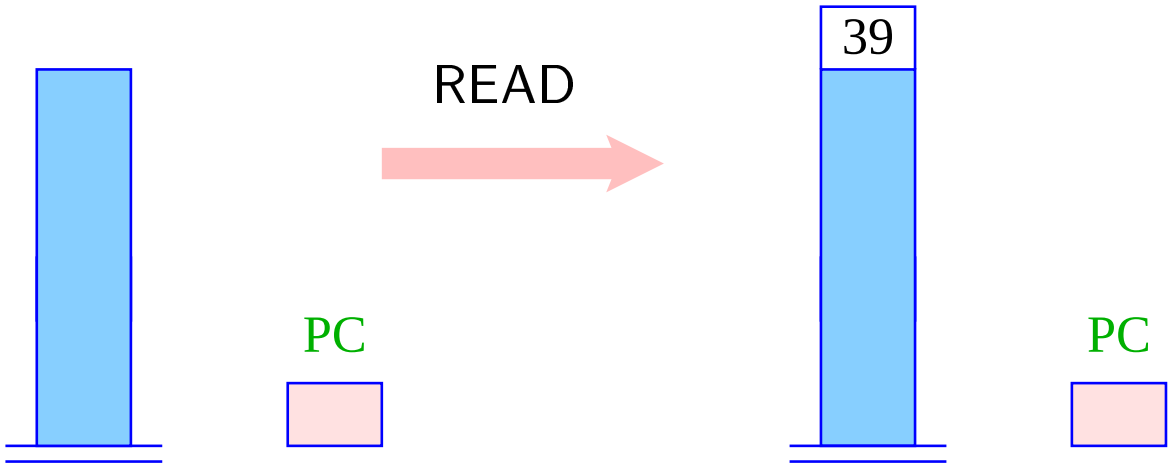
Bevor wir erklären, wie man MiniJava in JVM-Code übersetzt, erklären wir, was die einzelnen Befehle bewirken.

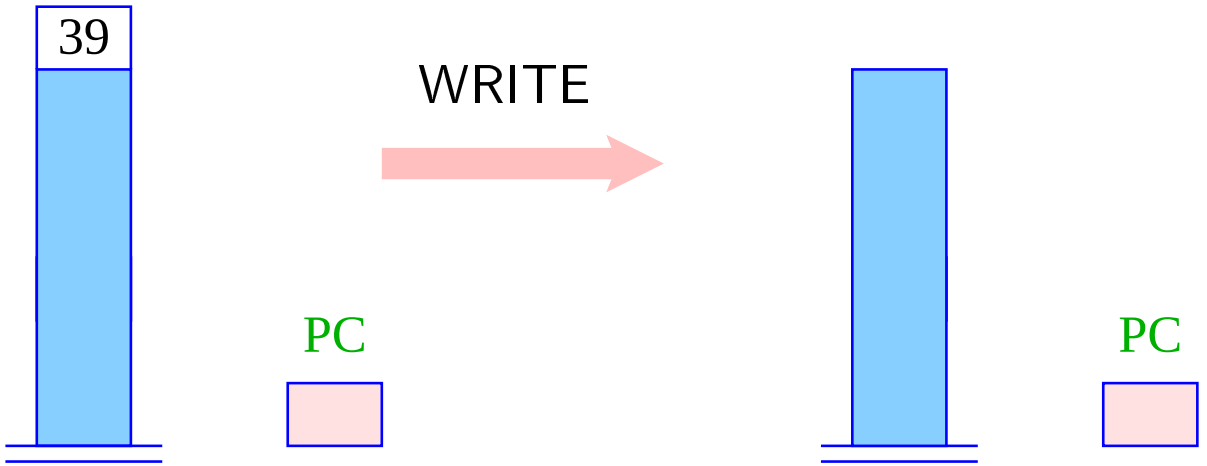
Idee:

- Befehle, die Argumente benötigen, erwarten sie am oberen Ende des Stack.
- Nach ihrer Benutzung werden die Argumente vom Stack herunter geworfen.
- Mögliche Ergebnisse werden oben auf dem Stack abgelegt.

Betrachten wir als Beispiele die IO-Befehle READ und WRITE.



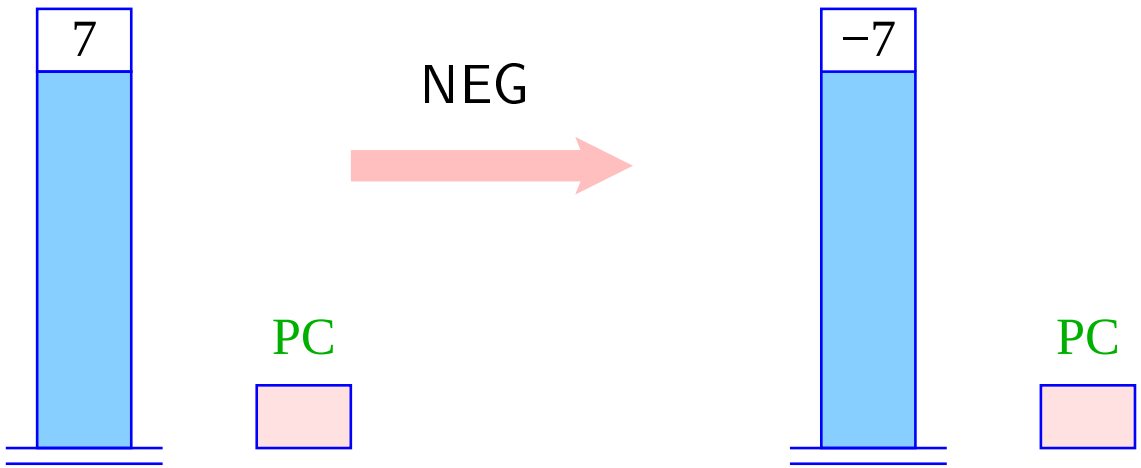
... falls 39 eingegeben wurde

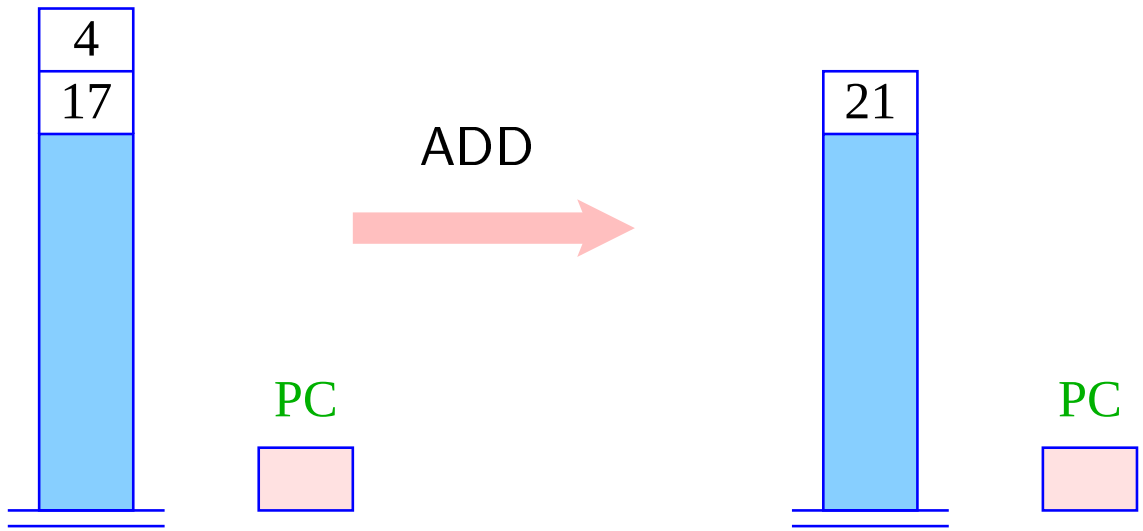


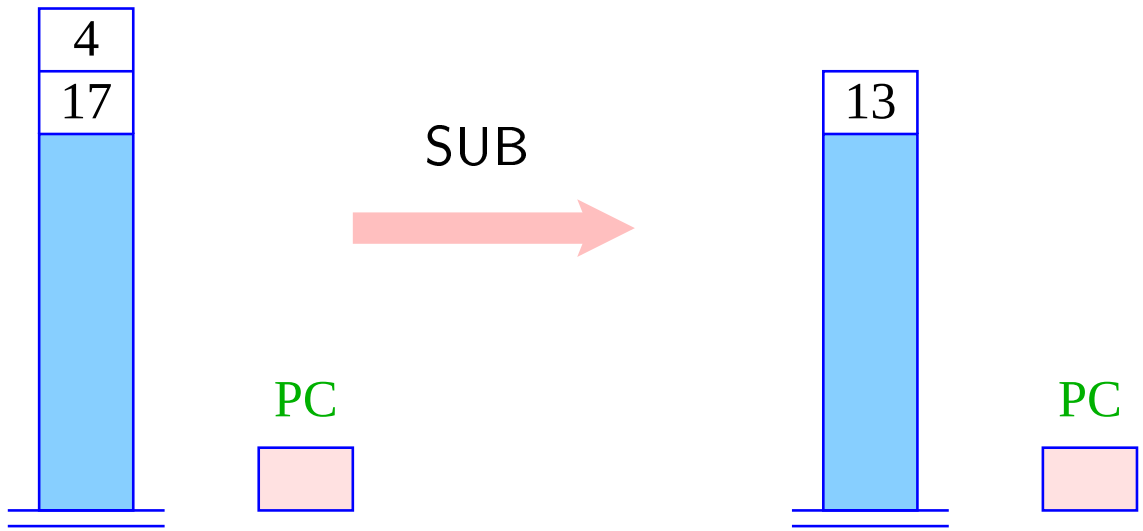
... wobei 39 ausgegeben wird

Arithmetik

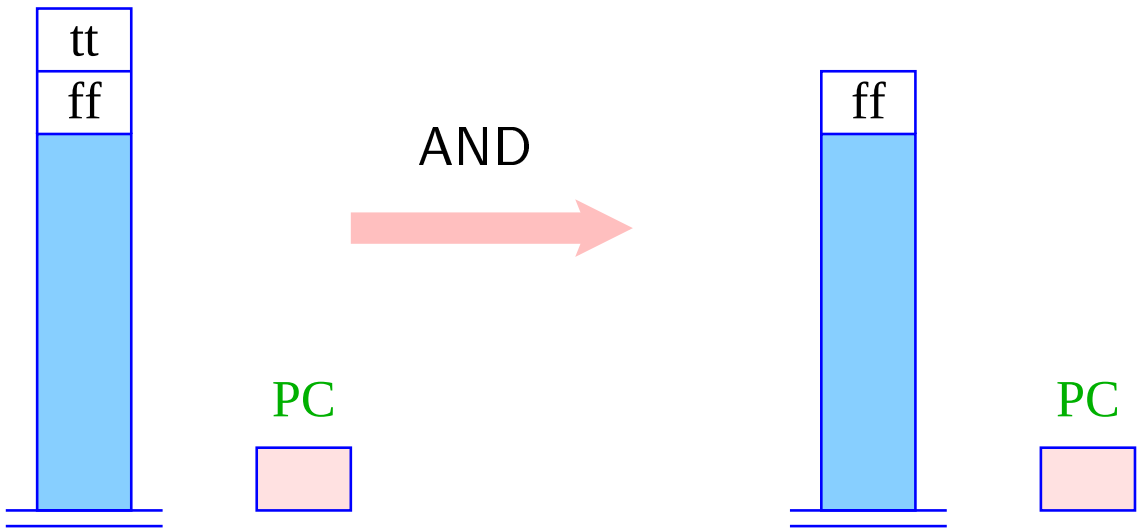
- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.

