Von MiniJava zur MiniJVM

Helmut Seidl

Abteilung Informatik

Universität Trier

1 Eine einfache Programmiersprache

Eine Programmiersprache bietet

- Datenstrukturen;
- Operationen auf Daten;
- Kontrollstrukturen zur Ablaufsteuerung.

Als Beispiel betrachten wir die sehr einfache Programmiersprache MiniJava.

Beispiel: Das GGT-Programm ...

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

MiniJava-Programme enthalten Zuweisungen, bedingte Verzeigungen, Schleifen und (rudimentäre) IO.

MiniJava-Programme enthalten keinerlei ...

- Felder, Strings und Objekte;
- Methoden-Aufrufe und Prozeduren;
- Threads, Exceptions und GUI-Elemente;
- Reflection, ...

Aber Achtung:

- Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich sogar mit einem MiniJava-Programm berechnen !!!

- MiniJava-Programme sind ausführbares Java ... Man muss sie nur geeignet dekorieren :-)

Aus dem GGT-Programm wird das Java-Programm:

```
import java.io.*;
public class GGT {
   public static void write(int i) {
       System.out.println(i);
       System.out.flush();
   }
   BufferedReader stdin = new BufferedReader (
                          new InputStreamReader (System.in));
   public static int read() throws IOException {
      return Integer.parseInt(stdin.readLine());
   }
   public static void main (String[] args) throws IOException {
```

das MiniJava-Programm

```
   }  // Ende der Definition von main();
}   // Ende der Definition der Klasse GGT;
```

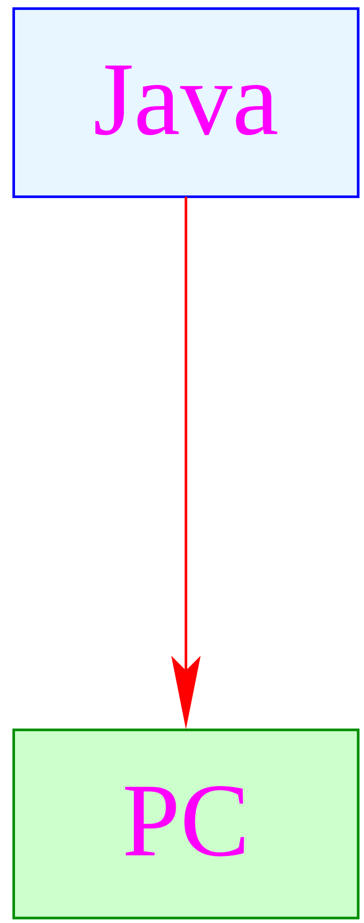
Solche Programme wollen wir auf einem realen Rechner ausführen.

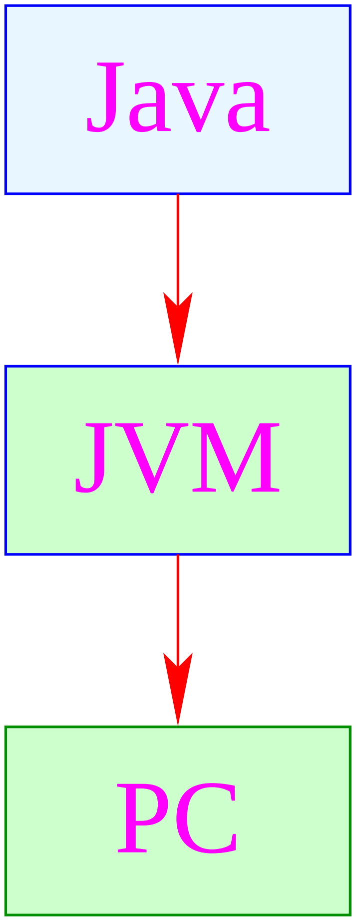
Problem 1:

- Hardware-Operationen sind sehr primitiv
- ... und von Prozessor zu Prozessor verschieden.

Idee:

- Übersetze nicht in eine Folge von Instruktionen eines realen Rechners, sondern einer selbst definierten abstrakten Maschine!
- Die abstrakte Maschine sollte leicht implementierbar sein.
- Die Übersetzung von MiniJava-Programmen sollte nicht zu schwierig sein ...


