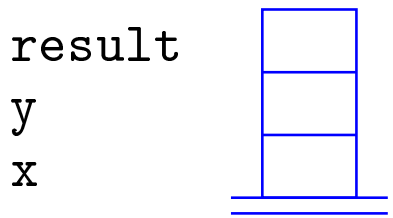
9.1 Übersetzung von Deklarationen

Betrachte Deklaration

```
int x, y, result;
```

Idee:

Wir reservieren der Reihe nach für die Variablen Zellen im Speicher:



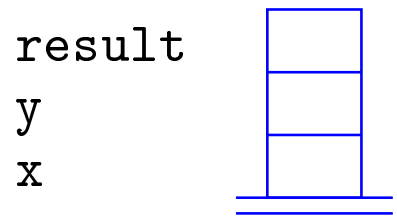
9.1 Übersetzung von Deklarationen

Betrachte Deklaration

```
int x, y, result;
```

Idee:

Wir reservieren der Reihe nach für die Variablen Zellen im Speicher:



$\Longrightarrow$

Übersetzung von `int` $x_0, \ldots, x_{n-1}$; $=$ ALLOC n

9.2 Übersetzung von Ausdrücken

Idee:

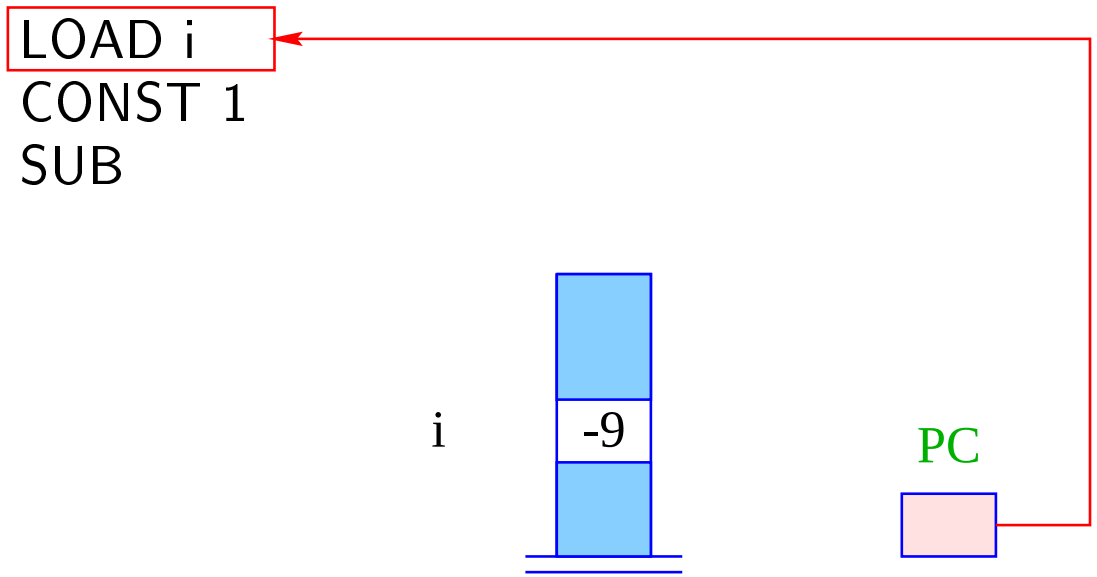
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

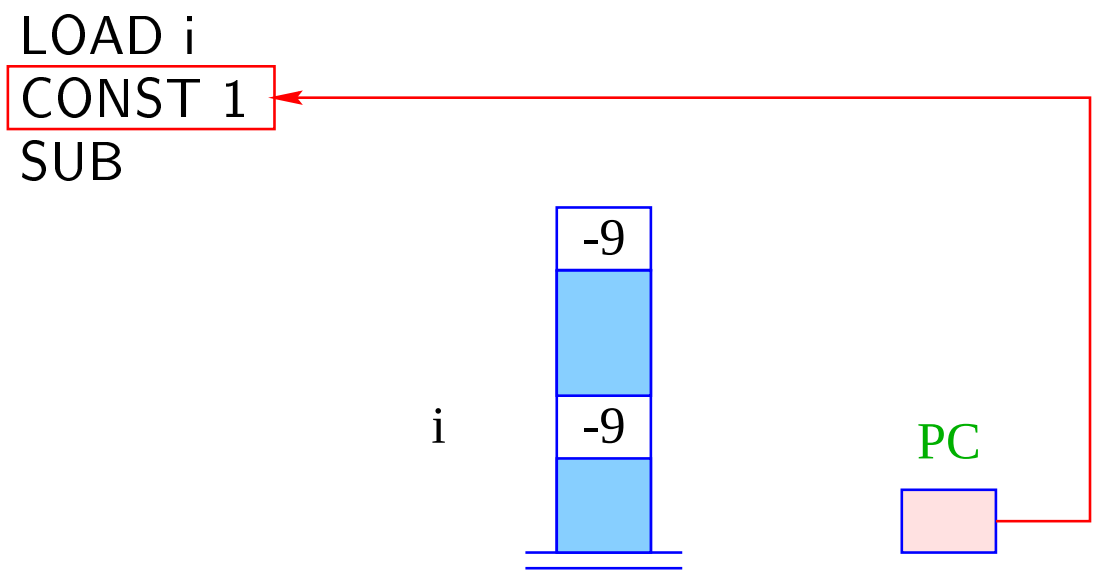
9.2 Übersetzung von Ausdrücken

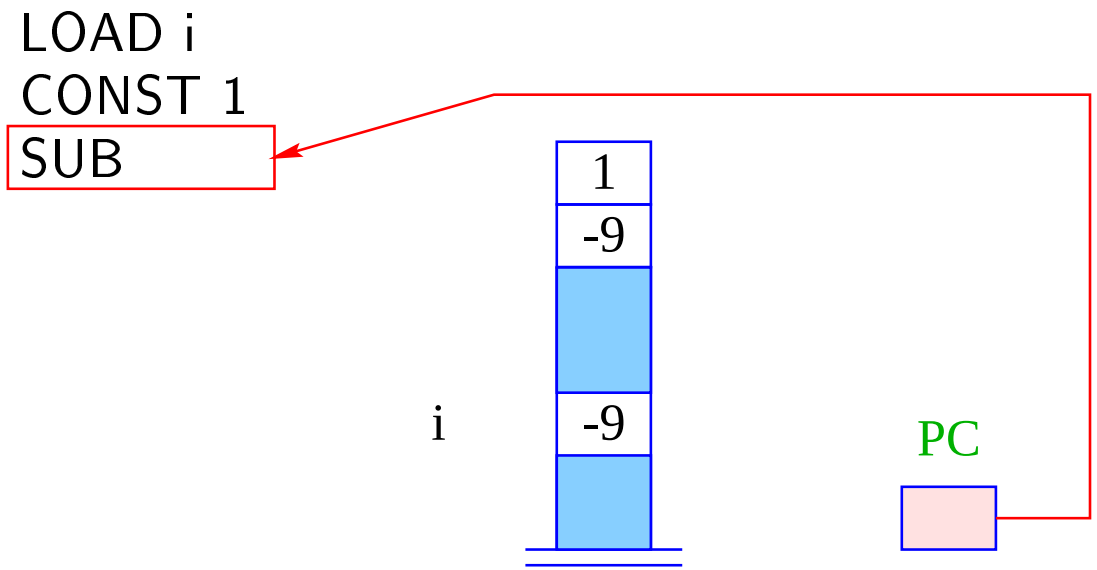
Idee:

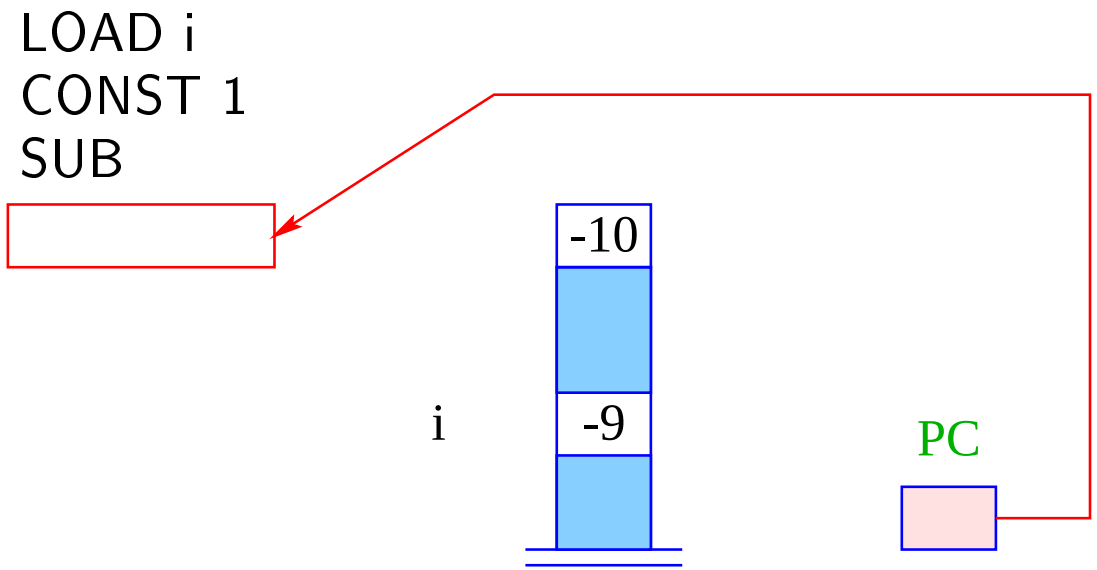
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

Übersetzung von	`x`	=	LOAD i	— `x` die i-te Variable
Übersetzung von	`17`	=	CONST 17	
Übersetzung von	`x - 1`	=	LOAD i CONST 1 SUB	


LOAD i
CONST 1
SUB
i
-9
PC


LOAD i
CONST 1
SUB
-9
i
-9
PC


LOAD i
CONST 1
SUB
1
-9
i
-9
PC


LOAD i
CONST 1
SUB
-10
i
-9
PC

Allgemein:

Übersetzung von	- expr	=	Übersetzung von expr NEG
Übersetzung von	expr_1 + expr_2	=	Übersetzung von expr_1 Übersetzung von expr_2 ADD

... analog für die anderen Operatoren ...

Beispiel:

Sei expr der Ausdruck: `(x + 7) * (y - 14)`

wobei `x` und `y` die 0. bzw. 1. Variable sind.

Dann liefert die Übersetzung:

```
LOAD 0
CONST 7
ADD
LOAD 1
CONST 14
SUB
MUL
```

9.3 Übersetzung von Zuweisungen

Idee:

- Übersetze den Ausdruck auf der rechten Seite.
  Das liefert eine Befehlsfolge, die den Wert der rechten Seite oben auf dem Stack ablegt.
- Speichere nun diesen Wert in der Zelle für die linke Seite ab!

9.4 Übersetzung von Zuweisungen

Idee:

- Übersetze den Ausdruck auf der rechten Seite.
  Das liefert eine Befehlsfolge, die den Wert der rechten Seite oben auf dem Stack ablegt.
- Speichere nun diesen Wert in der Zelle für die linke Seite ab!