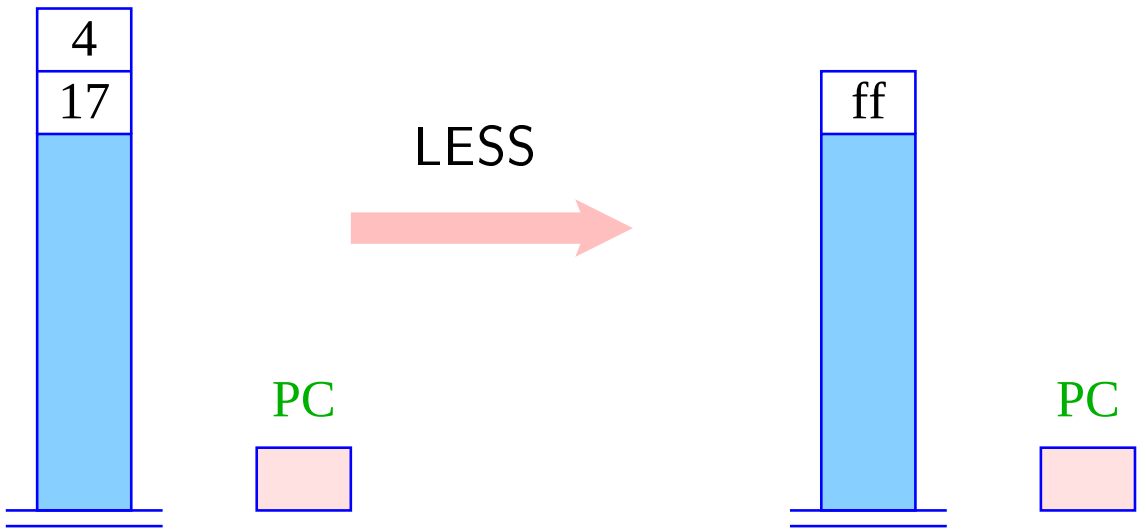

7
NEG
−7
PC
PC


4
17
ADD
21
PC
PC


4
17
SUB
13
PC
PC

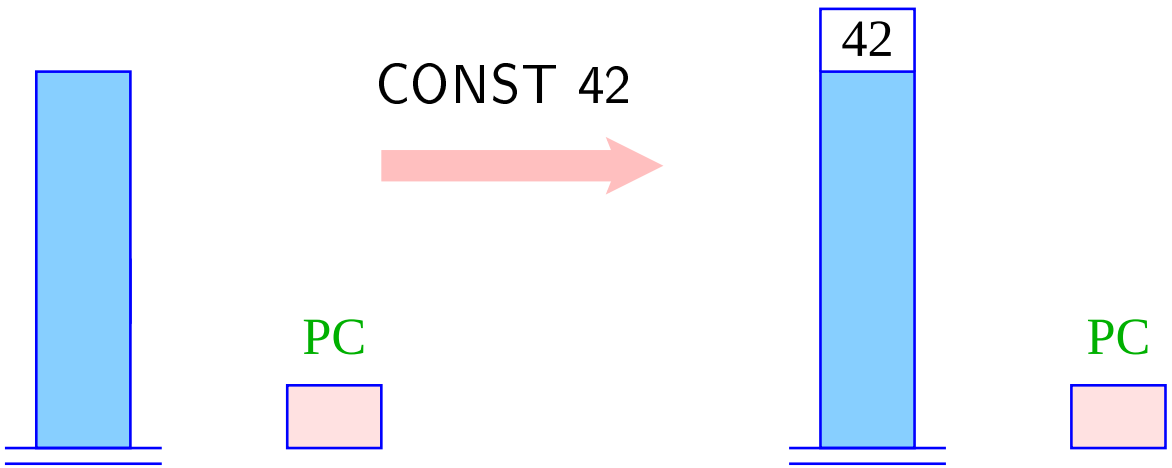
- Die übrigen arithmetischen Operationen `MUL`, `DIV`, `MOD` funktionieren völlig analog.
- Die logischen Operationen `NOT`, `AND`, `OR` ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von `true` und `false` arbeiten (hier: “tt” und “ff”).
- Auch die Vergleiche arbeiten so – nur konsumieren sie ganze Zahlen und liefern einen logischen Wert.

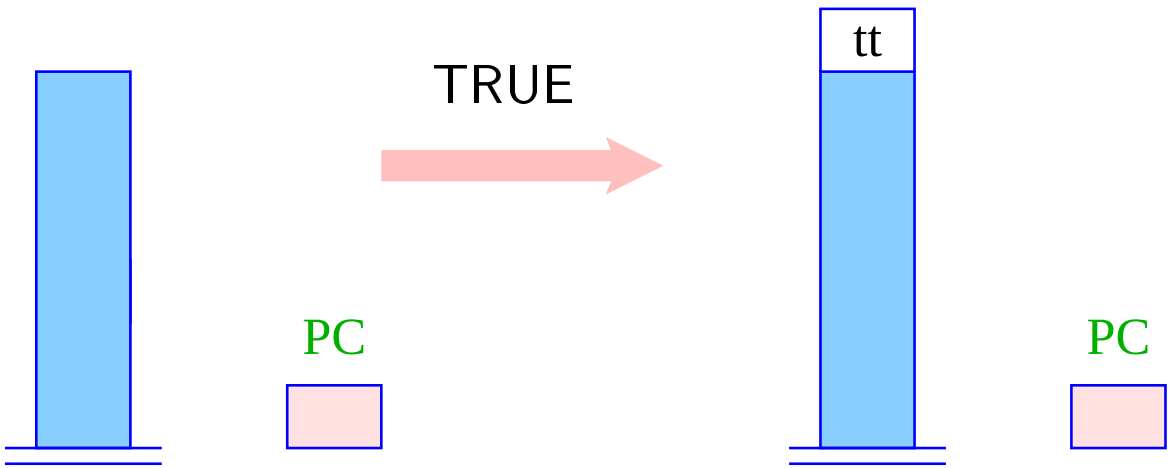

tt
ff
AND
ff
PC
PC

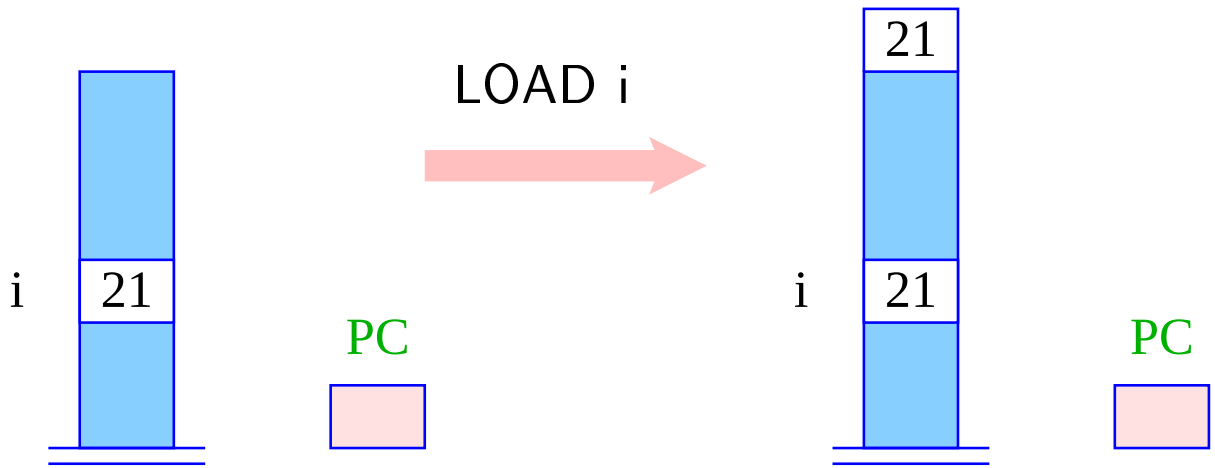

4
17
LESS
ff
PC
PC

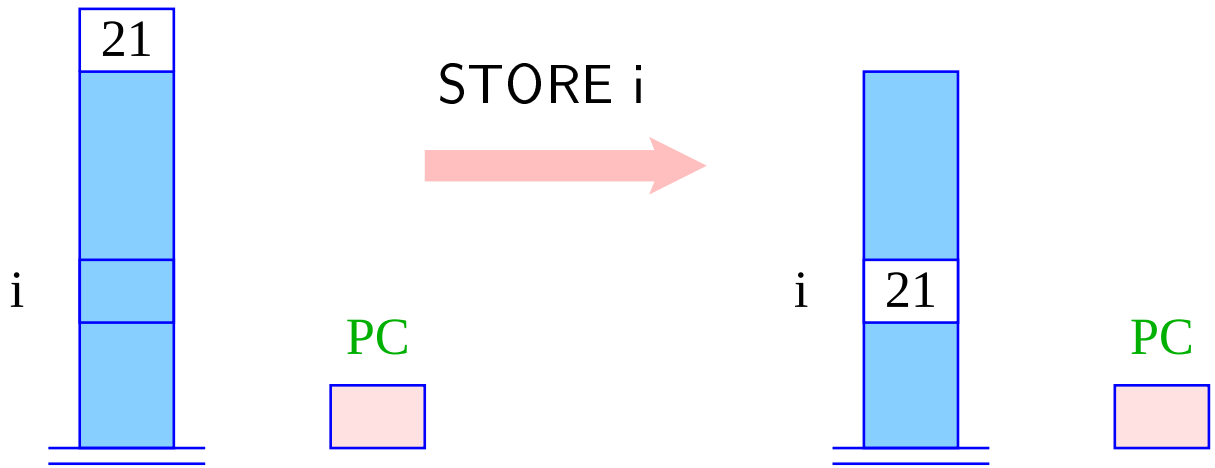
Laden und Speichern

- Konstanten-Lade-Befehle legen einen neuen Wert oben auf dem Stack ab.
- `LOAD i` legt dagegen den Wert aus der i-ten Zelle oben auf dem Stack ab.
- `STORE i` speichert den obersten Wert in der i-ten Zelle ab.