Java
PC


Java
JVM
PC

Problem 2:

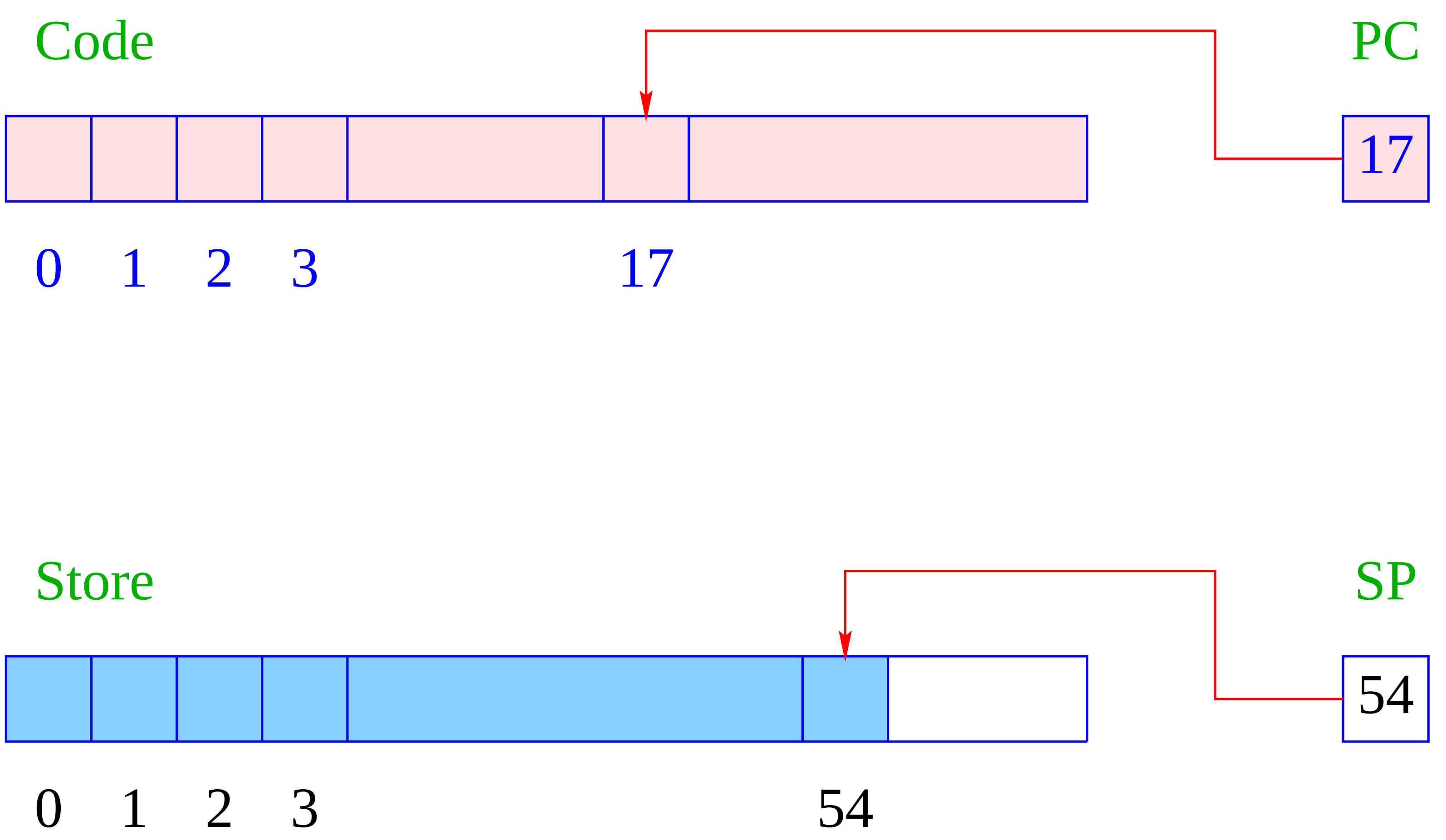
- Die Code-Erzeugung muss für sämtliche MiniJava-Programme funktionieren.
- Im Gegensatz zu Maschinen-Instruktionen können MiniJava-Programme beliebig komplex sein :-(

Idee:

- Gib formale Regeln an, wie MiniJava-Programme aus einfacheren Bestandteilen aufgebaut sind $\Longrightarrow$ kontextfreie Grammatiken
- Definiere die Übersetzungs-Funktion rekursiv über den Programm-Aufbau !!!

