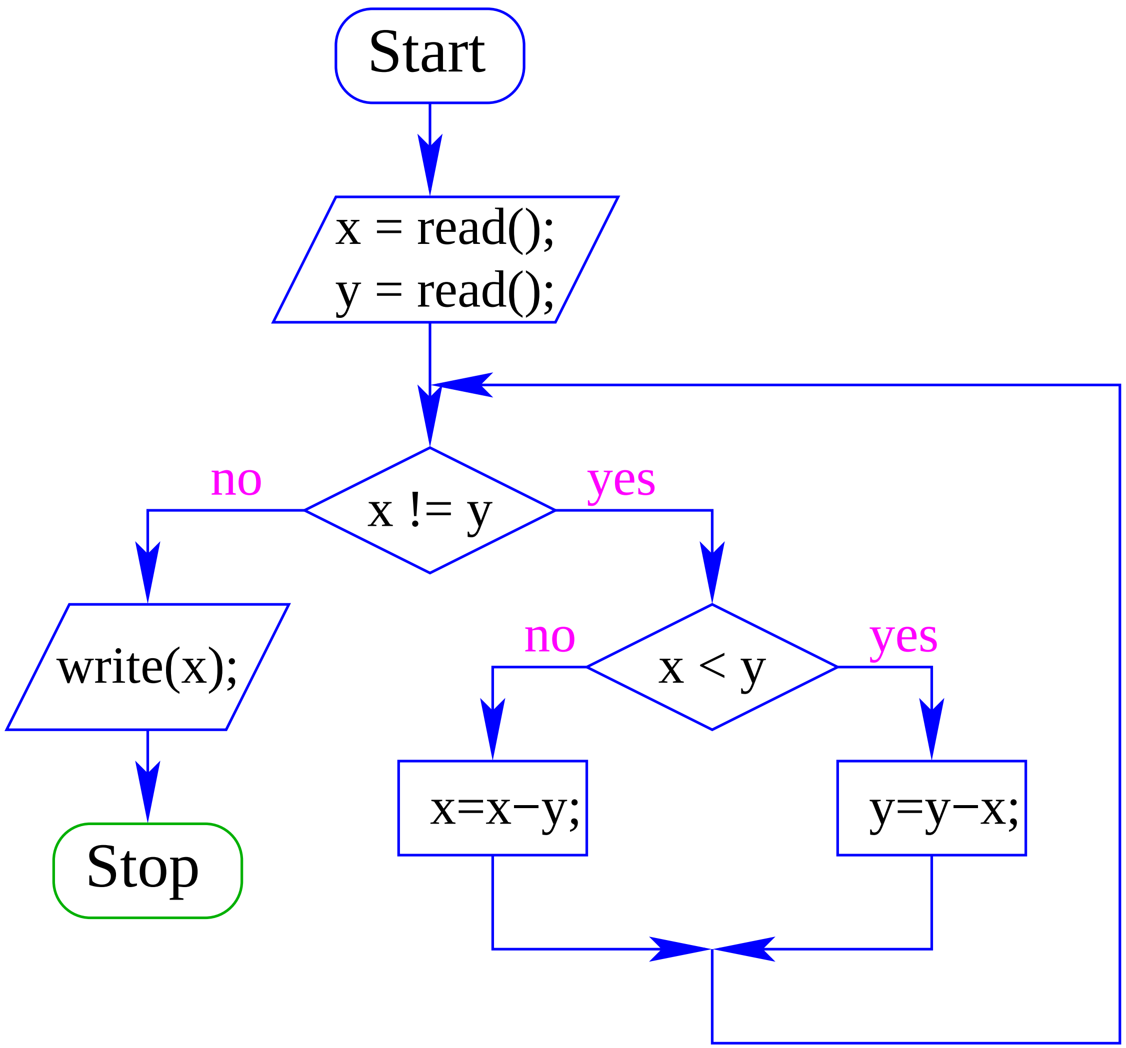

Start
x == 12, y == 6
x = read();
y = read();
no
yes
x != y
no
yes
write(x);
x < y
x=x−y;
y=y−x;
Stop


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