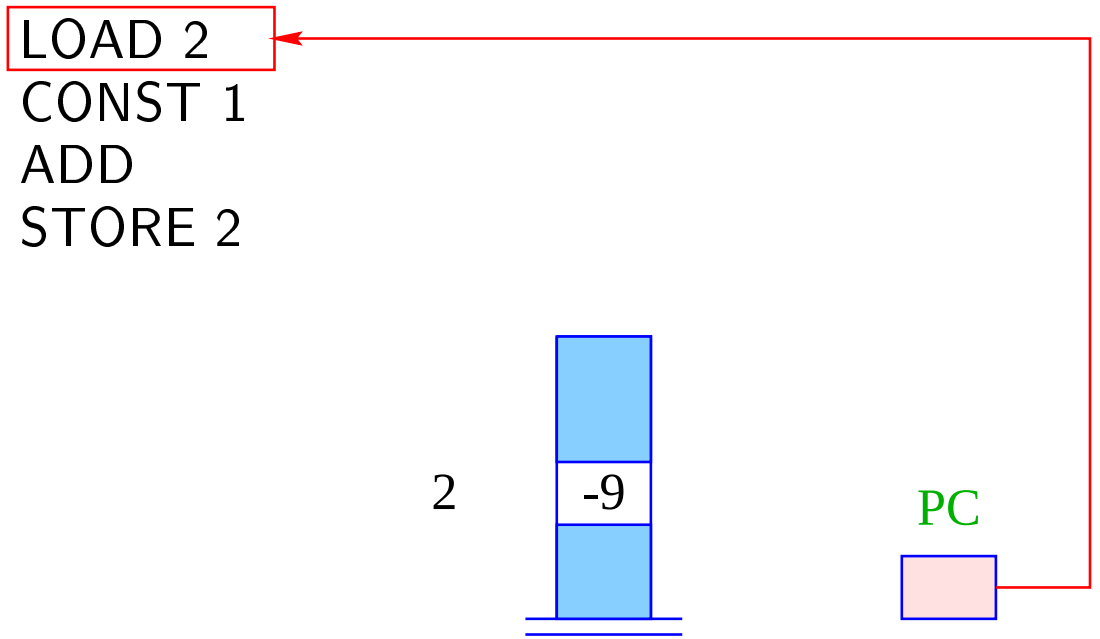
Sei `x` die Variable Nr. i. Dann ist

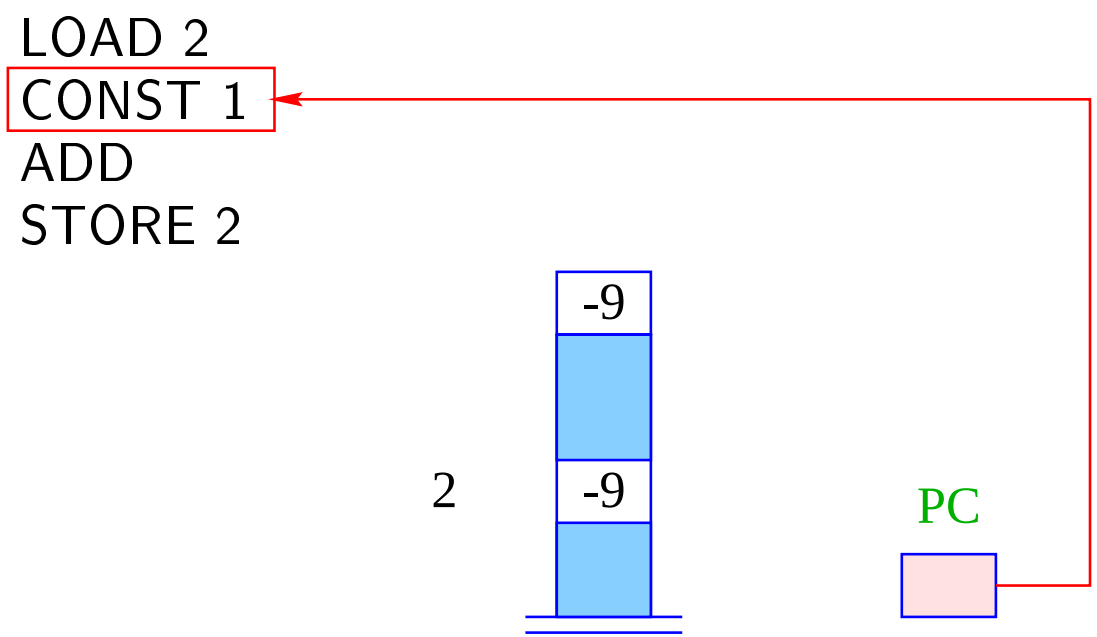
Übersetzung von	`x = expr;`	=	Übersetzung von	expr
			STORE i	

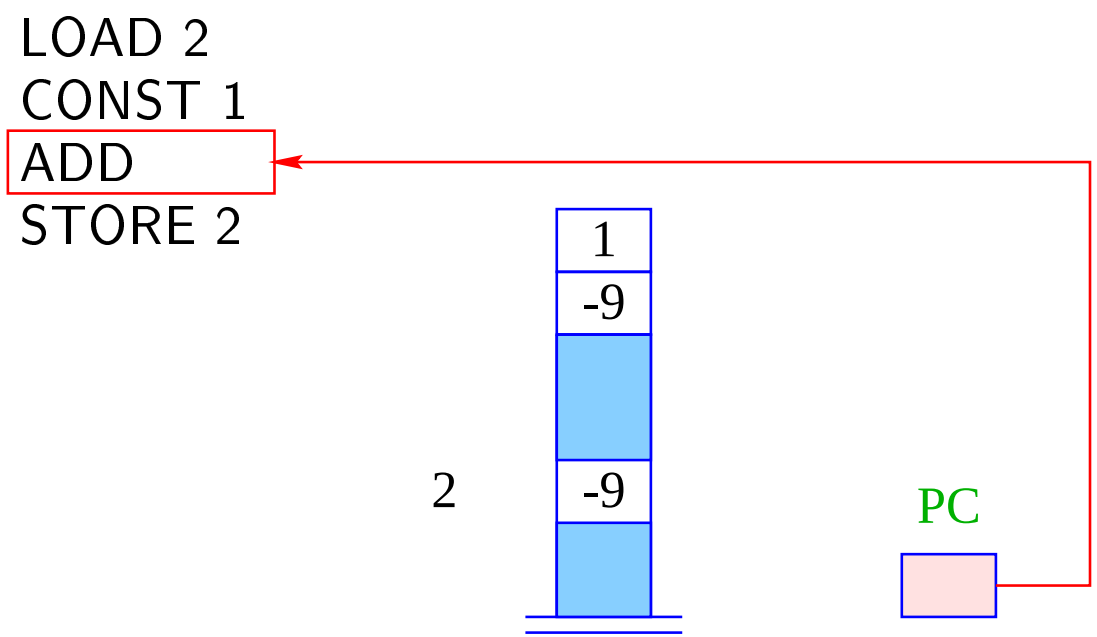
Beispiel:

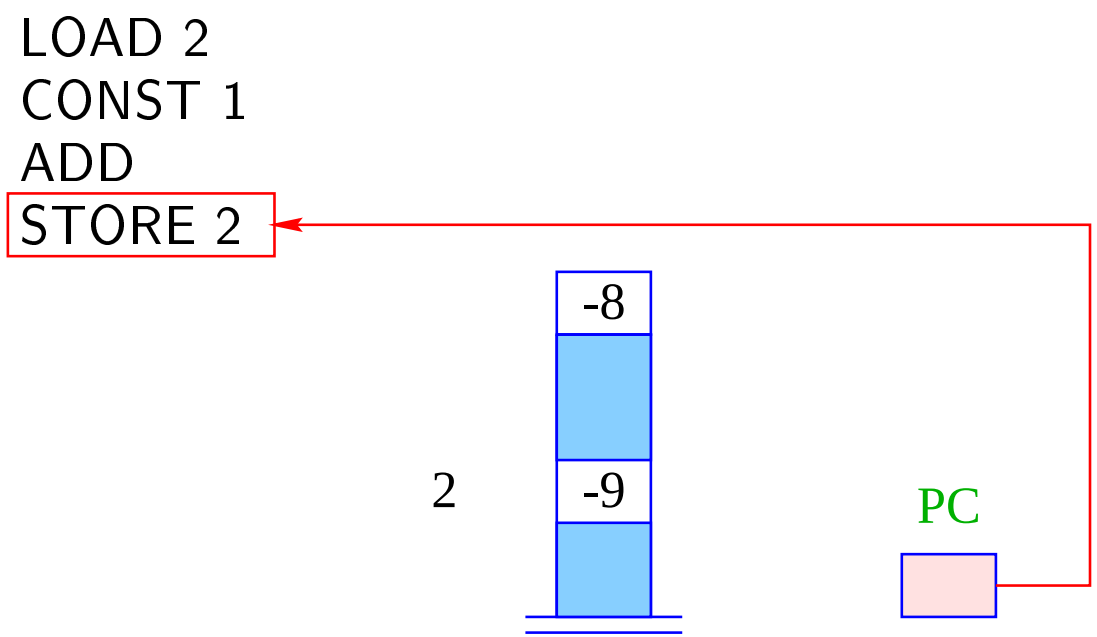
Für `x = x + 1;` (`x` die 2. Variable) liefert das:

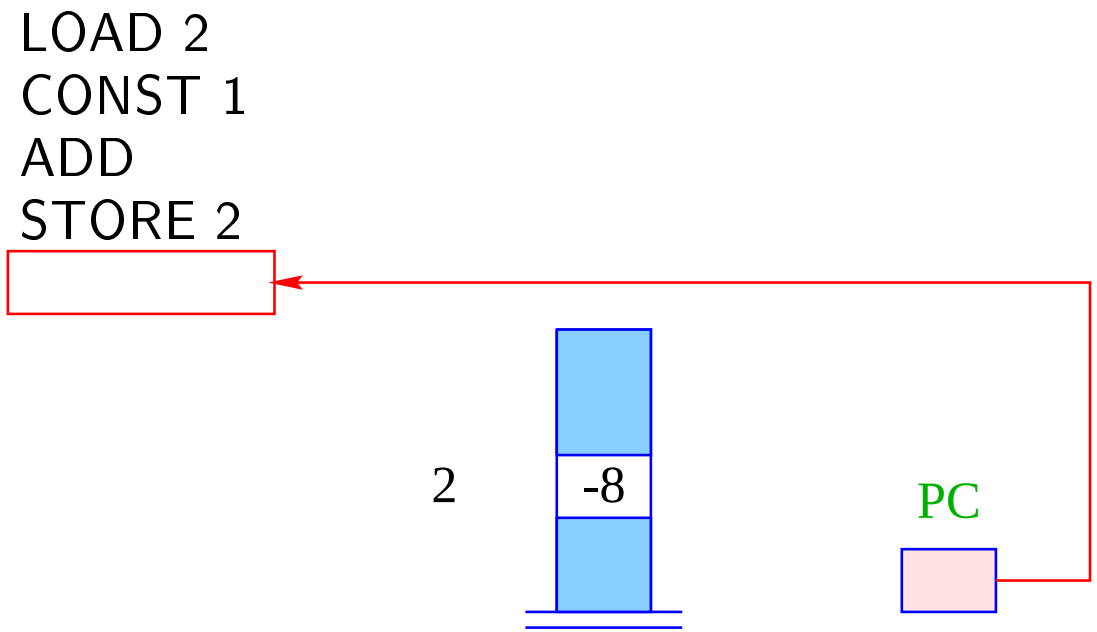
```
LOAD 2
CONST 1
ADD
STORE 2
```


LOAD 2
CONST 1
ADD
STORE 2
2
-9
PC


LOAD 2
CONST 1
ADD
STORE 2
-9
2
-9
PC


LOAD 2
CONST 1
ADD
STORE 2
1
-9
2
-9
PC


LOAD 2
CONST 1
ADD
STORE 2
-8
2
-9
PC


LOAD 2
CONST 1
ADD
STORE 2
2
-8
PC

Bei der Übersetzung von `x = read();` und `write(expr);`
gehen wir analog vor :-)

Sei `x` die Variable Nr. i. Dann ist

Übersetzung von	`x = read();`	=	READ STORE i
Übersetzung von	`write( expr);`	=	Übersetzung von expr WRITE

9.5 Übersetzung von `if`-Statements

Bezeichne stmt das `if`-Statement

`if (` cond `)` stmt_1 `else` stmt_2

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für cond, stmt_1 und stmt_2.
- Diese ordnen wir hinter einander an.
- Dann fügen wir Sprünge so ein, dass in Abhängigkeit des Ergebnisses der Auswertung der Bedingung jeweils entweder nur stmt_1 oder nur stmt_2 ausgeführt wird.