CONST 42
42
PC
PC

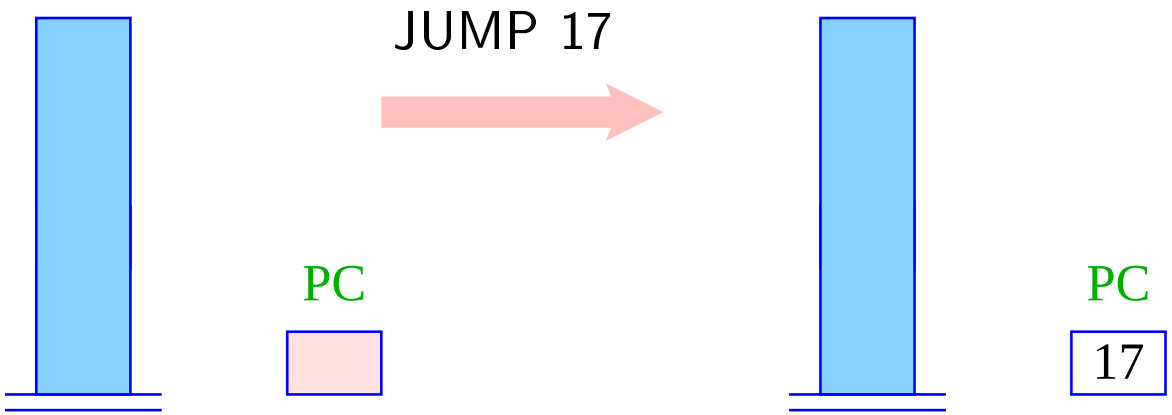

TRUE
tt
PC
PC

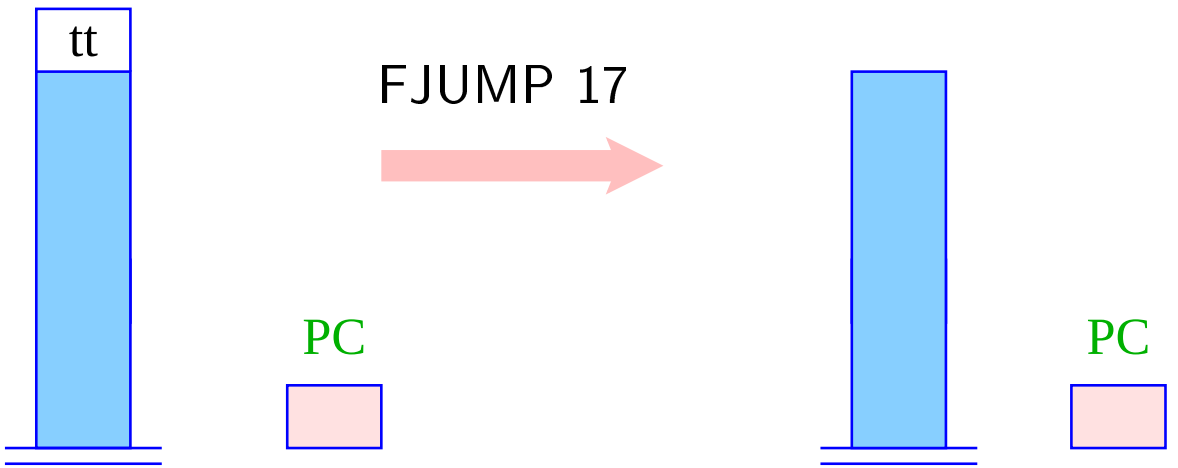

LOAD i
i
21
PC
21
i
21
PC

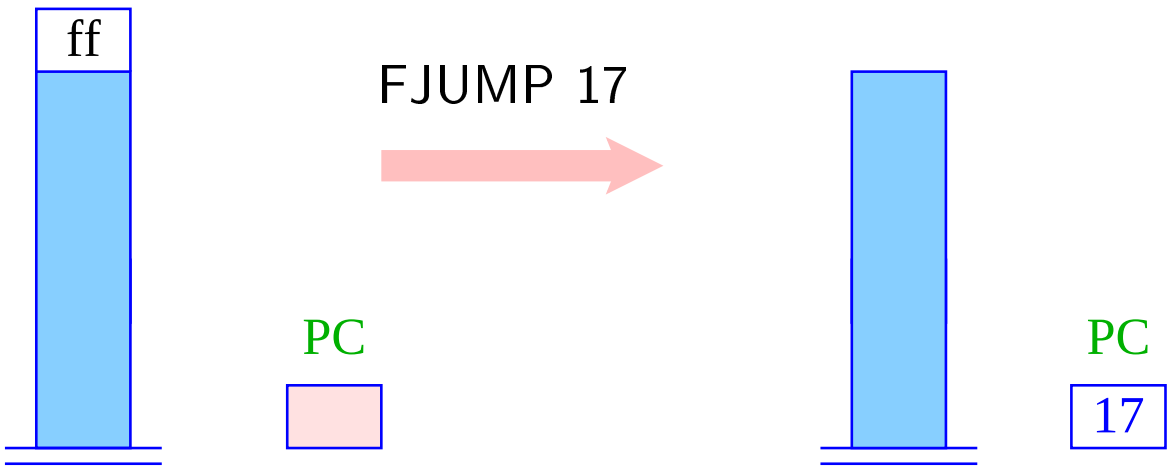

21
STORE i
i
PC
i
21
PC

Sprünge

- Sprünge verändern die Reihenfolge, in der die Befehle abgearbeitet werden, indem sie den PC modifizieren.
- Ein unbedingter Sprung überschreibt einfach den alten Wert des PC mit einem neuen.
- Ein bedingter Sprung tut dies nur, sofern eine geeignete Bedingung erfüllt ist.

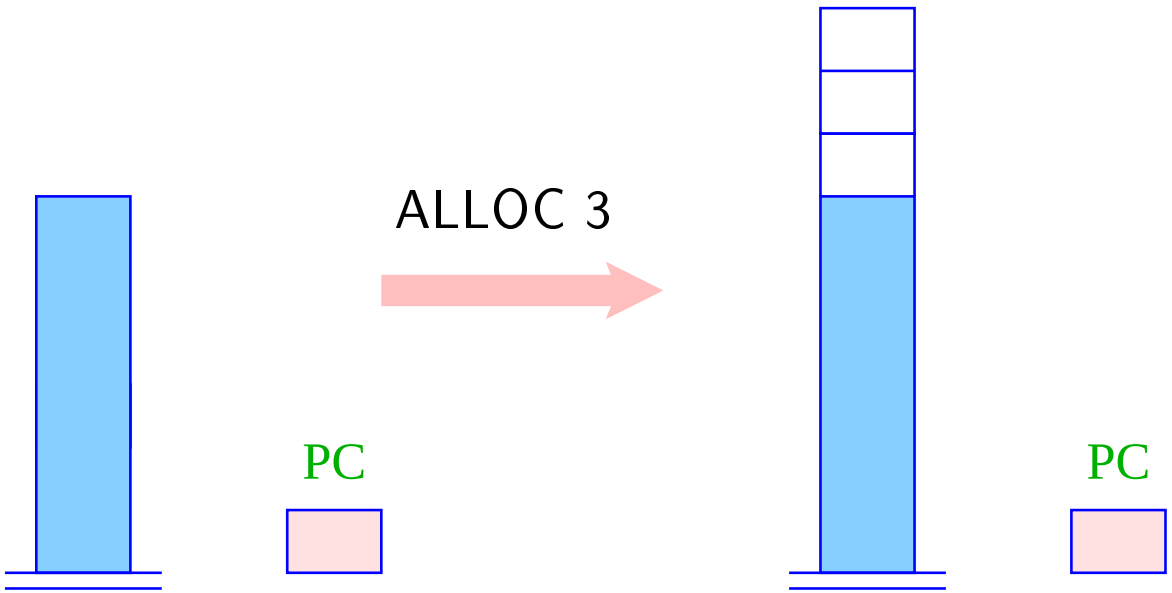

JUMP 17
PC
PC
17


tt
FJUMP 17
PC
PC


ff
FJUMP 17
PC
PC
17

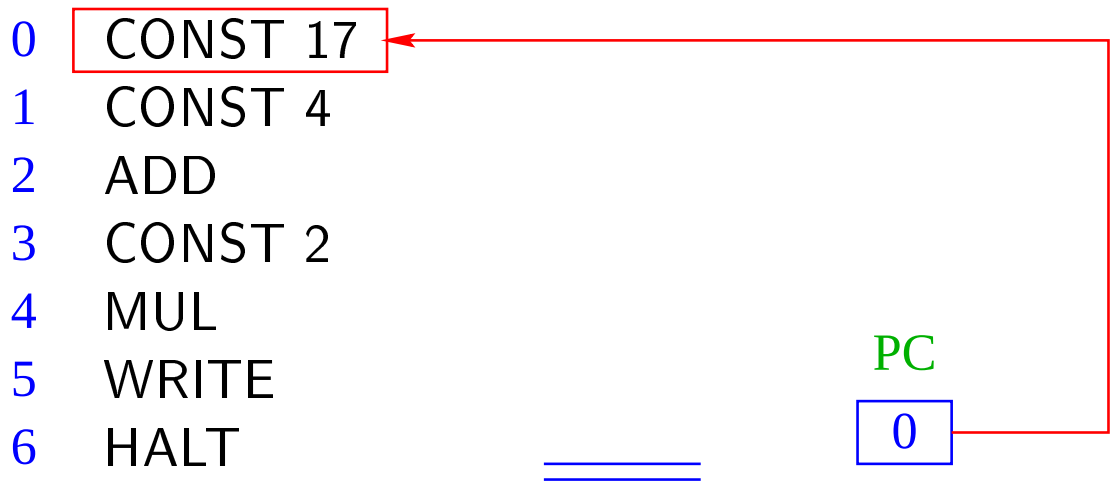
Allokierung von Speicherplatz

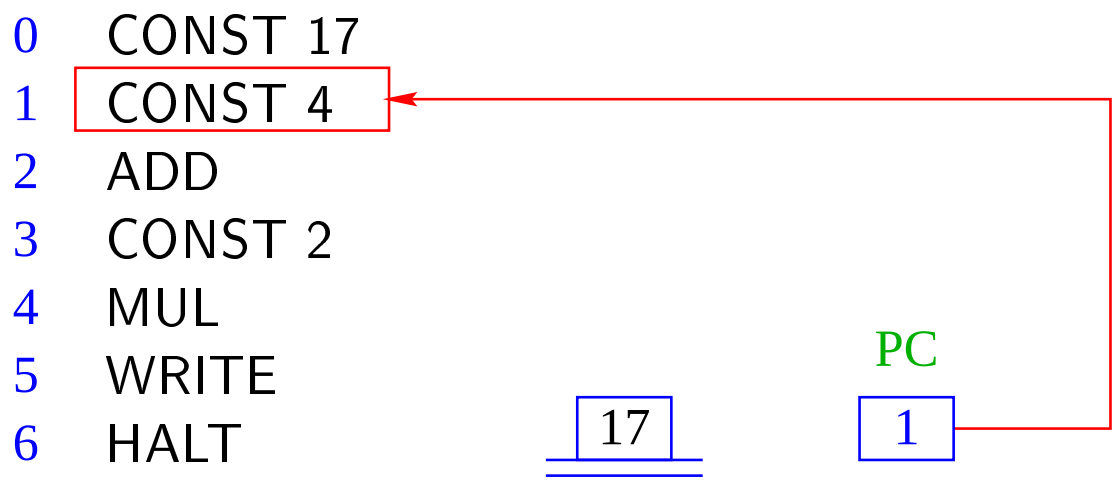
- Wir beabsichtigen, jeder Variablen unseres MiniJava-Programms eine Speicher-Zelle zuzuordnen.
- Um Platz für i Variablen zu schaffen, muss der SP einfach um i erhöht werden.
- Das ist die Aufgabe von `ALLOC i`.

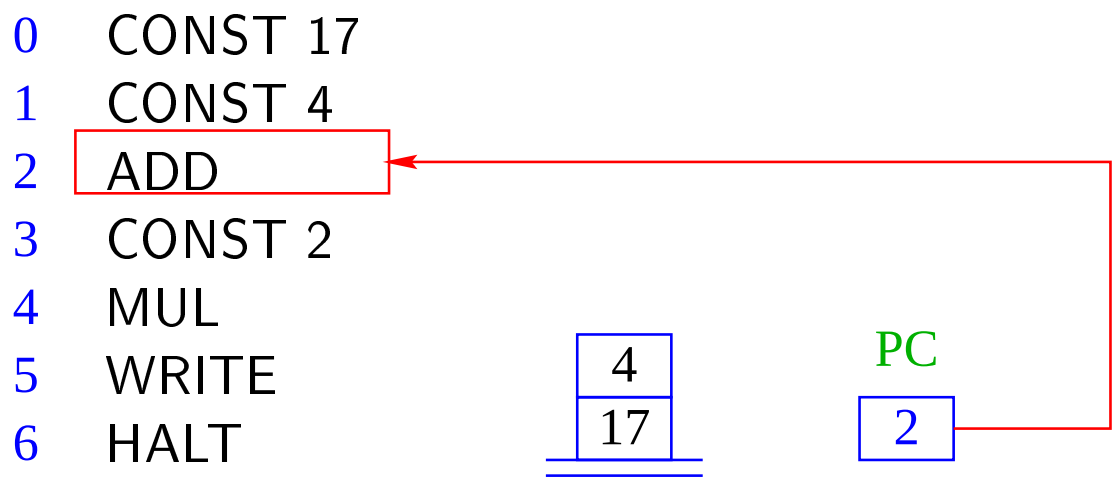

ALLOC 3
PC
PC

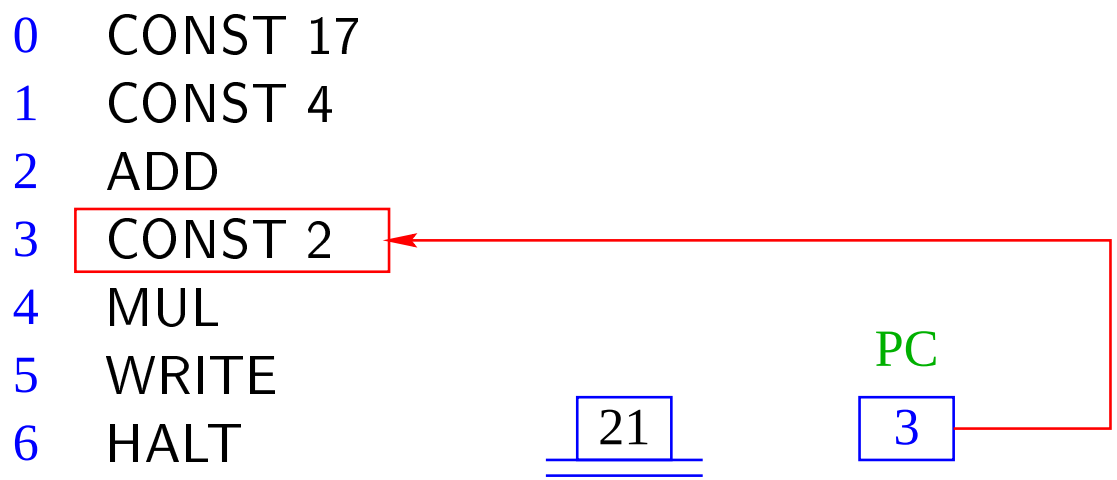
Ein Beispiel-Programm:

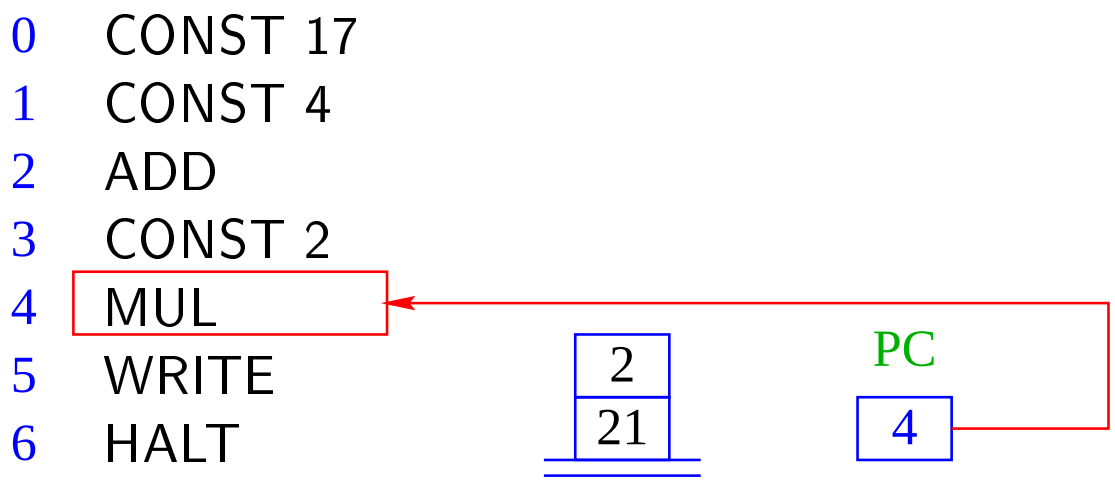
```
CONST 17
CONST 4
ADD
CONST 2
MUL
WRITE
HALT
```

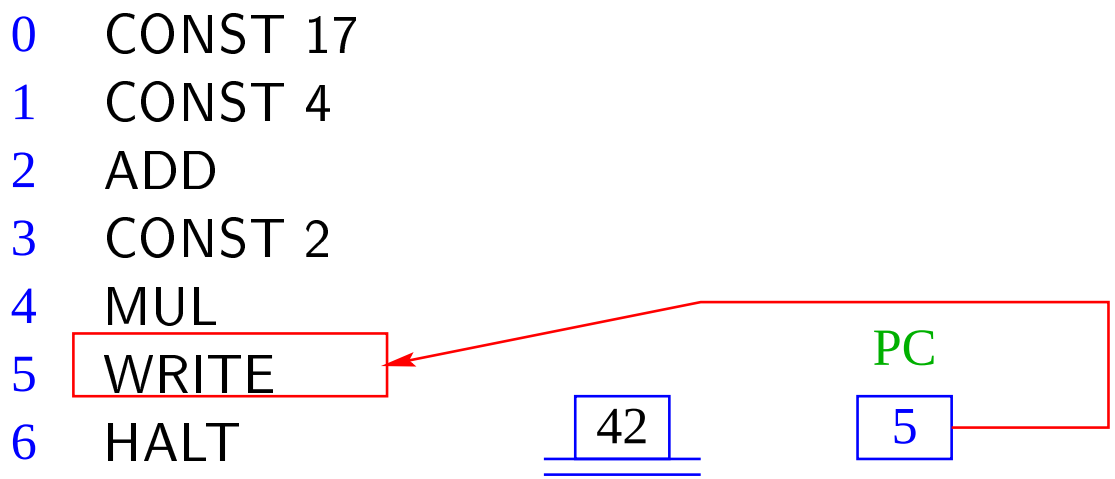

0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
0

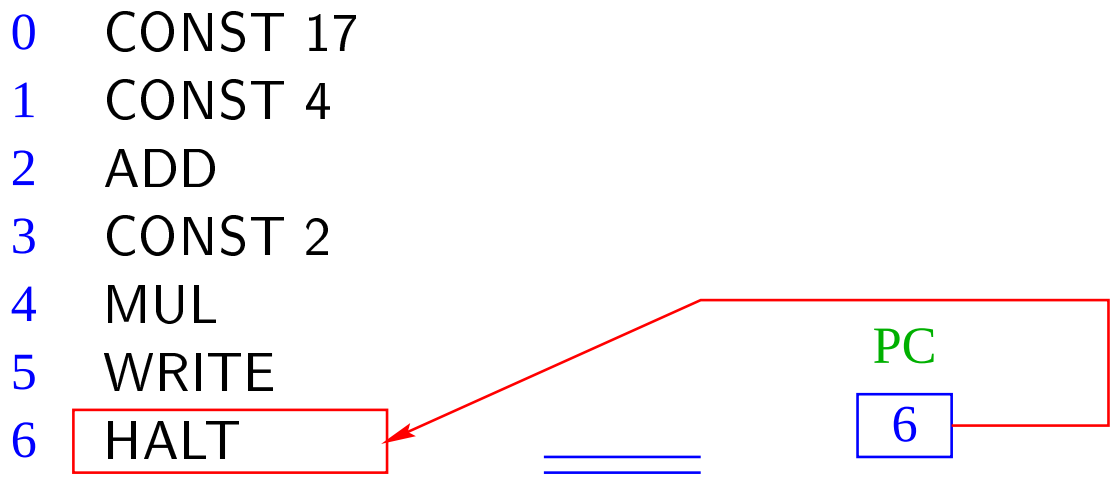

0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
1
17


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
4
17
PC
2


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
3
21


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
2
21
PC
4


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
5
42


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
6

Ausführung eines JVM-Programms:

```
PC = 0;
IR = Code[PC];
while (IR != HALT) {
   PC = PC + 1;
   execute(IR);
   IR = Code[PC];
}
```

- IR = Instruction Register, d.h. eine Variable, die den nächsten auszuführenden Befehl enthält.
- `execute(IR)` führt den Befehl in IR aus.
- `Code[PC]` liefert den Befehl, der in der Zelle in Code steht, auf die PC zeigt.

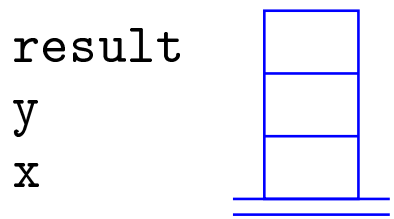
9.1 Übersetzung von Deklarationen

Betrachte Deklaration

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

Idee:

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



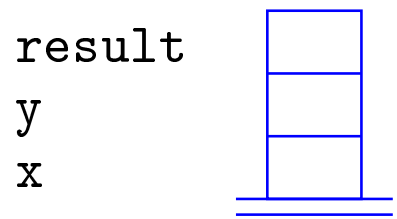
9.1 Übersetzung von Deklarationen

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:

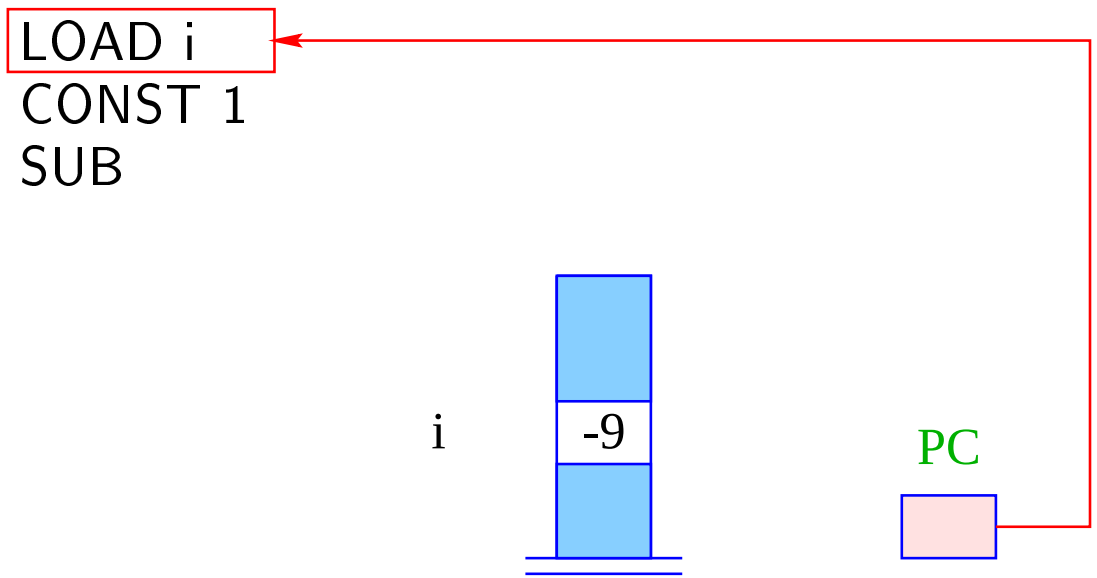
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

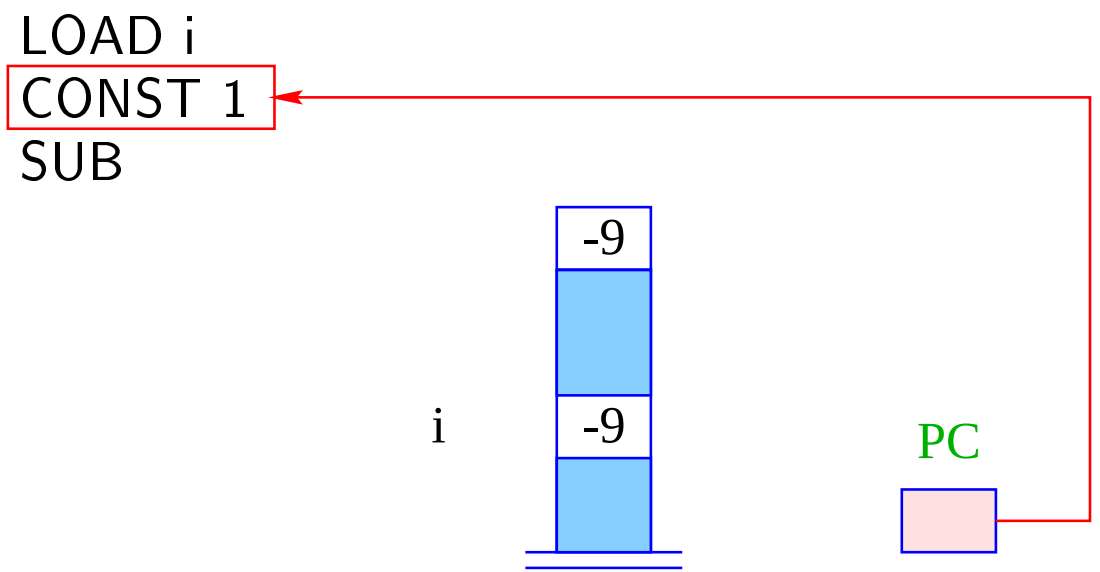
9.2 Übersetzung von Ausdrücken

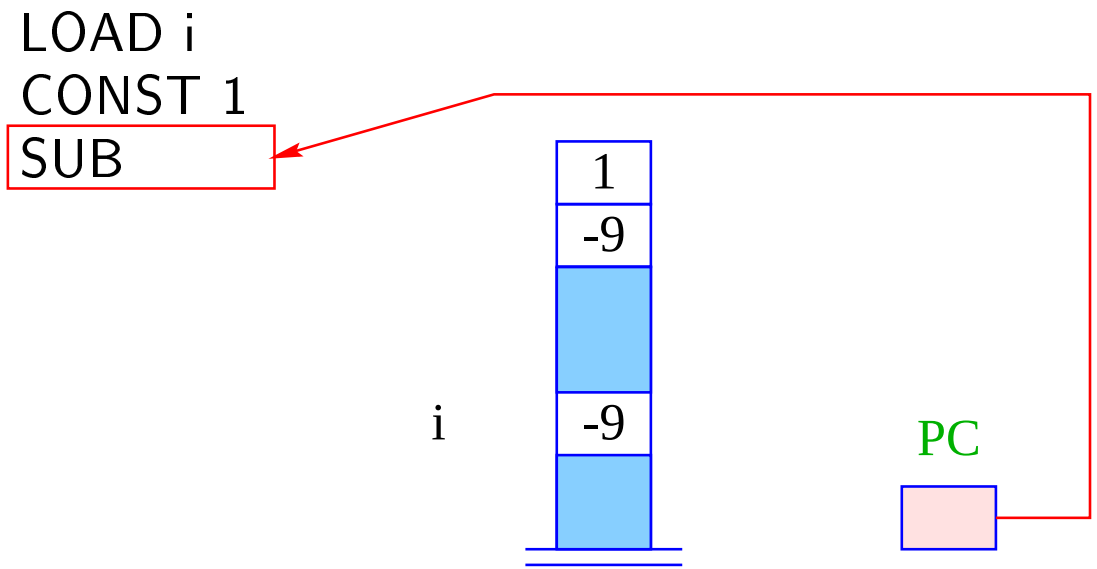
Idee:

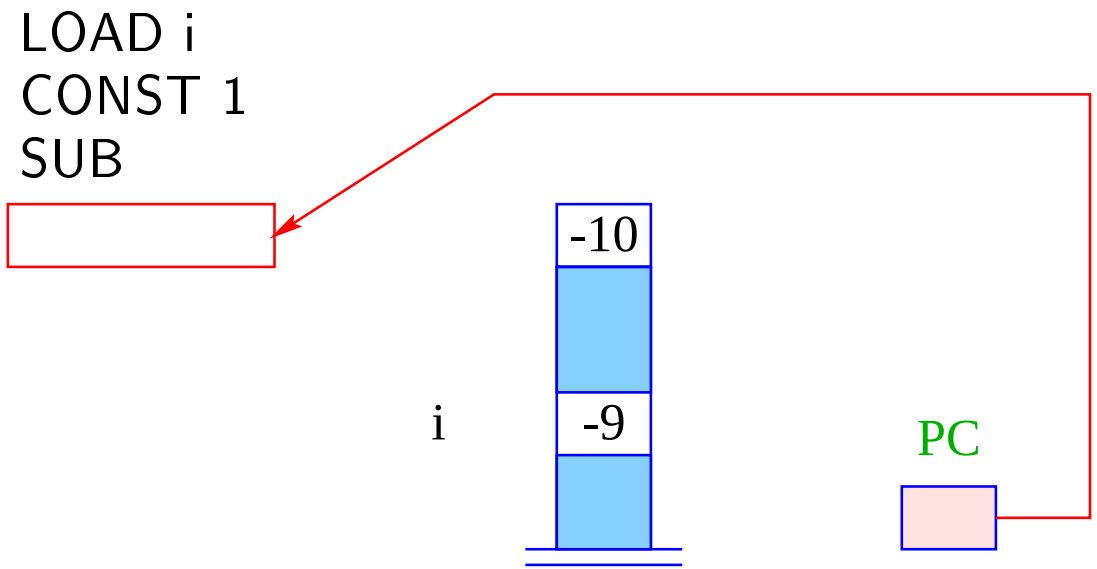
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

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!

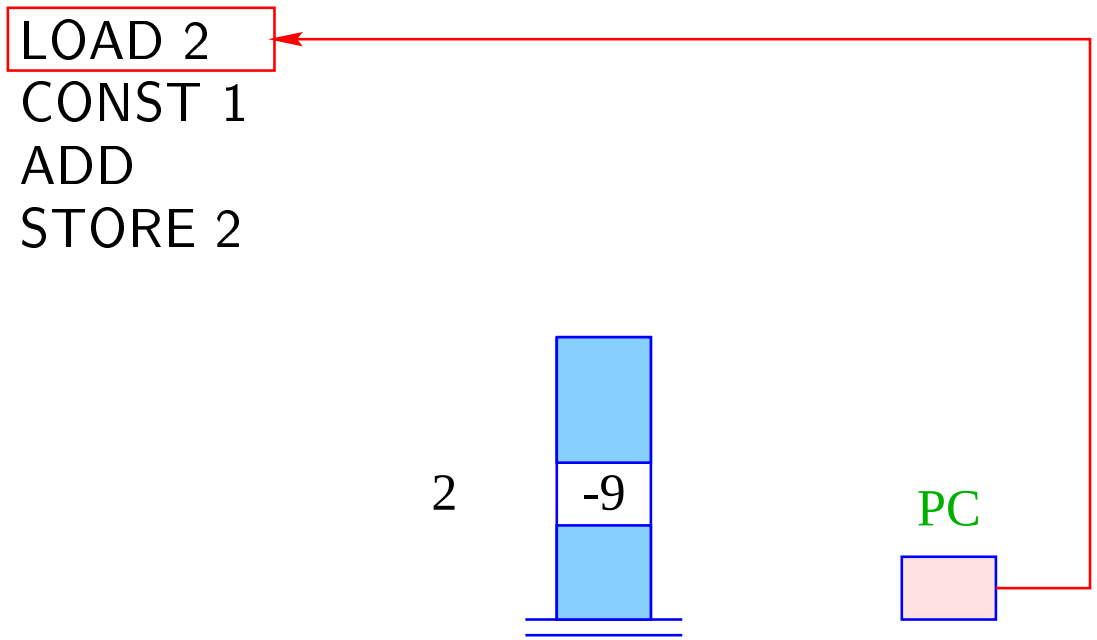
Sei `x` die Variable Nr. i. Dann ist

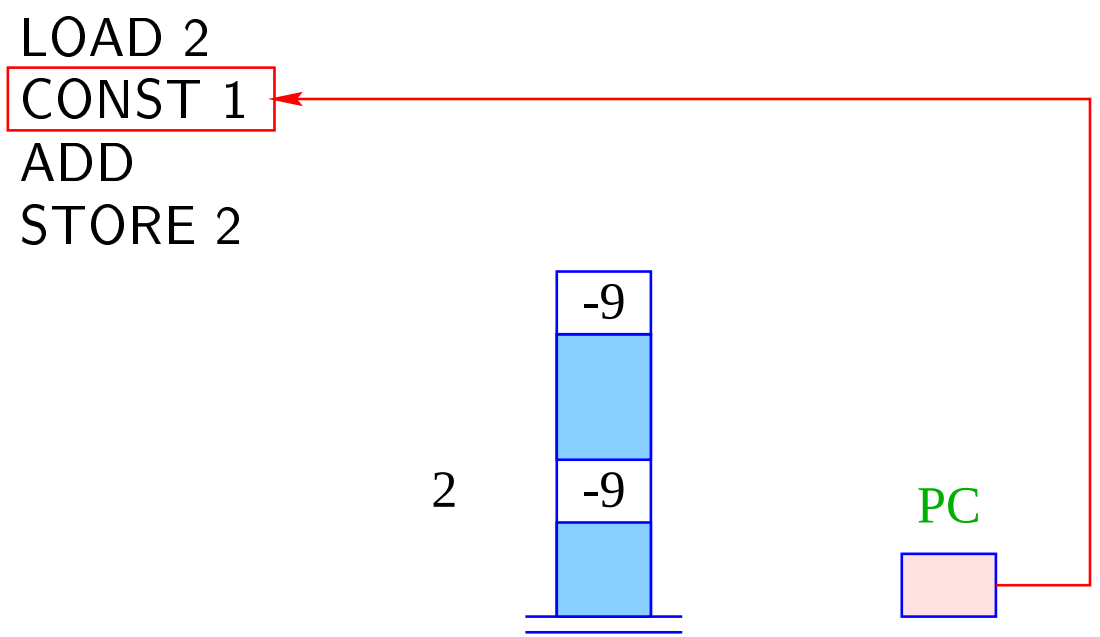
Übersetzung von `x = expr;` = Übersetzung von `expr`
`STORE i`

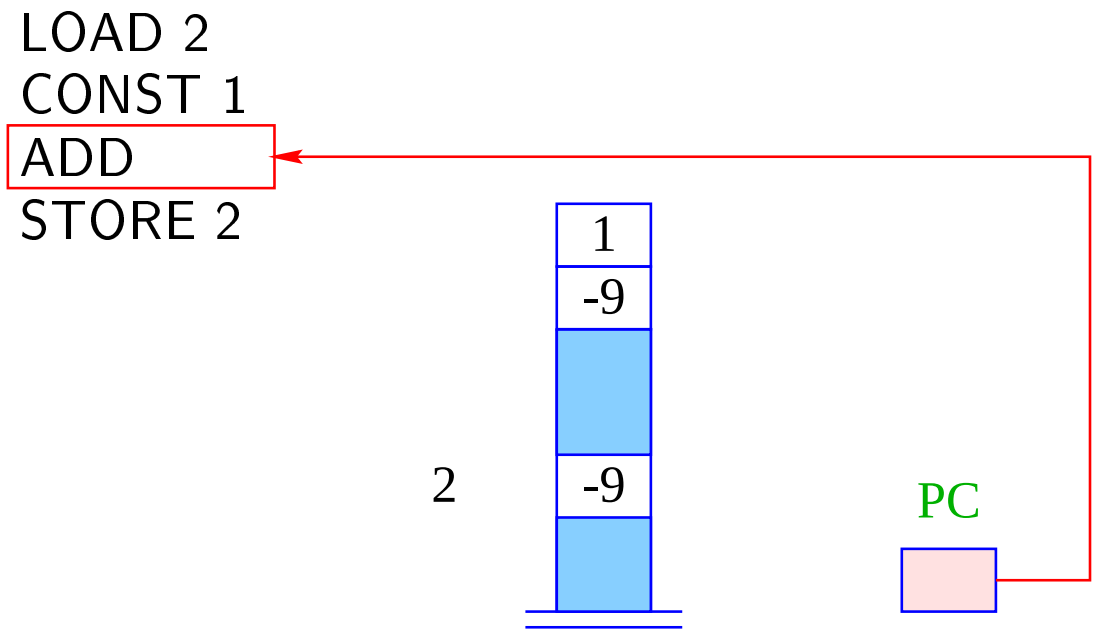
Beispiel:

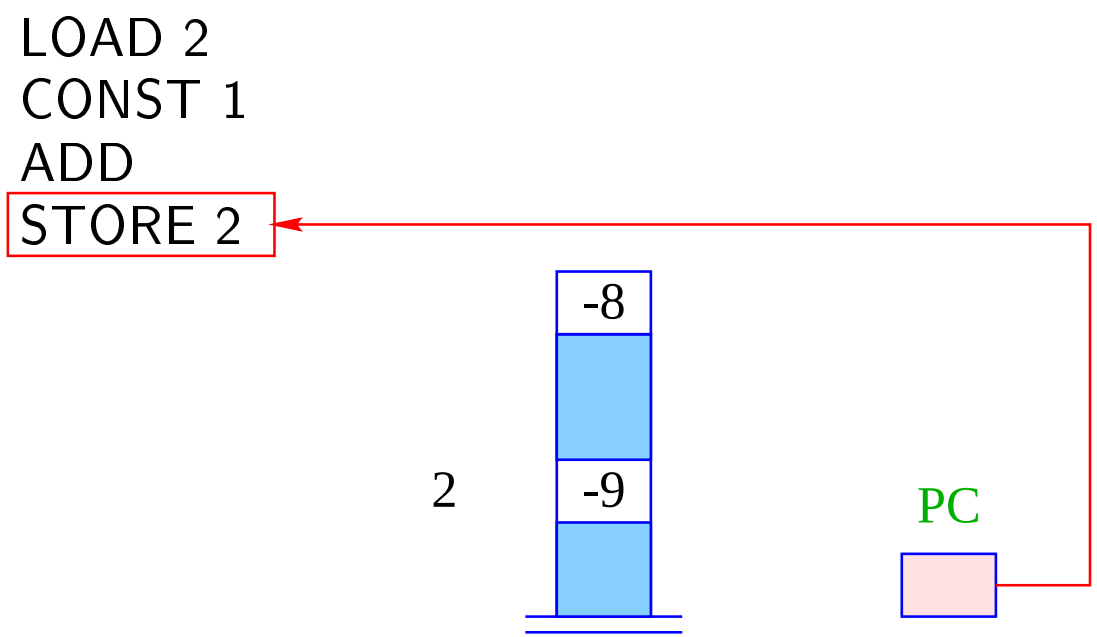
Für `x = x + 1;` (x die 2. Variable) liefert das:

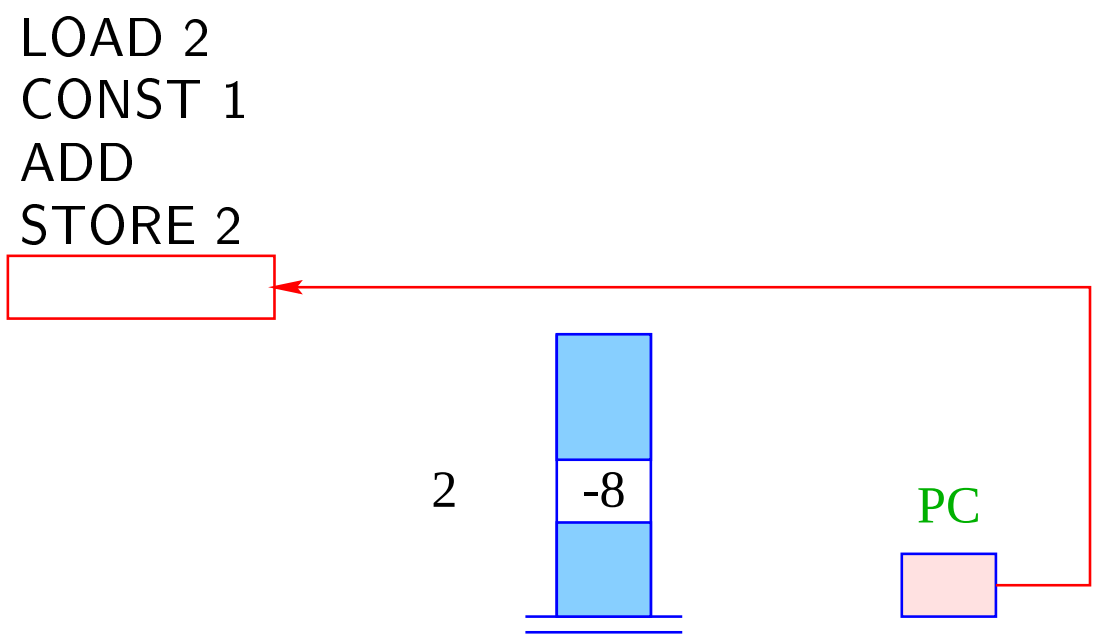
```
LOAD 2
CONST 1
ADD
STORE 2
```


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.

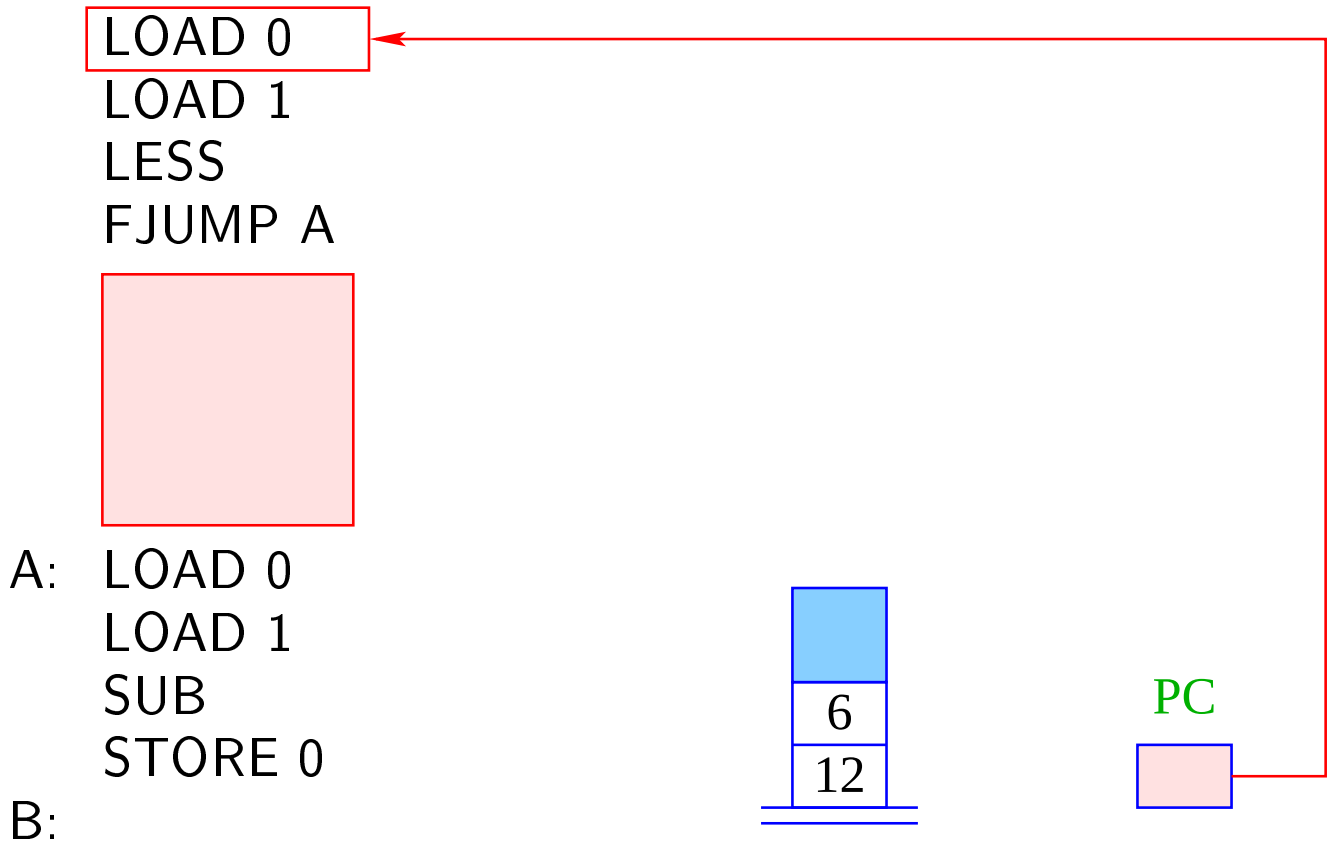
Beispiel:

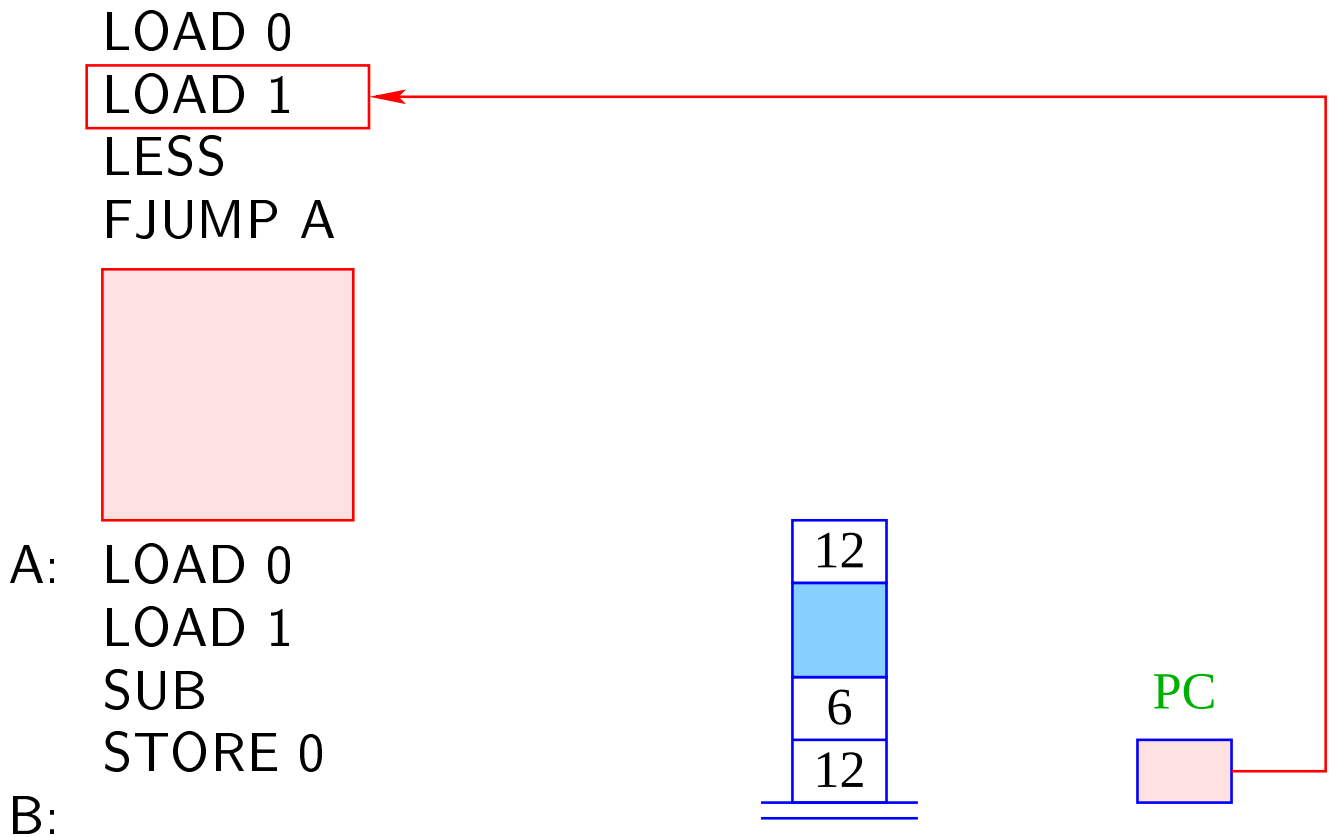
Für das Statement:

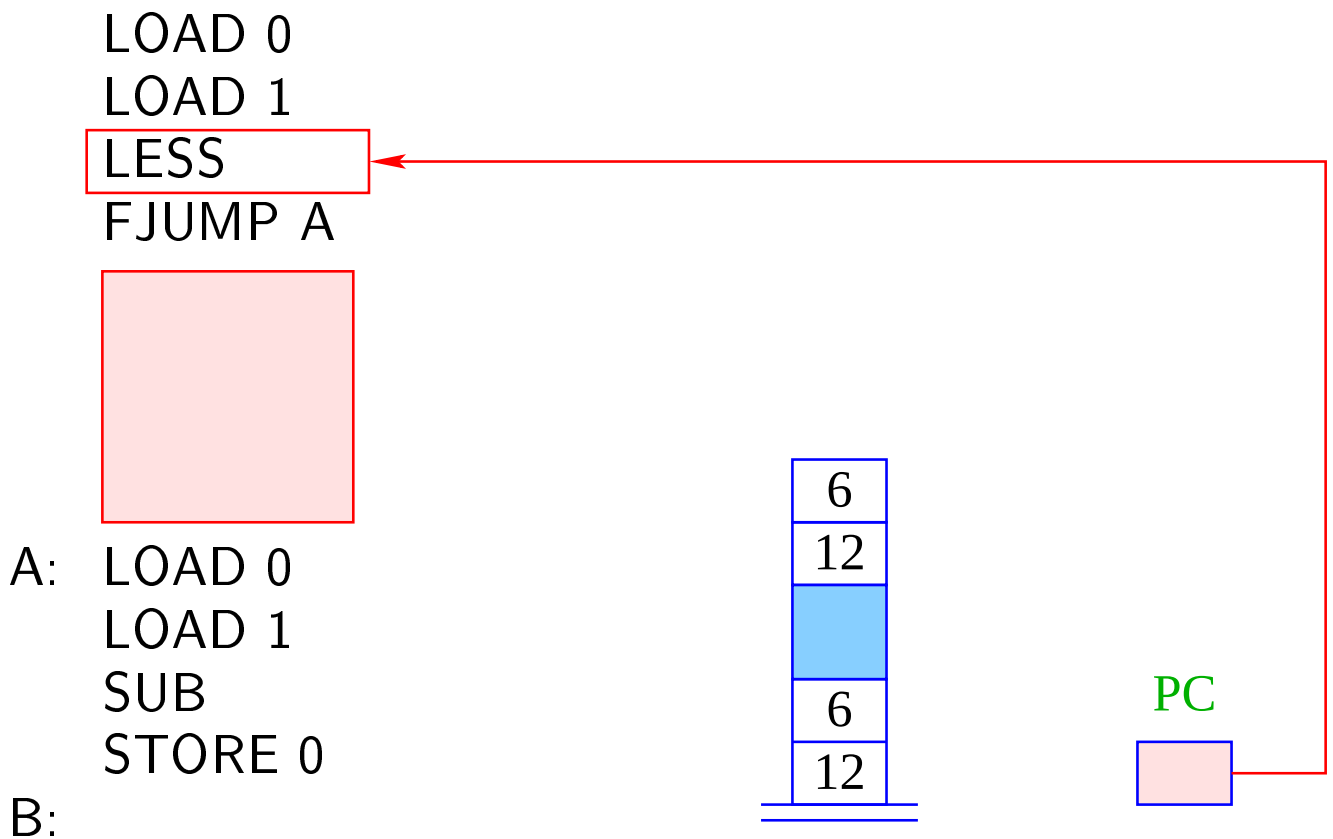
```
if (x < y) y = y - x;
else x = x - y;
```

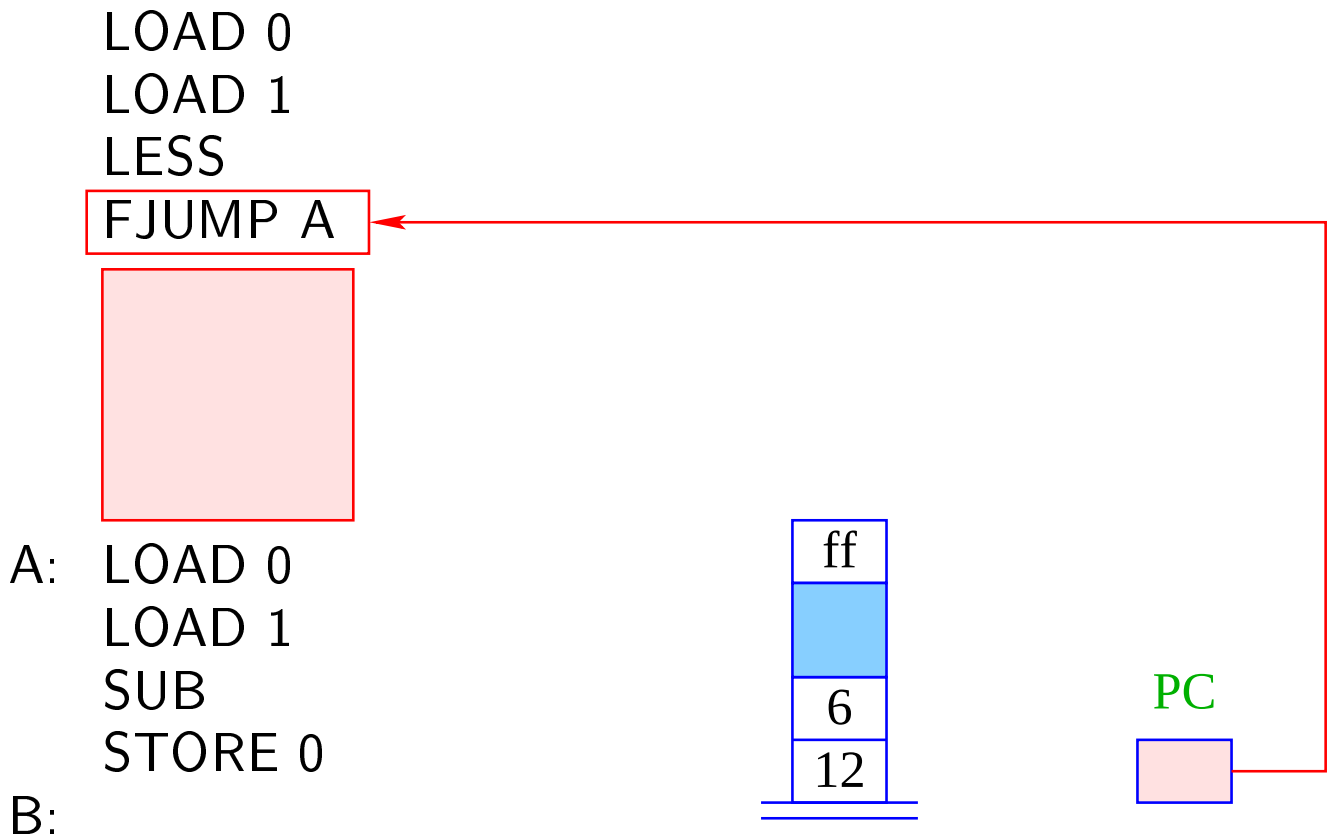
(`x` und `y` die 0. bzw. 1. Variable) ergibt das:

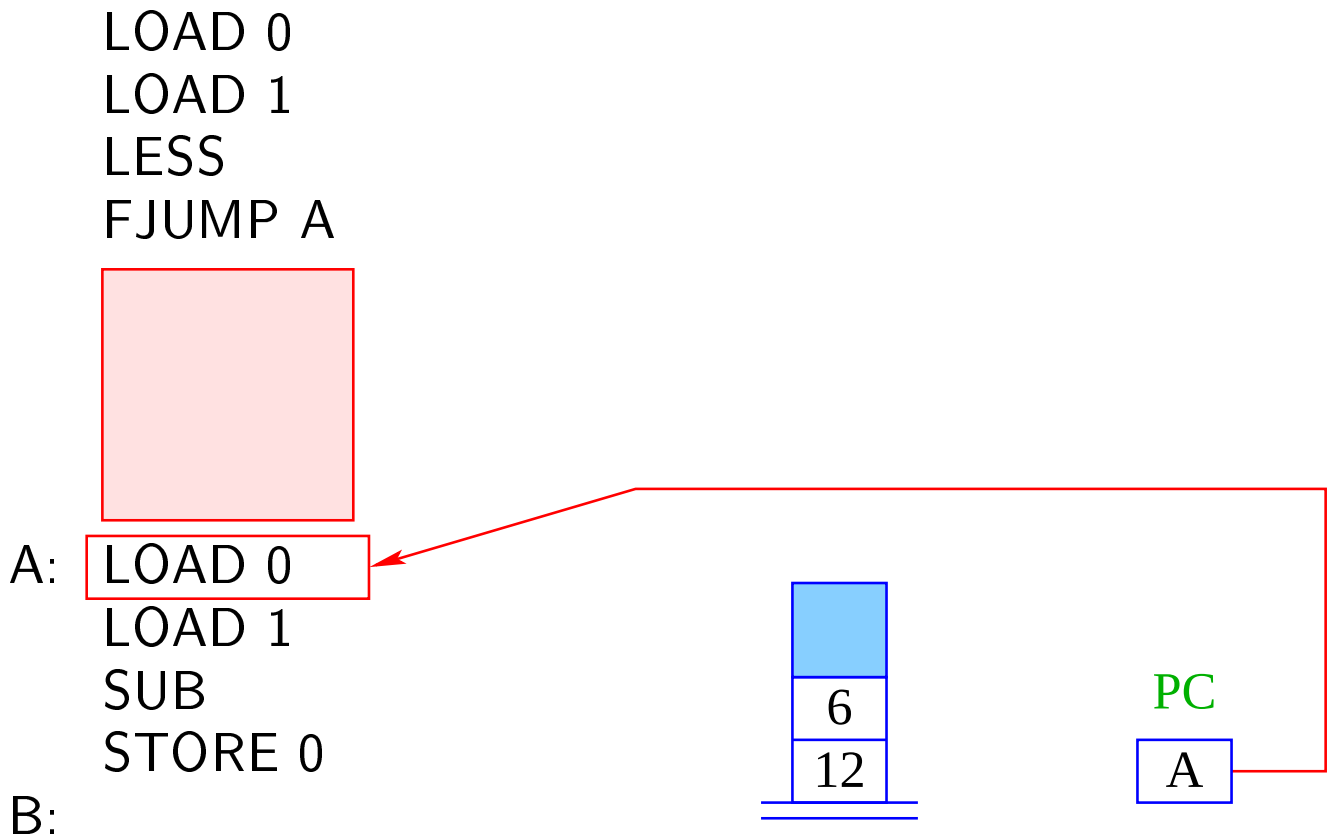
```
    LOAD 0          LOAD 1          A:  LOAD 0
    LOAD 1          LOAD 0              LOAD 1
    LESS            SUB                 SUB
    FJUMP A         STORE 1             STORE 0
                    JUMP B          B:  ...
```