2 Von MiniJava zur JVM

Architektur der JVM:

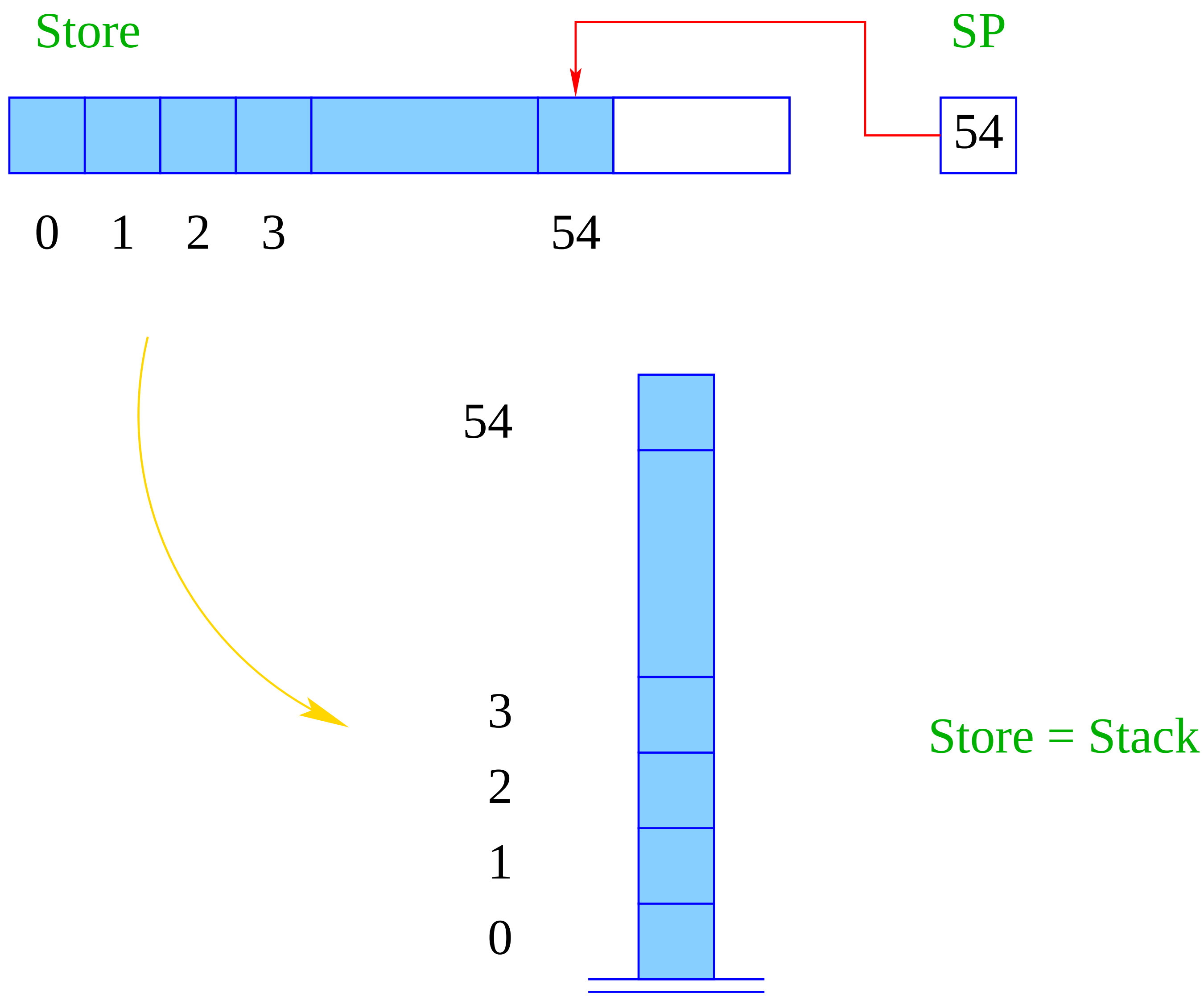


Code	=	enthält JVM-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter –
		zeigt auf nächsten auszuführenden Befehl;
Store	=	Speicher für Daten;
		jede Zelle kann einen Wert aufnehmen;
SP	=	Stack-Pointer –
		zeigt auf oberste belegte Zelle.