Folglich (mit A, B zwei neuen Marken):

Übersetzung von stmt =

Übersetzung von cond
FJUMP A
Übersetzung von stmt_1
JUMP B
A: Übersetzung von stmt_2
B: ...

- Marke A markiert den Beginn des else-Teils.
- Marke B markiert den ersten Befehl hinter dem `if`-Statement.
- Falls die Bedingung sich zu `false` evaluiert, wird der `then`-Teil übersprungen (mithilfe von FJUMP A).
- Nach Abarbeitung des `then`-Teils muss in jedem Fall hinter dem gesamten `if`-Statement fortgefahren werden. Dazu dient JUMP B.

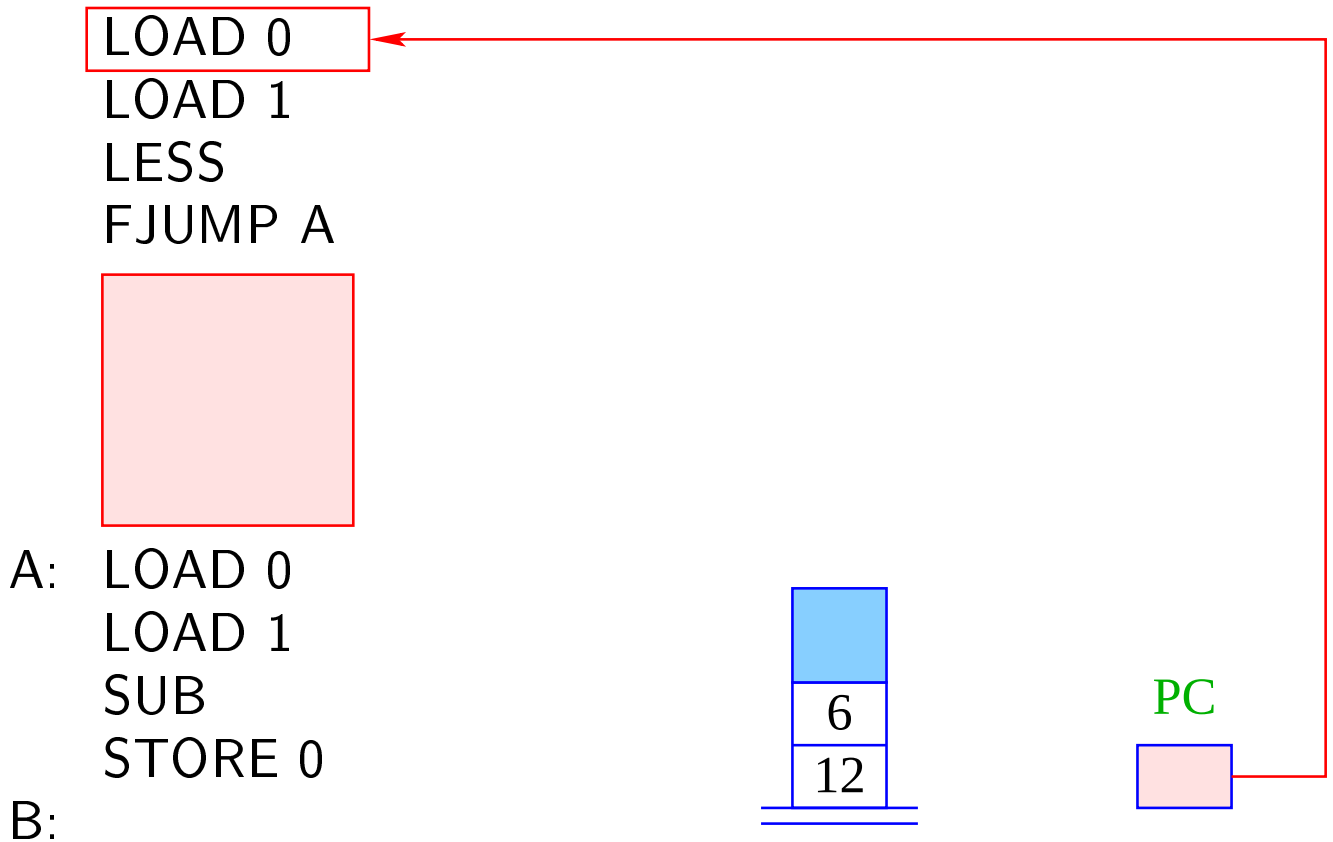
Beispiel:

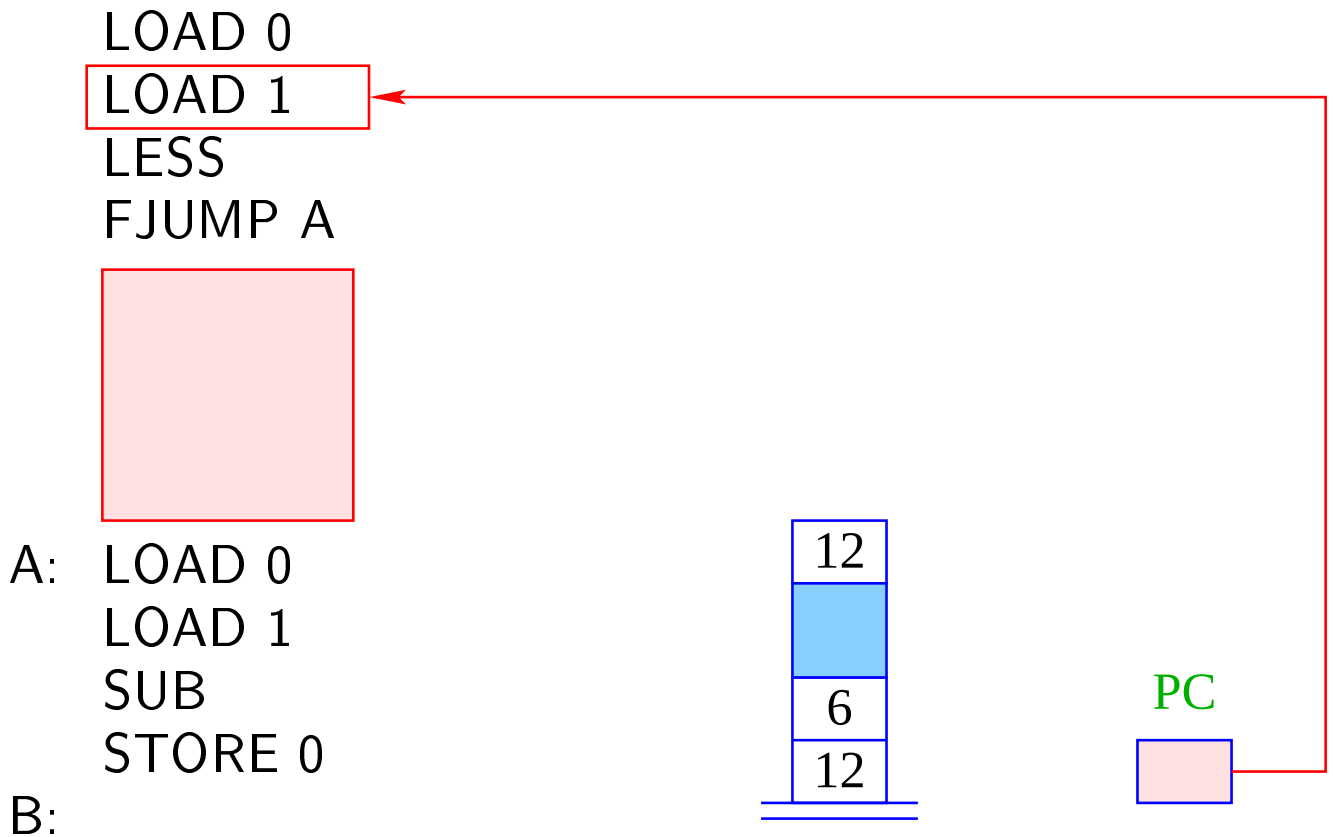
Für das Statement:

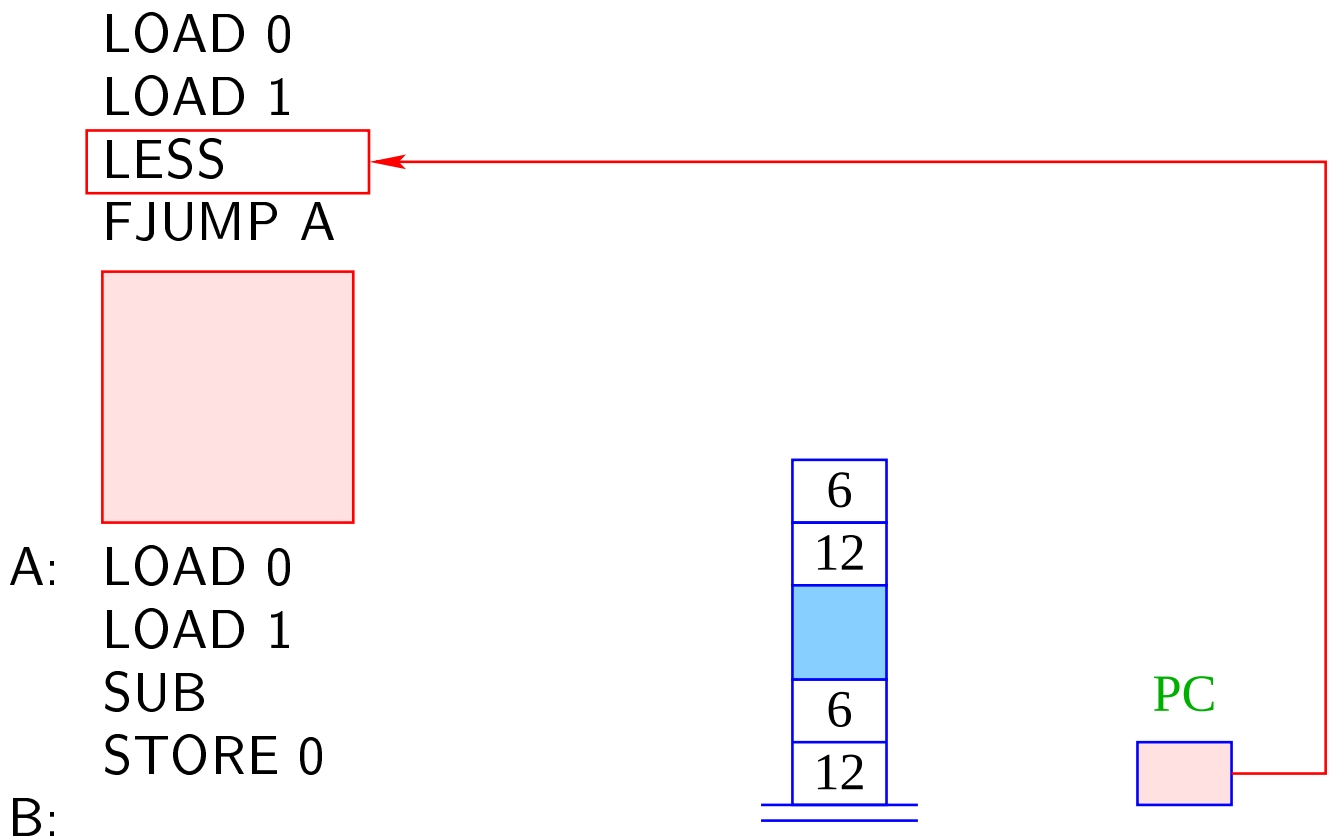
```
if (x < y) y = y - x;
else x = x - y;
```