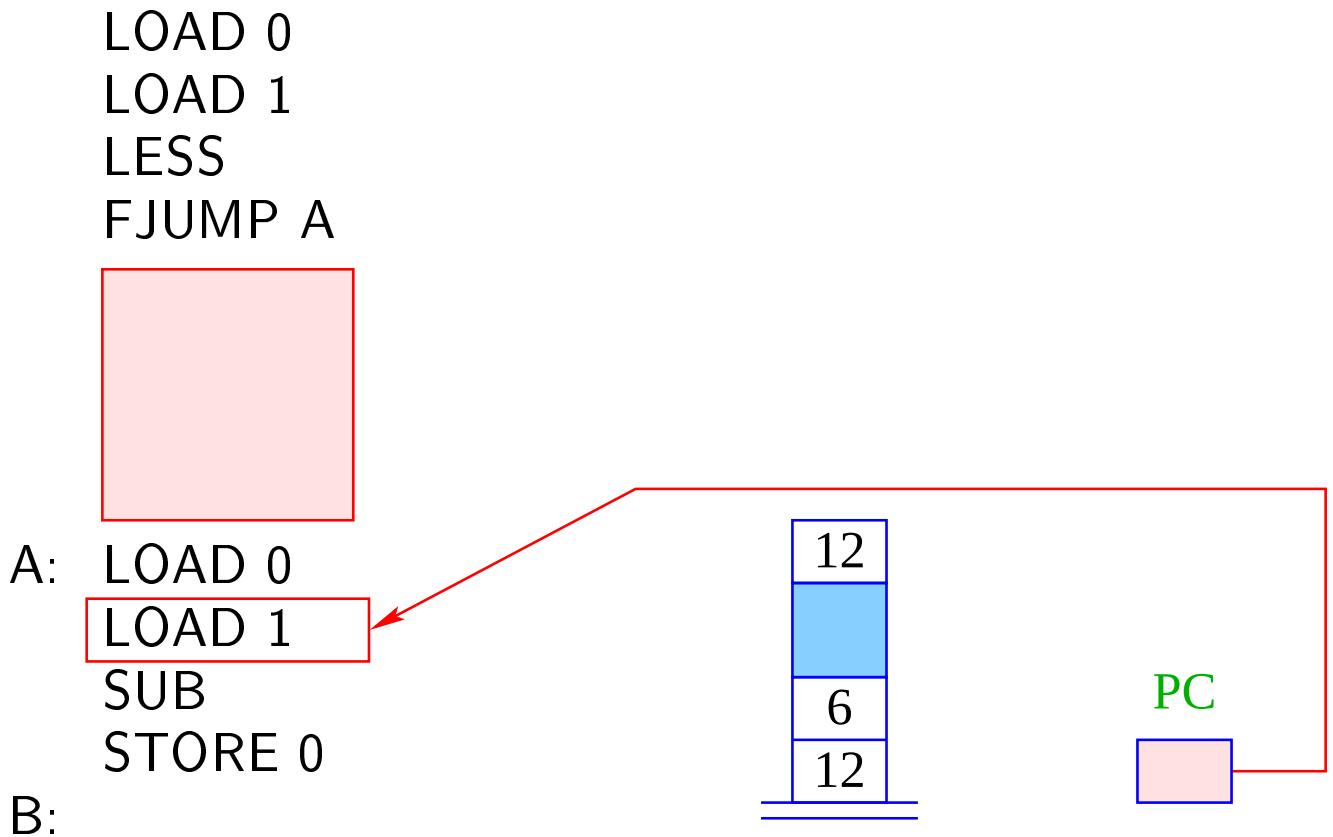

LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
12
PC

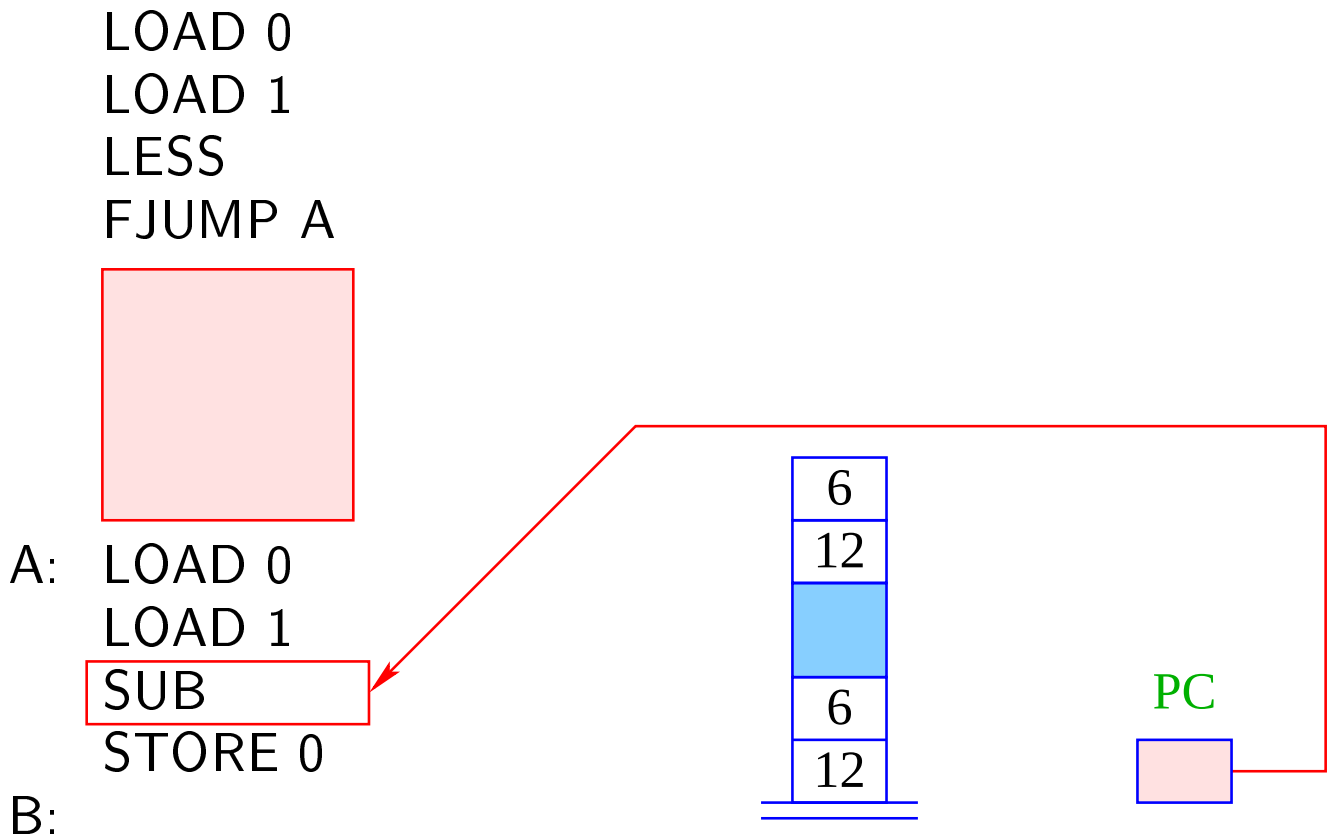

LOAD 0
LOAD 1
LESS
FJUMP A
A: LOAD 0
LOAD 1
SUB
STORE 0
B:
12
6
12
PC

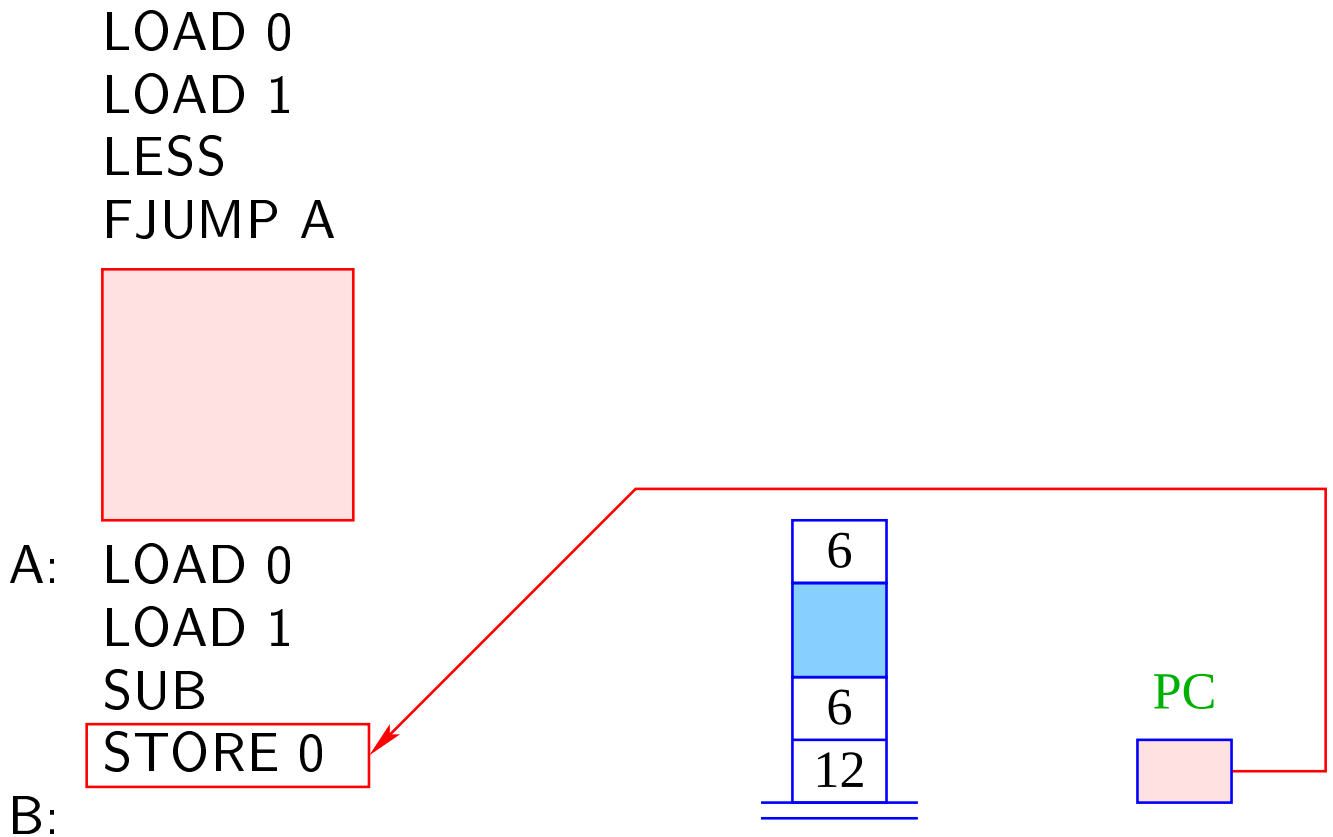

LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
6
12
6
12
PC

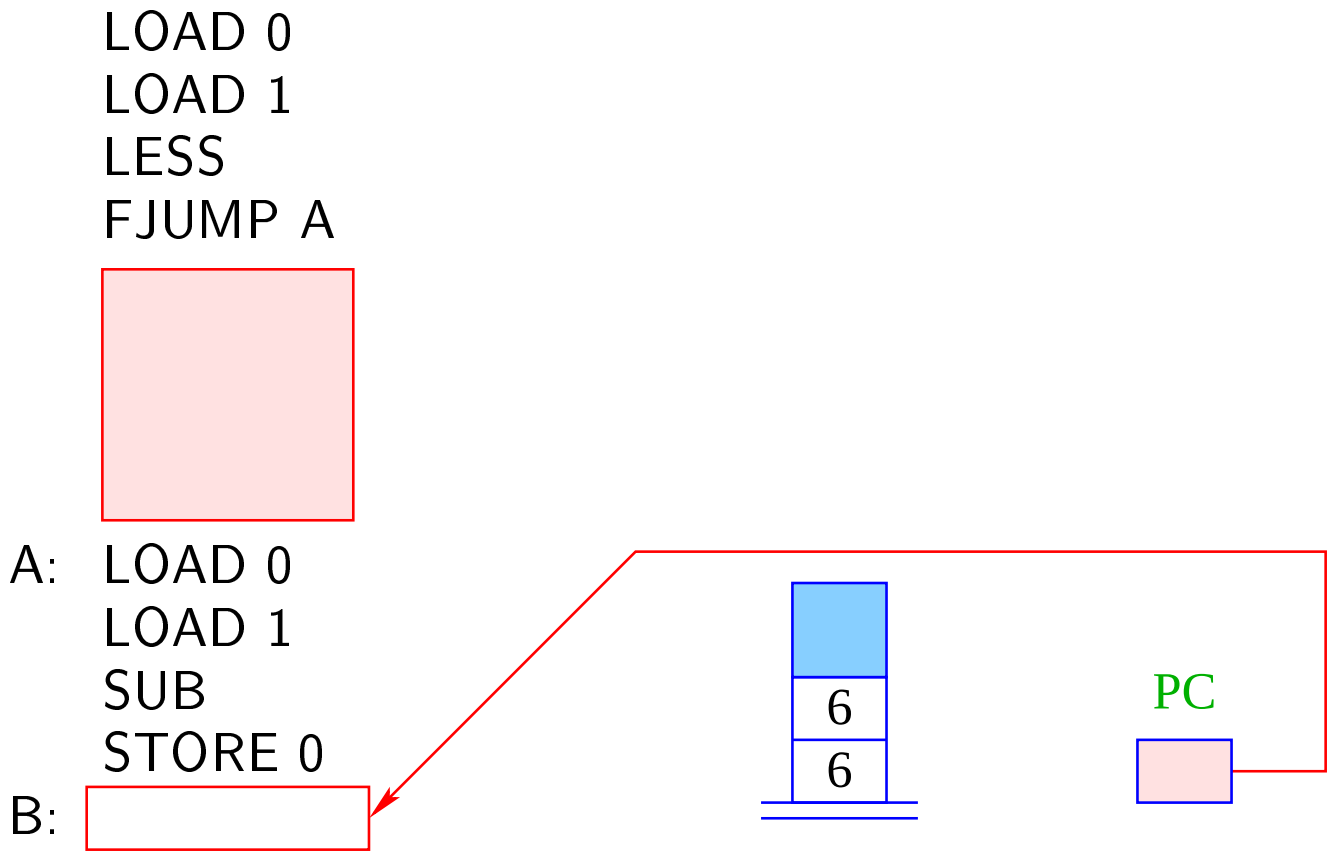

LOAD 0
LOAD 1
LESS
FJUMP A
A:
LOAD 0
LOAD 1
SUB
STORE 0
B:
ff
6
12
PC


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
    LESS
    FJUMP B
                    LOAD 0
                    CONST 1
                    SUB
                    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.