Achtung:

- Programm wie Daten liegen im Speicher – aber in verschiedenen Abschnitten.
- Programm-Ausführung holt nacheinander Befehle aus Code und führt die entsprechenden Operationen auf Store aus.

Konvention:


Store
SP
54
0 1 2 3 54
54
3
2
1
0
Store = Stack

Befehle der JVM:

`int`-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
`boolean`-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendung des Programms:	HALT

Ein Beispiel-Programm:

```
    ALLOC 2
    READ
    STORE 0
    READ
    STORE 1
A:  LOAD 0
    LOAD 1
    NEQ
    FJUMP D
    LOAD 0
    LOAD 1
    LESS
    FJUMP B
    LOAD 1
    LOAD 0
    SUB
    STORE 1
    JUMP C
B:  LOAD 0
    LOAD 1
    SUB
    STORE 0
C:  JUMP A
D:  LOAD 1
    WRITE
    HALT
```

- Das Programm berechnet den GGT :-)
- Die Marken (Labels) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

$$\begin{array}{rcr} \mathsf{A} & = & 5 \\ \mathsf{B} & = & 18 \\ \mathsf{C} & = & 22 \\ \mathsf{D} & = & 23 \end{array}$$

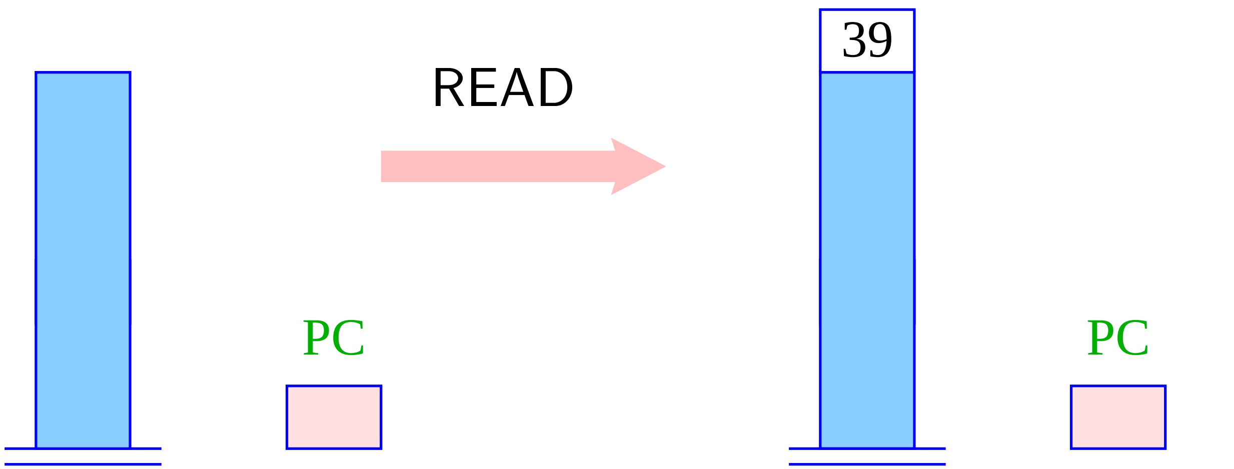
- ... können vom Compiler “leicht” in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden).

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

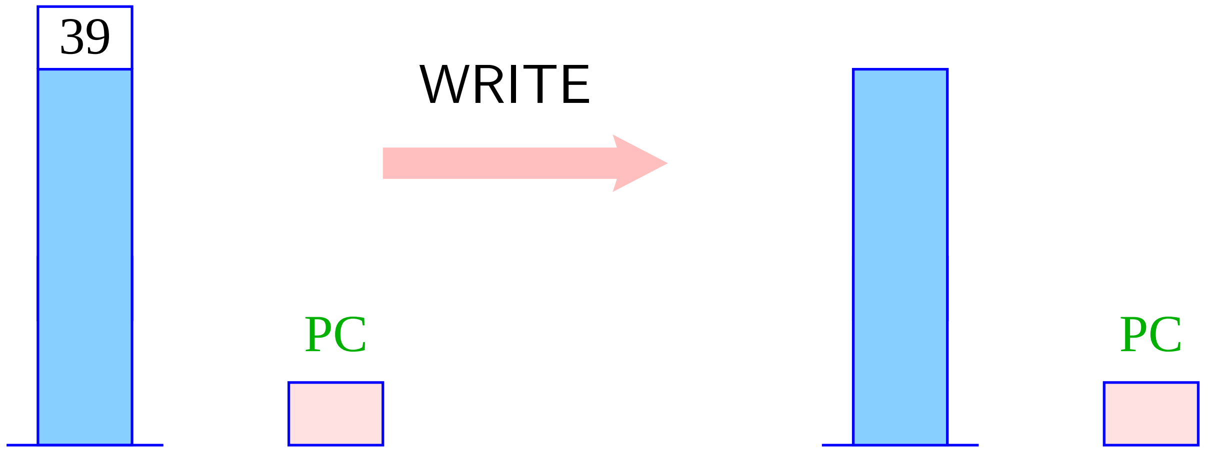
Idee:

- 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.



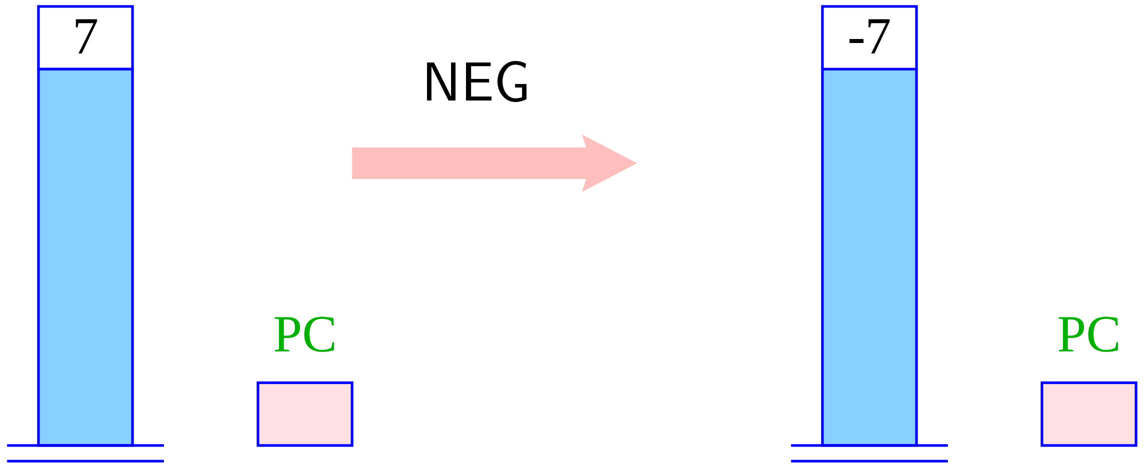
... falls 39 eingegeben wurde

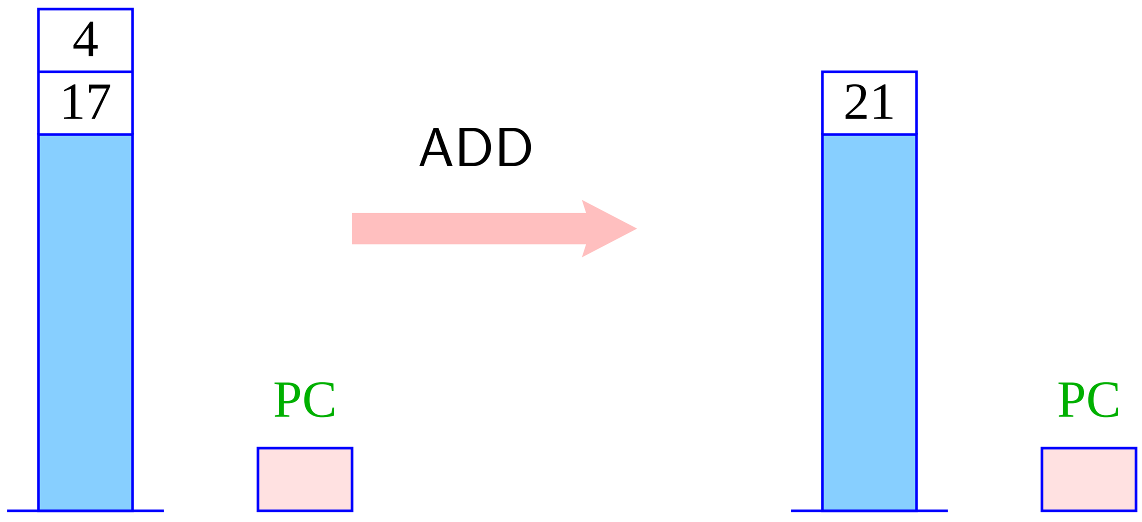