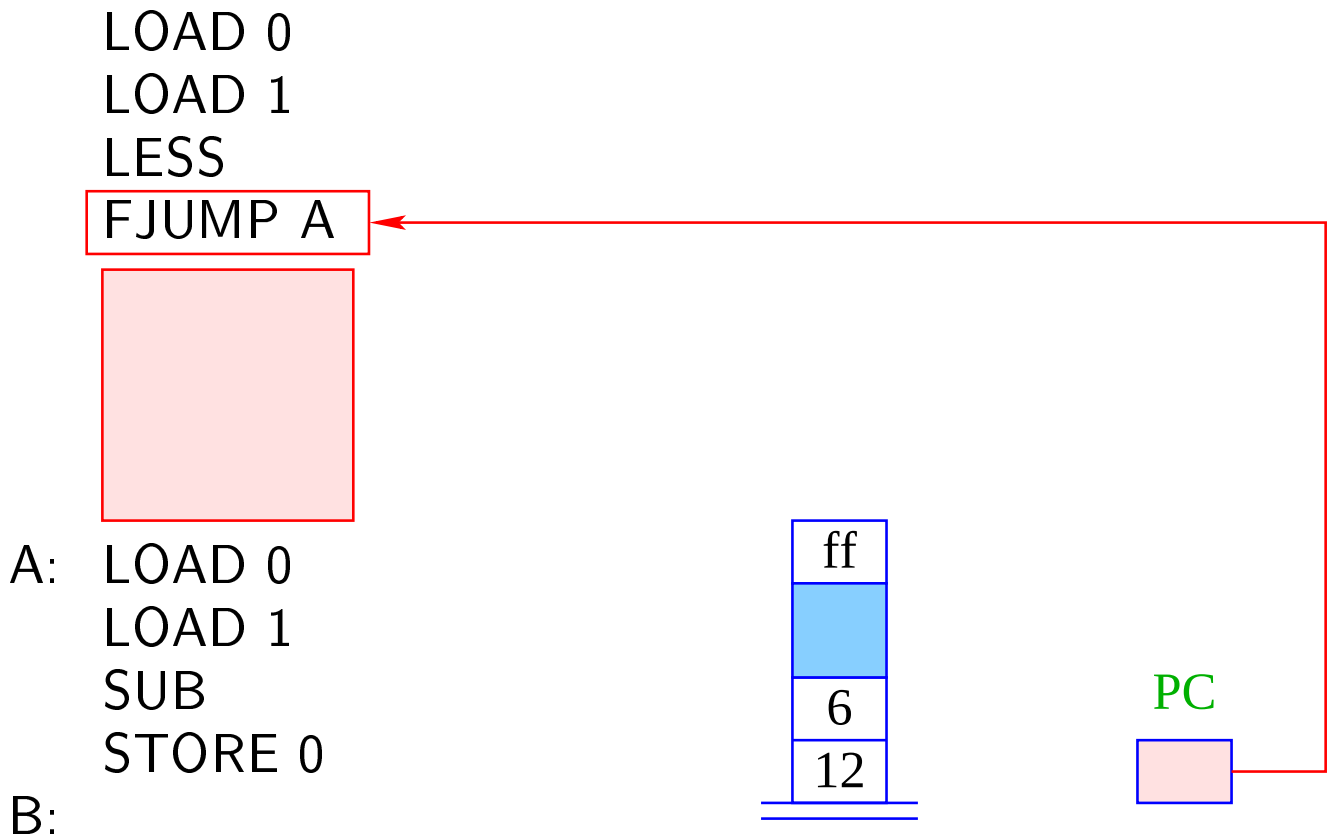
(`x` und `y` die 0. bzw. 1. Variable) ergibt das:

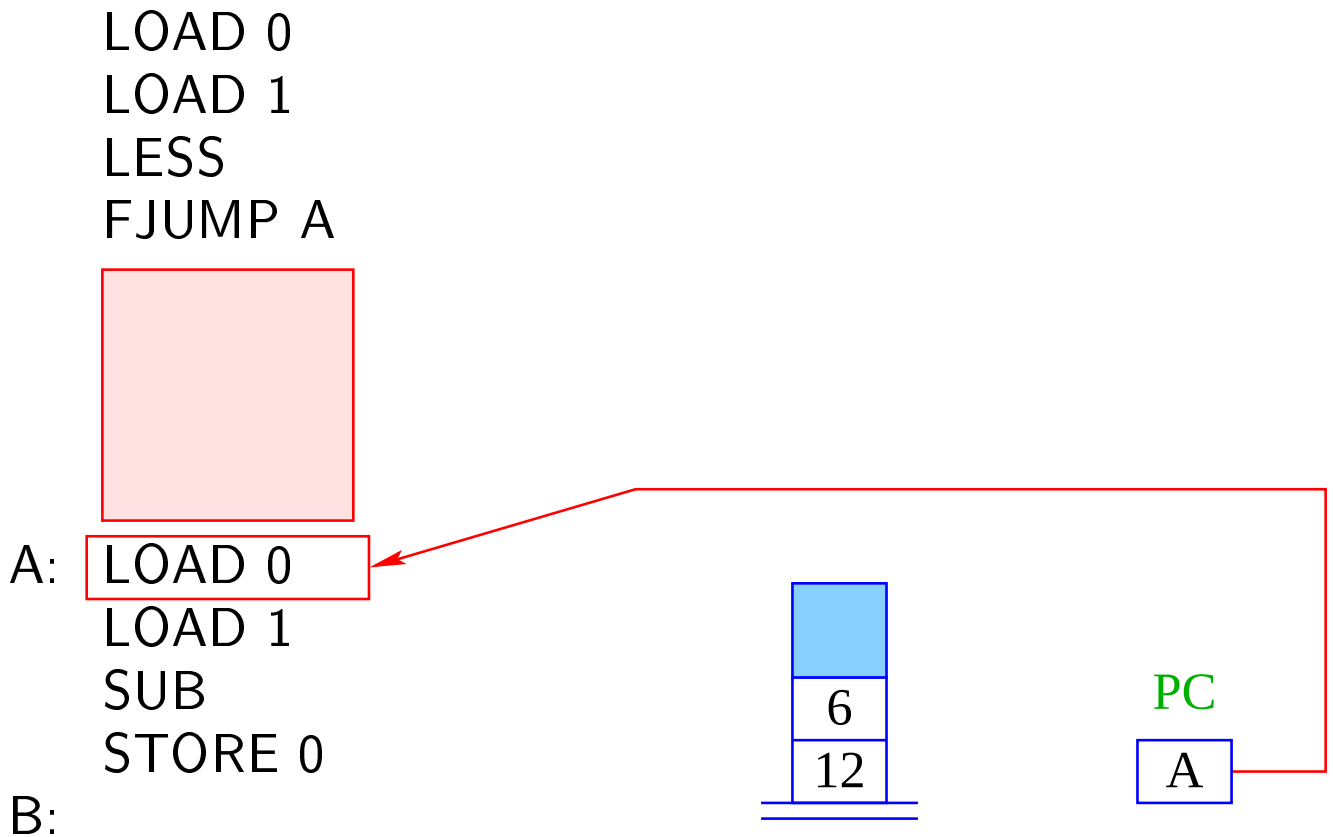
```
    LOAD 0          LOAD 1          A:  LOAD 0
    LOAD 1          LOAD 0              LOAD 1
    LESS            SUB                 SUB
    FJUMP A         STORE 1             STORE 0
                    JUMP B          B:  ...
```

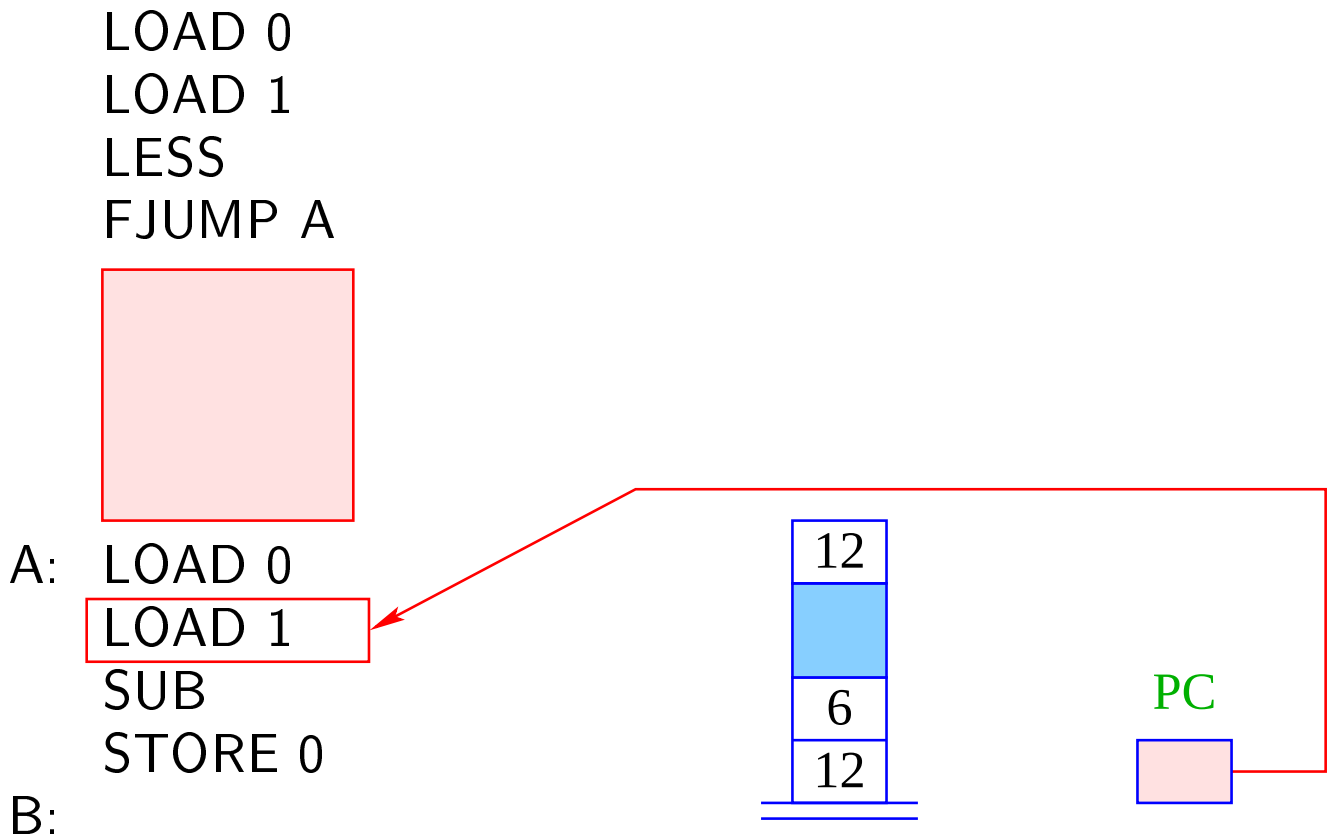

LOAD 0
LOAD 1
LESS
FJUMP A
A: LOAD 0
LOAD 1
SUB
STORE 0
B:
6
12
PC

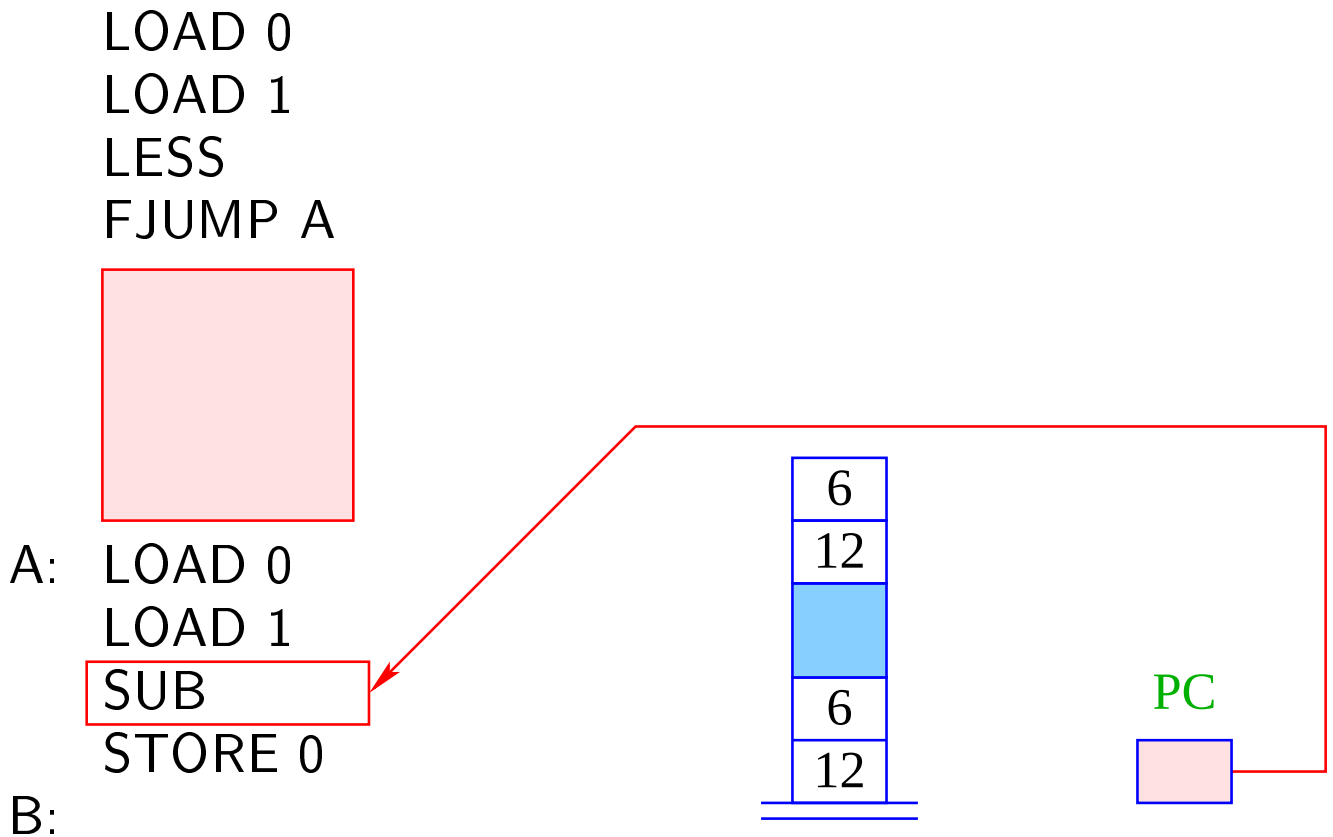

LOAD 0
LOAD 1
LESS
FJUMP A
A: LOAD 0
LOAD 1
SUB
STORE 0
B:
12
6
12
PC

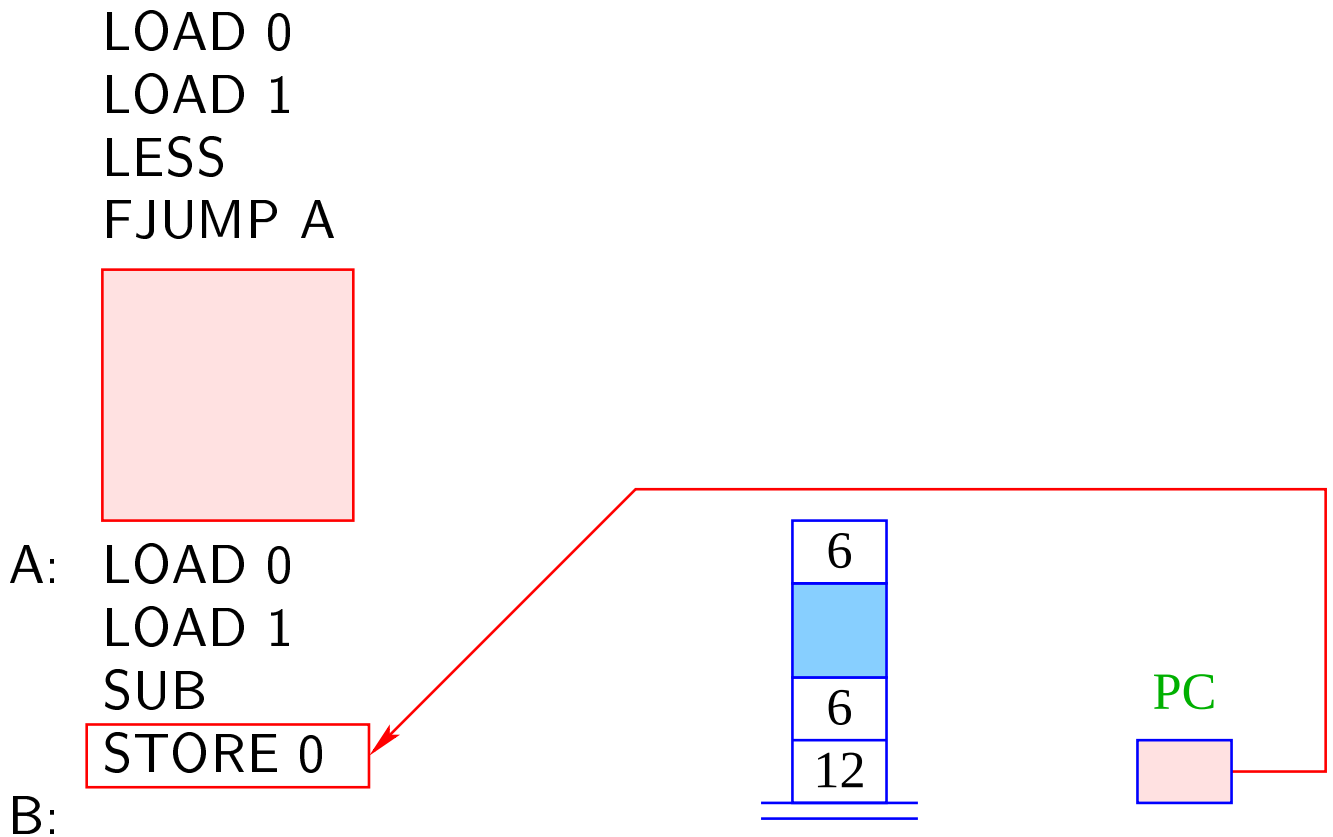

LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
12
6
12
PC

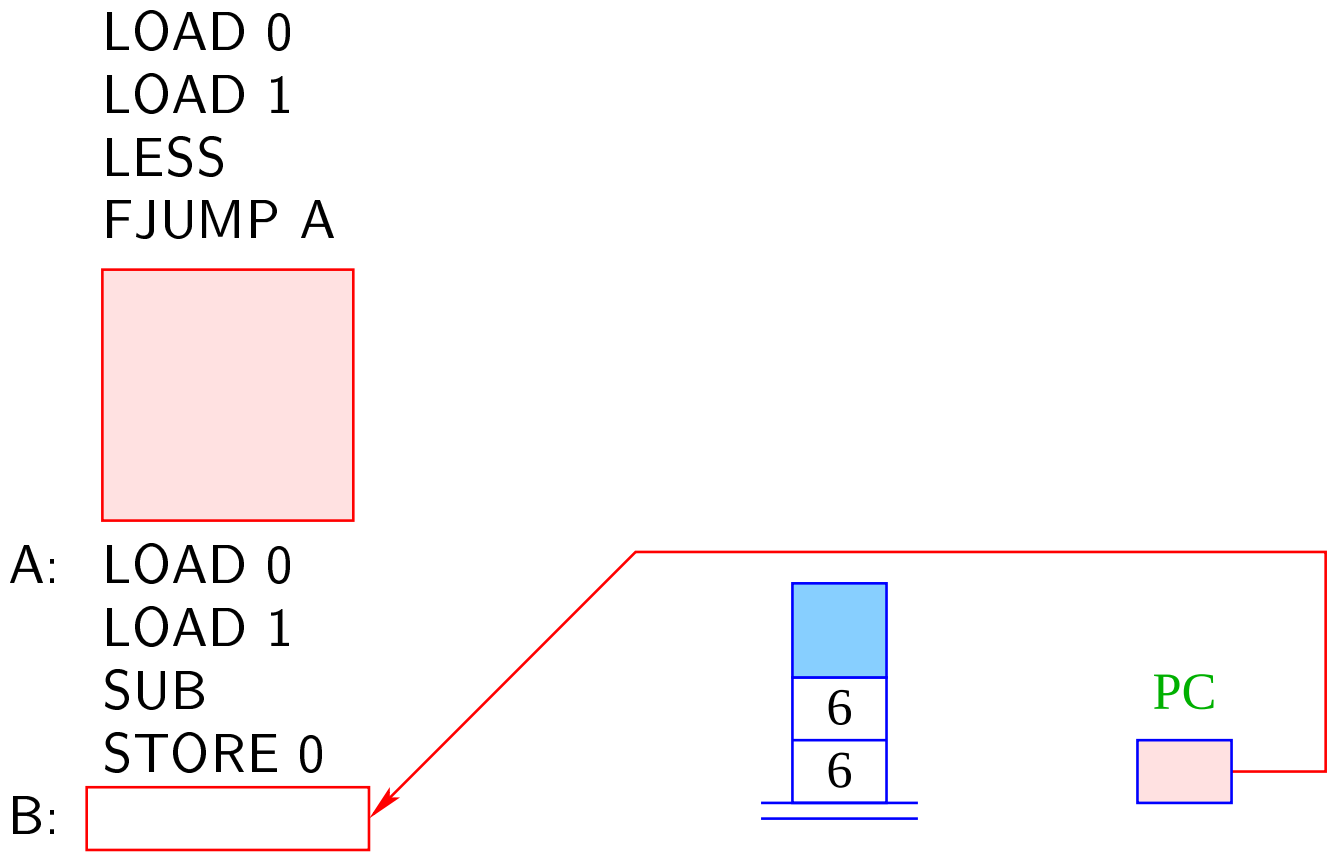

LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
ff
6
12
PC


LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
12
PC
A


LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
12
6
12
PC


LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
12
6
12
PC


LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
6
12
PC


LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
6
PC

9.6 Übersetzung von `while`-Statements

Bezeichne stmt das `while`-Statement

`while ( cond )` stmt_1

Idee:

- Wir erzeugen erst einmal Befehlsfolgen für cond und stmt_1.
- Diese ordnen wir hinter einander an.
- Dann fügen wir Sprünge so ein, dass in Abhängigkeit des Ergebnisses der Auswertung der Bedingung entweder hinter das `while`-Statement gesprungen wird oder stmt_1 ausgeführt wird.
- Nach Ausführung von stmt_1 müssen wir allerdings wieder an den Anfang des Codes zurückspringen :-)

Folglich (mit A, B zwei neuen Marken):

Übersetzung von stmt = A: Übersetzung von cond
FJUMP B
Übersetzung von stmt_1
JUMP A
B: ...

- Marke A markiert den Beginn des while-Statements.
- Marke B markiert den ersten Befehl hinter dem `while`-Statement.
- Falls die Bedingung sich zu `false` evaluiert, wird die Schleife verlassen (mithilfe von FJUMP B).
- Nach Abarbeitung des Rumpfs muss das `while`-Statement erneut ausgeführt werden. Dazu dient JUMP A.