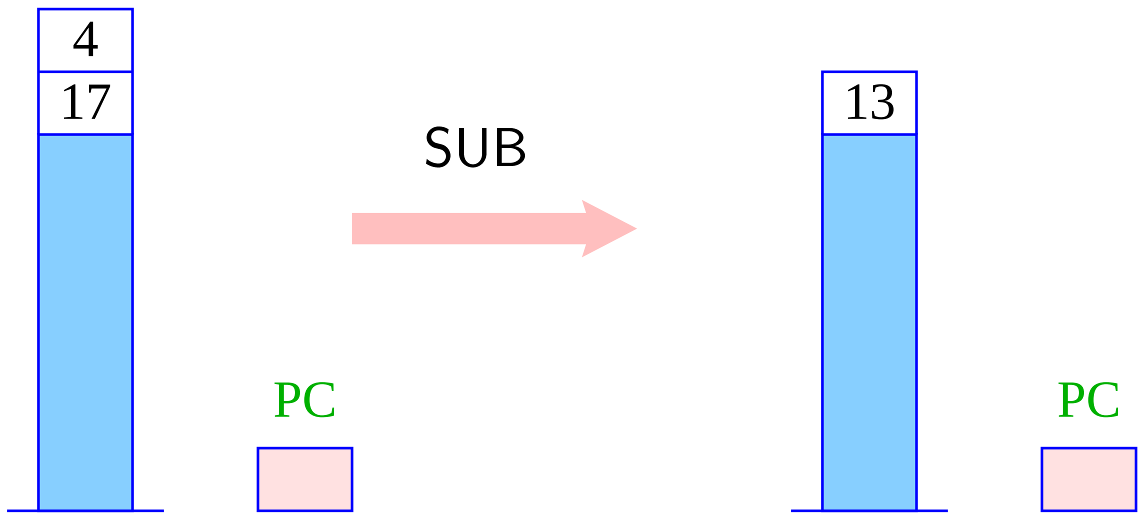
... wobei 39 ausgegeben wird

Arithmetik

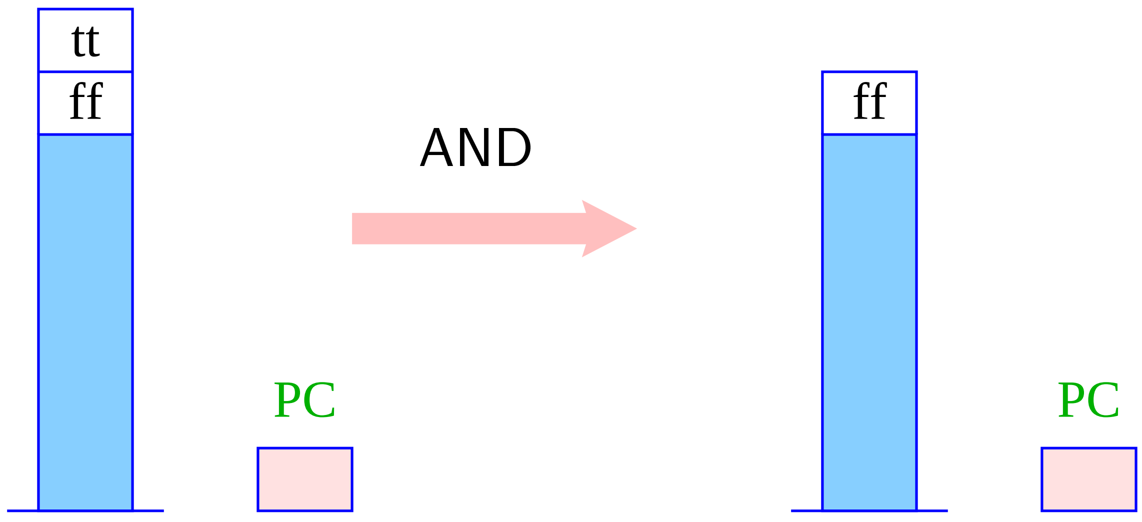
- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.