Beispiel:

Für das Statement:

```
while (1 < x) x = x - 1;
```

(`x` die 0. Variable) ergibt das:

```
A:  CONST 1          LOAD 0
    LOAD 0           CONST 1
    LESS             SUB
    FJUMP B          STORE 0
                     JUMP A
                B:   ...
```

9.7 Übersetzung von Statement-Folgen

Idee:

- Wir erzeugen zuerst Befehlsfolgen für die einzelnen Statements in der Folge.
- Dann konkatenieren wir diese.

9.8 Übersetzung von Statement-Folgen

Idee:

- Wir erzeugen zuerst Befehlsfolgen für die einzelnen Statements in der Folge.
- Dann konkatenieren wir diese.

Folglich:

Übersetzung von $\text{stmt}_1 \ldots \text{stmt}_k$ = Übersetzung von stmt_1
...
Übersetzung von stmt_k

Beispiel:

Für die Statement-Folge

```
y = y * x;
x = x - 1;
```

(`x` und `y` die 0. bzw. 1. Variable) ergibt das:

LOAD 1	LOAD 0
LOAD 0	CONST 1
MUL	SUB
STORE 1	STORE 0

9.9 Übersetzung ganzer Programme

Nehmen wir an, das Programm prog bestehe aus einer Deklaration von n Variablen, gefolgt von der Statement-Folge ss.

Idee:

- Zuerst allokieren wir Platz für die deklarierten Variablen.
- Dann kommt der Code für ss.
- Dann HALT.

9.9 Übersetzung ganzer Programme

Nehmen wir an, das Programm prog bestehe aus einer Deklaration von n Variablen, gefolgt von der Statement-Folge ss.

Idee:

- Zuerst allokieren wir Platz für die deklarierten Variablen.
- Dann kommt der Code für ss.
- Dann HALT.

Folglich:

Übersetzung von prog = ALLOC n
Übersetzung von ss
HALT

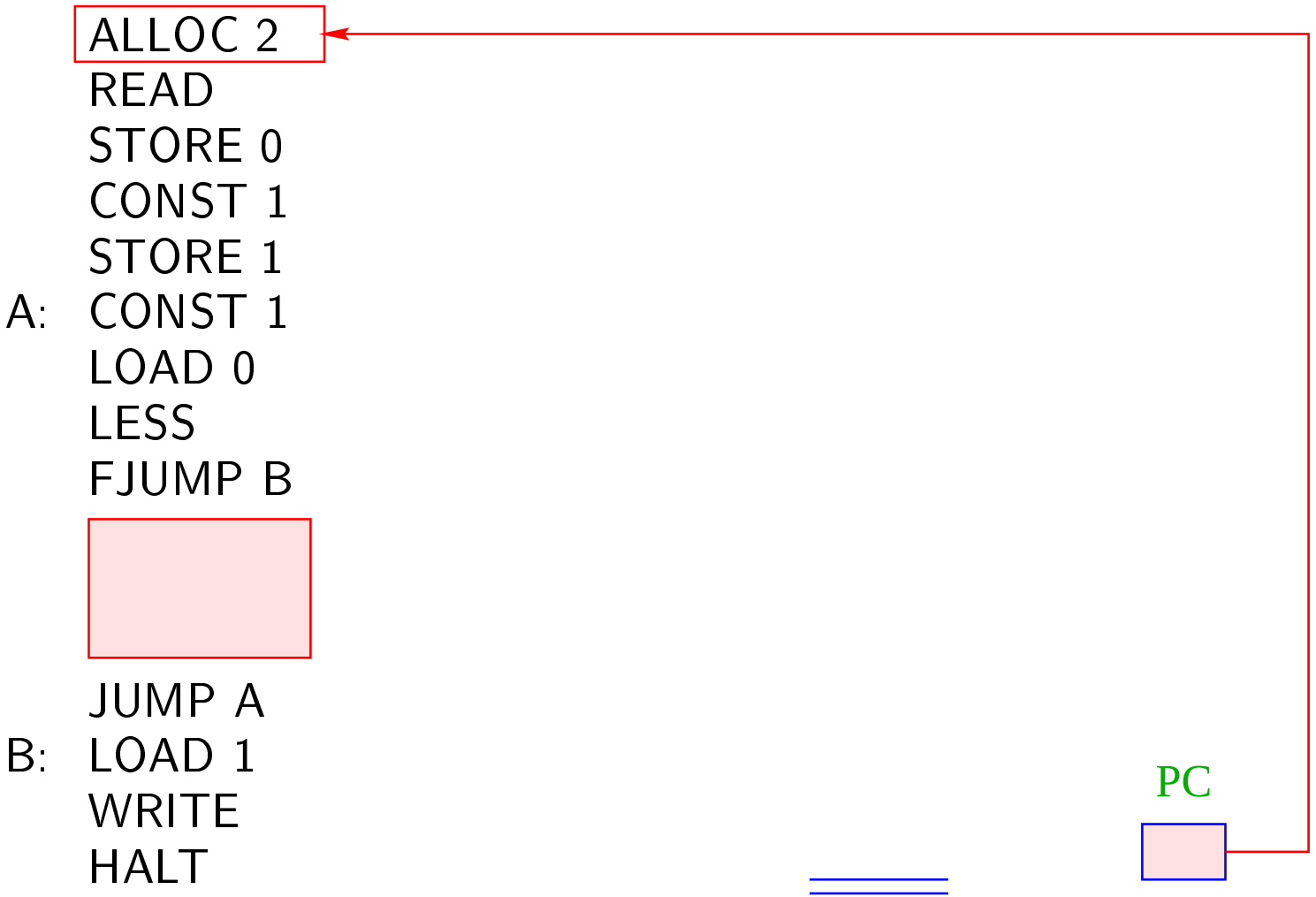
Beispiel:

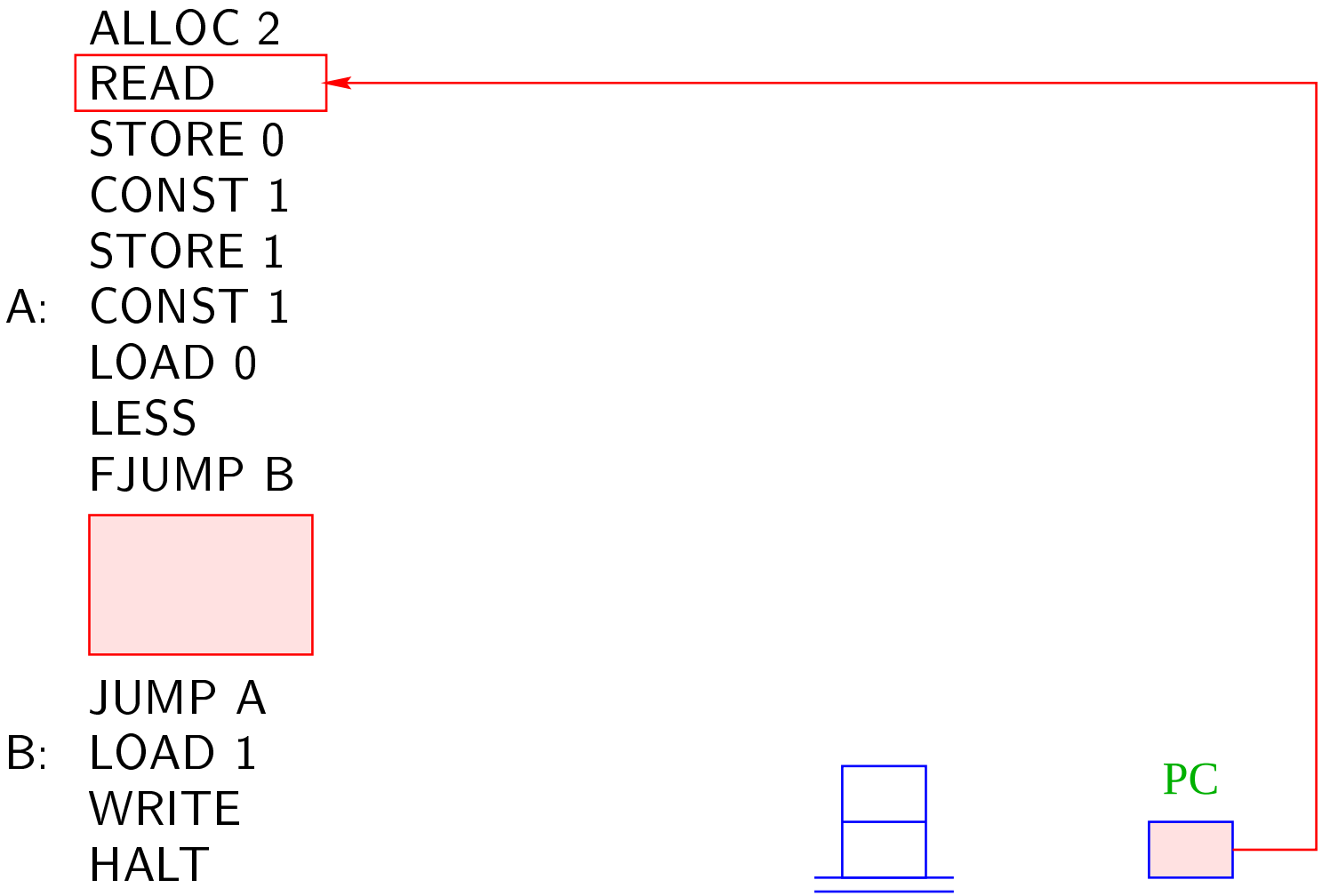
Für das Programm

```
int x, y;
x = read();
y = 1;
while (1 < x) {
    y = y * x;
    x = x - 1;
}
write(y);
```

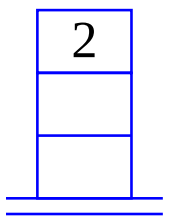
ergibt das (`x` und `y` die 0. bzw. 1. Variable) :

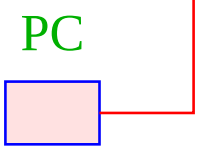
```
        ALLOC 2
        READ
        STORE 0
        CONST 1
        STORE 1
A:      CONST 1
        LOAD 0
        LESS
        FJUMP B
        LOAD 1
        LOAD 0
        MUL
        STORE 1
        LOAD 0
        CONST 1
        SUB
        STORE 0
        JUMP A
B:      LOAD 1
        WRITE
        HALT
```


ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B
JUMP A
B: LOAD 1
WRITE
HALT
PC


ALLOC 2
READ
STORE 0
CONST 1
STORE 1
A: CONST 1
LOAD 0
LESS
FJUMP B
JUMP A
B: LOAD 1
WRITE
HALT
PC

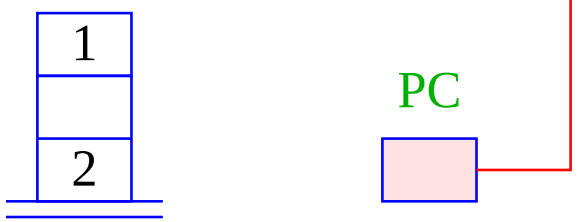
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```





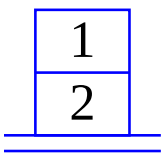
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

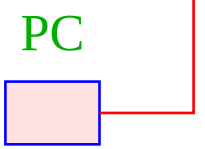
```
        ALLOC 2
        READ
        STORE 0
        CONST 1
        STORE 1
A:      CONST 1
        LOAD 0
        LESS
        FJUMP B
        
        JUMP A
B:      LOAD 1
        WRITE
        HALT
```



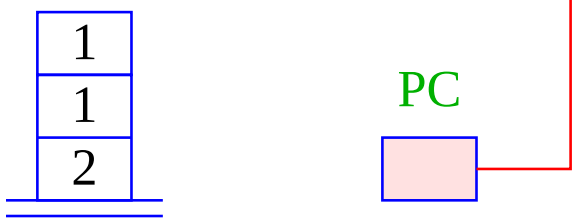
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
```

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

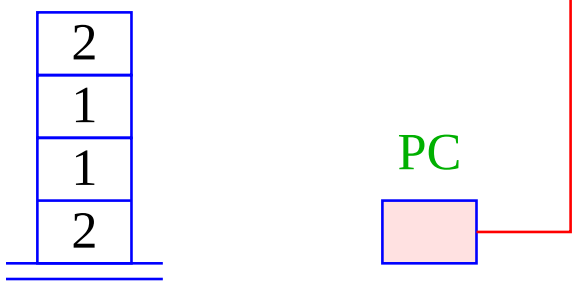




```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B

     JUMP A
B:   LOAD 1
     WRITE
     HALT
```

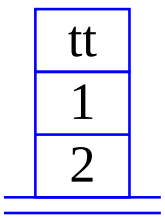


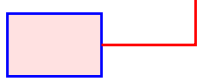
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```



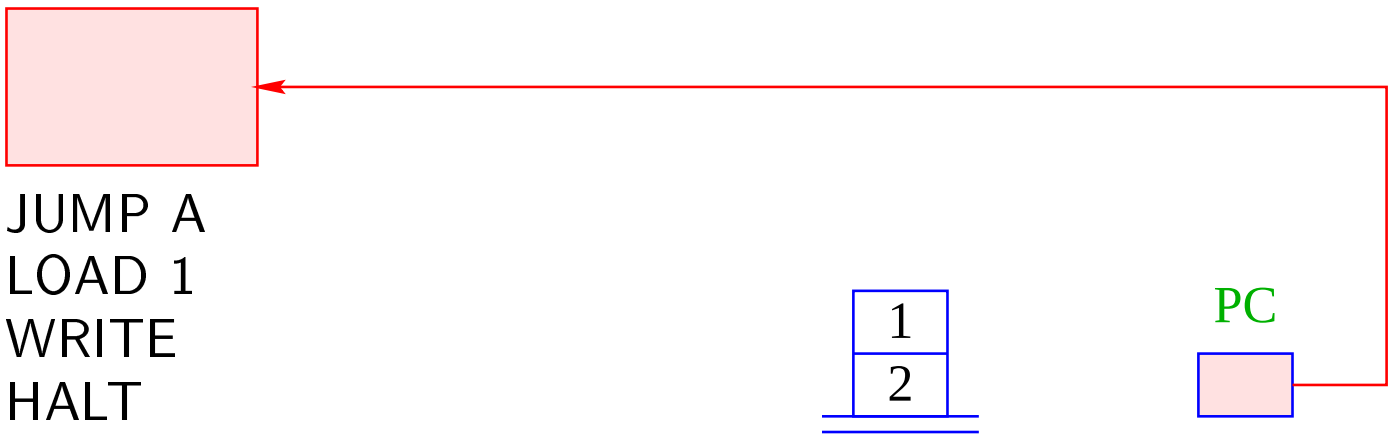
```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B
```

```
     JUMP A
B:   LOAD 1
     WRITE
     HALT
```





```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
```

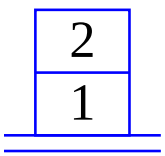


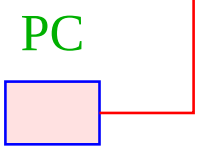

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

```
     ALLOC 2
     READ
     STORE 0
     CONST 1
     STORE 1
A:   CONST 1
     LOAD 0
     LESS
     FJUMP B
     
     JUMP A
B:   LOAD 1
     WRITE
     HALT
```

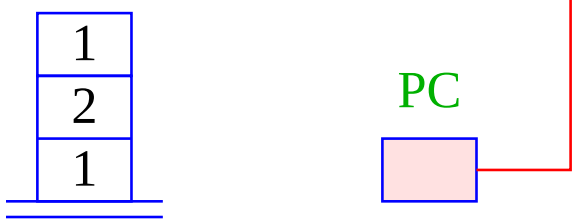
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
```

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

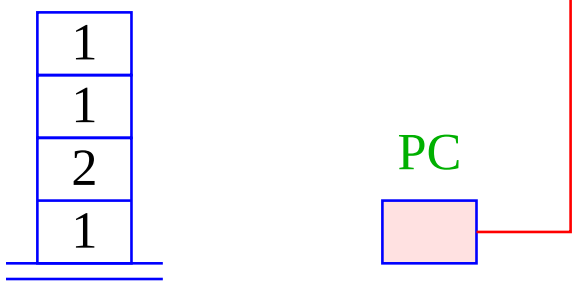




```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

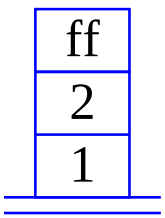


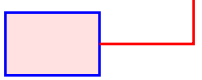
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```



```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
```

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```



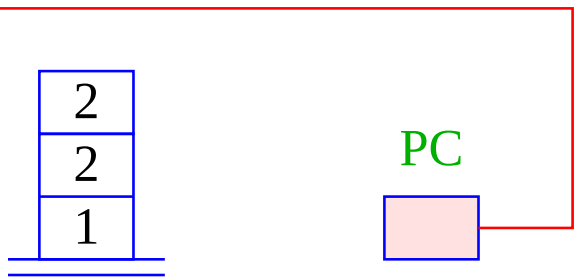


```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B
```

```
    JUMP A
B:  LOAD 1
    WRITE
    HALT
```

```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
```

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```



```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B
```

```
      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

Bemerkungen:

- Die Übersetzungsfunktion, die für ein MiniJava-Programm JVM-Code erzeugt, arbeitet rekursiv auf der Struktur des Programms.
- Im Prinzip lässt sie sich zu einer Übersetzungsfunktion von ganz Java erweitern.
- Zu lösende Übersetzungs-Probleme:
  - mehr Datentypen;
  - Prozeduren;
  - Klassen und Objekte.

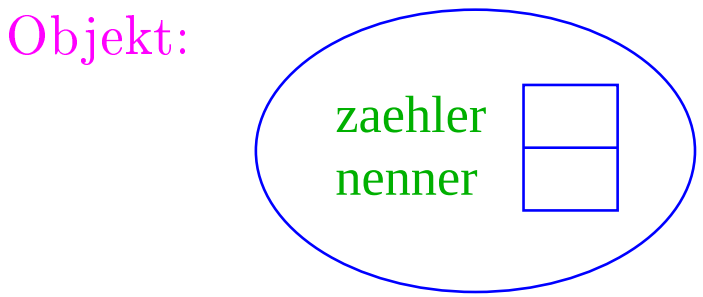
↑Compilerbau

10 Klassen und Objekte

Datentyp	=	Spezifikation von Datenstrukturen
Klasse	=	Datentyp + Operationen
Objekt	=	konkrete Datenstruktur

Beispiel: Rationale Zahlen

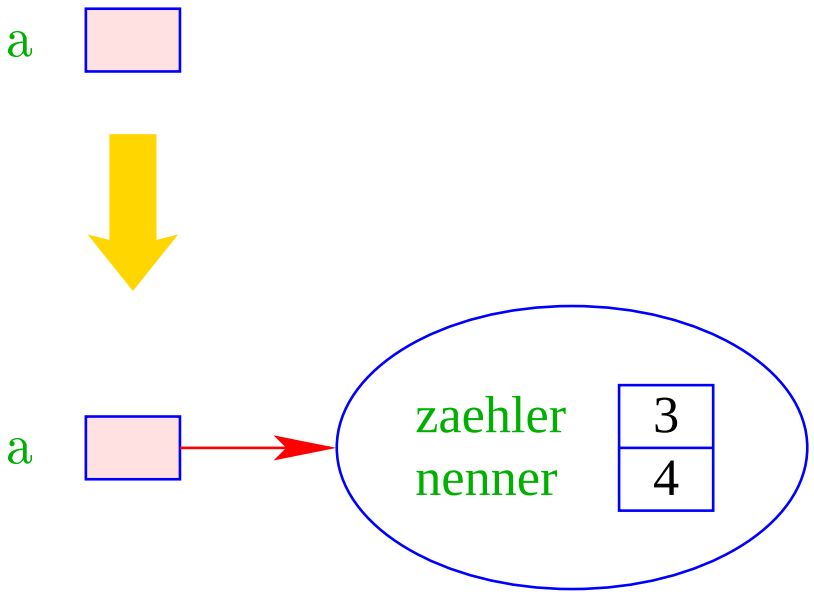
- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $q = \frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q.
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` enthalten:



- Die Daten-Komponenten eines Objekts heißen Instanz-Variablen oder Attribute.

- `Rational` name `;` deklariert eine Variable für Objekte der Klasse `Rational`.
- Das Kommando `new Rational(...)` legt das Objekt an, ruft einen Konstruktor für dieses Objekt auf und liefert das neue Objekt zurück:

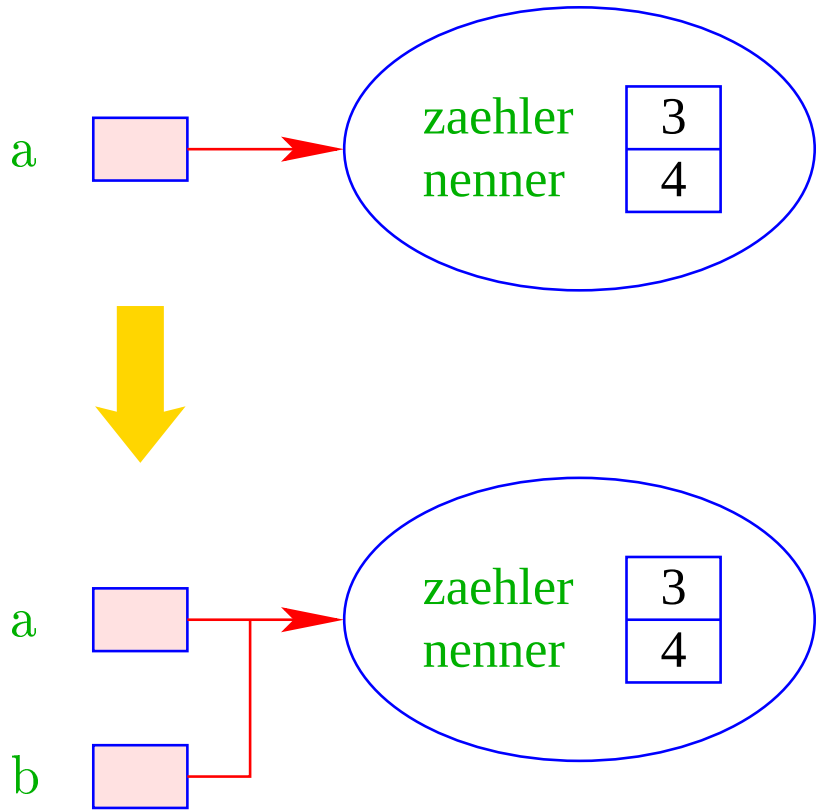
```
a = new Rational (3,4);
```



- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objekts initialisieren kann.

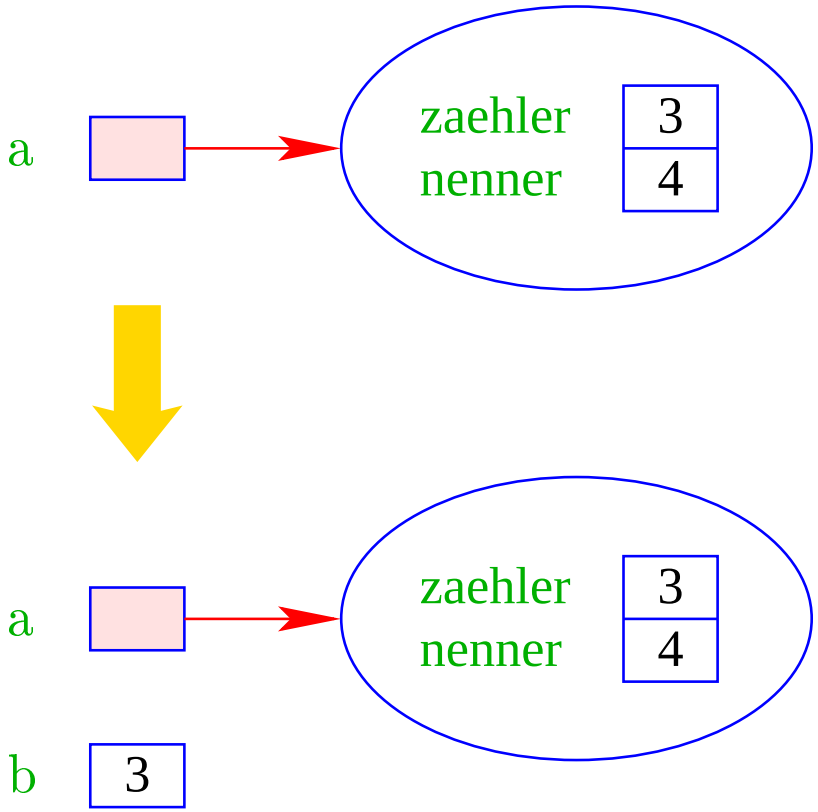
- Der Wert einer `Rational`-Variable ist ein Verweis auf einen Speicherbereich.
- `Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:

```
Rational b = a;
```

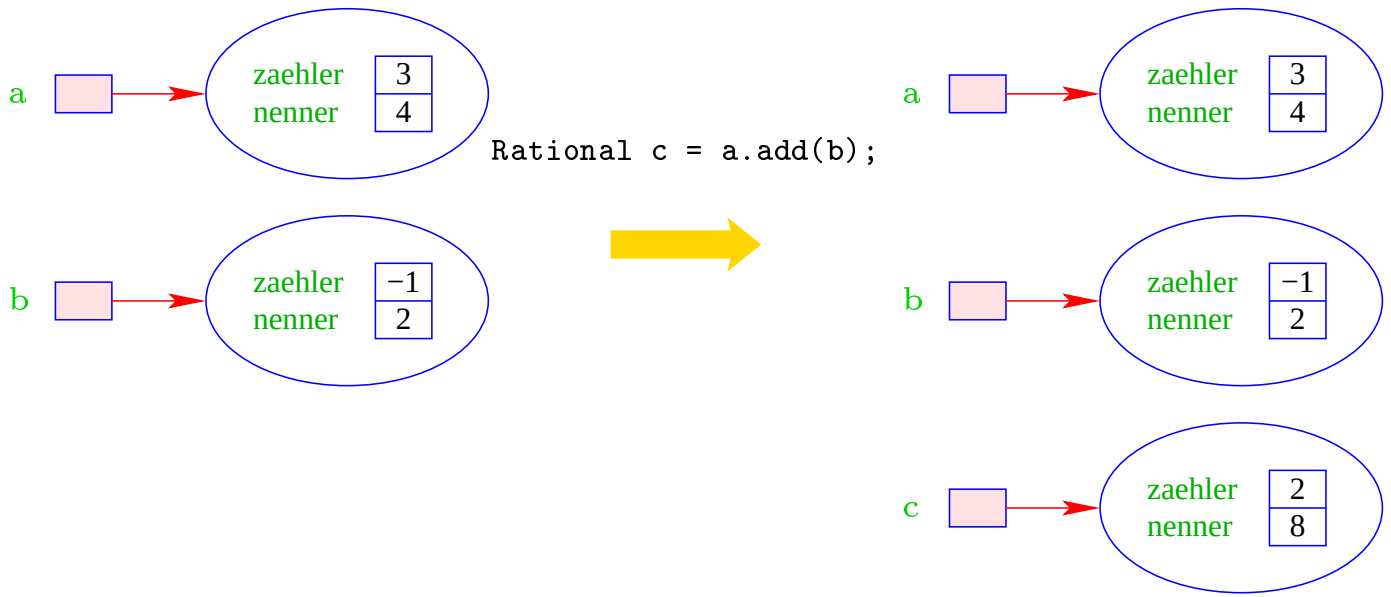


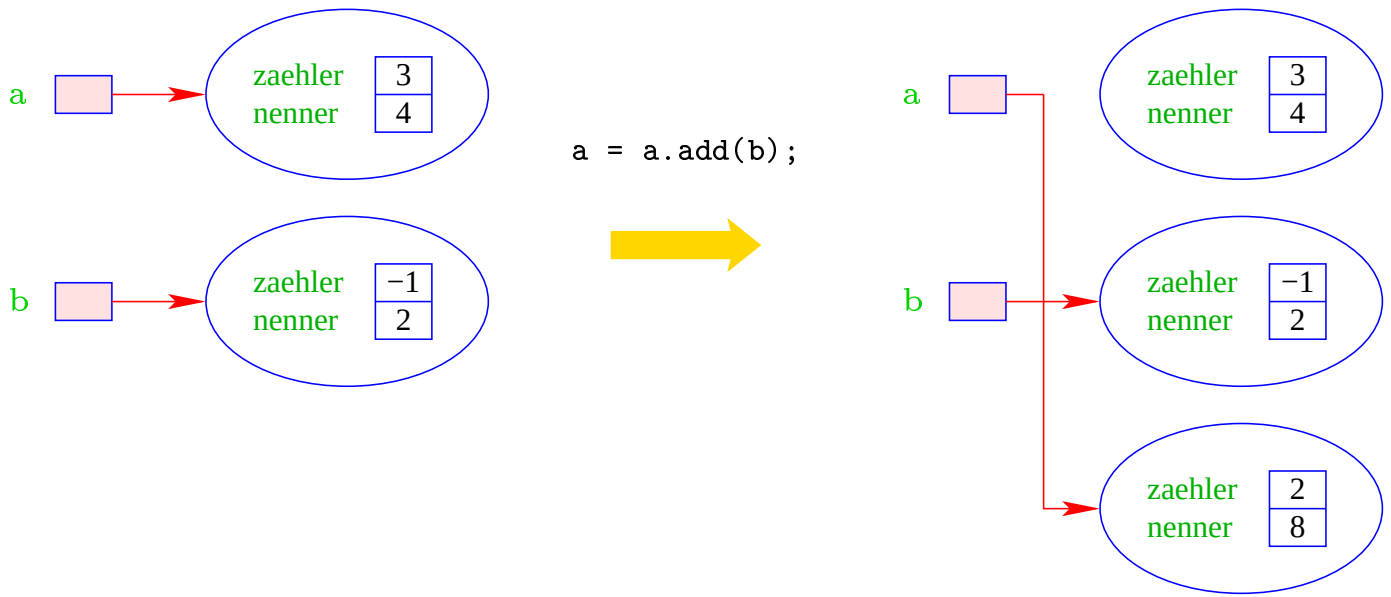
- `a.zaehler` liefert den Wert des Attributs `zaehler` des Objekts `a`:

```
int b = a.zaehler;
```



- `a.add(b)` ruft die Operation `add` für `a` mit dem zusätzlichen aktuellen Parameter `b` auf:





- Die Operationen auf Objekten einer Klasse heißen auch Methoden, genauer: Objekt-Methoden.