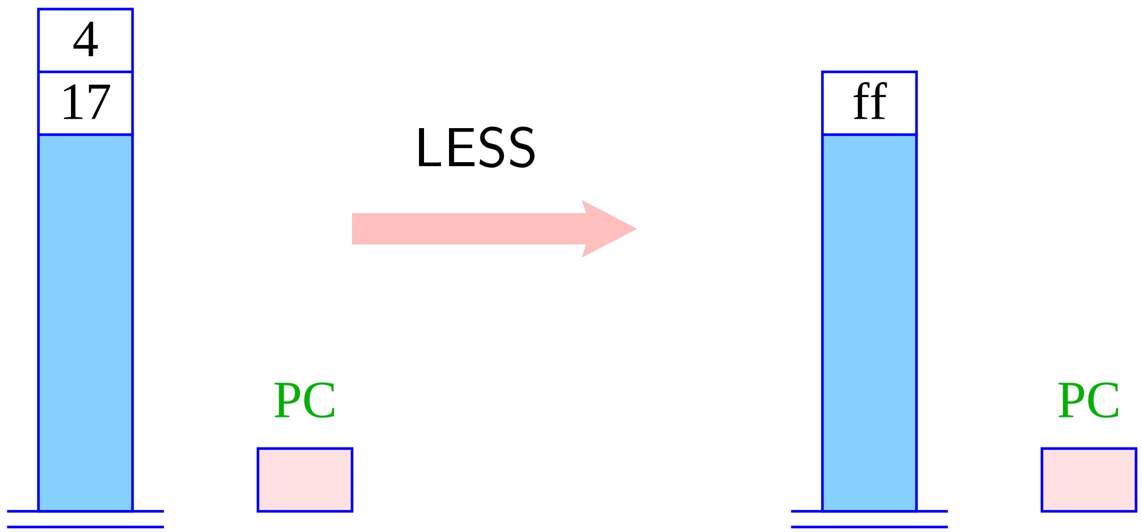

7
NEG
-7
PC
PC


4
17
ADD
21
PC
PC


4
17
SUB
PC
13
PC

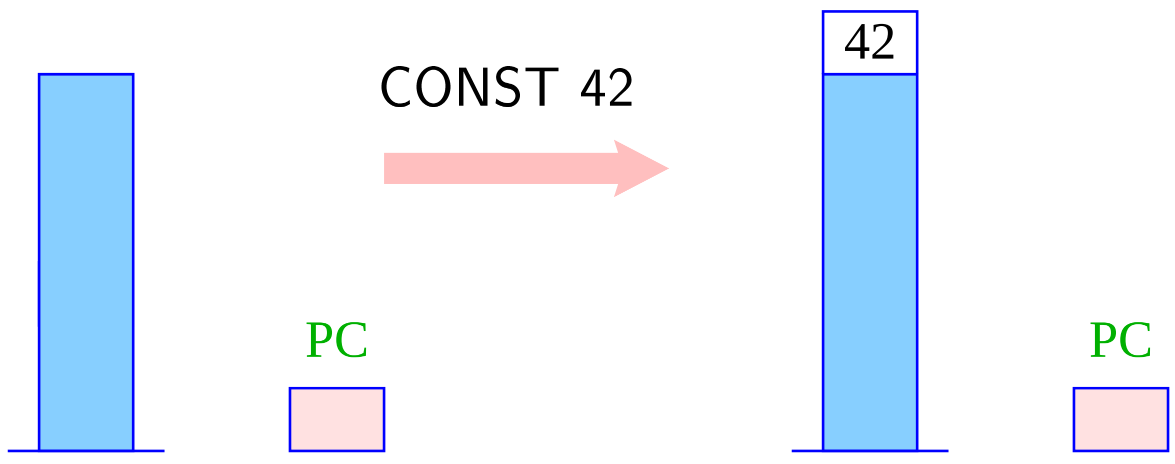
- Die übrigen arithmetischen Operationen MUL, DIV, MOD funktionieren völlig analog.
- Die logischen Operationen NEG, AND, OR ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von `true` und `false` arbeiten (hier: “tt” und “ff”).
- Auch die Vergleiche arbeiten so – nur konsumieren sie ganze Zahlen und liefern einen logischen Wert.

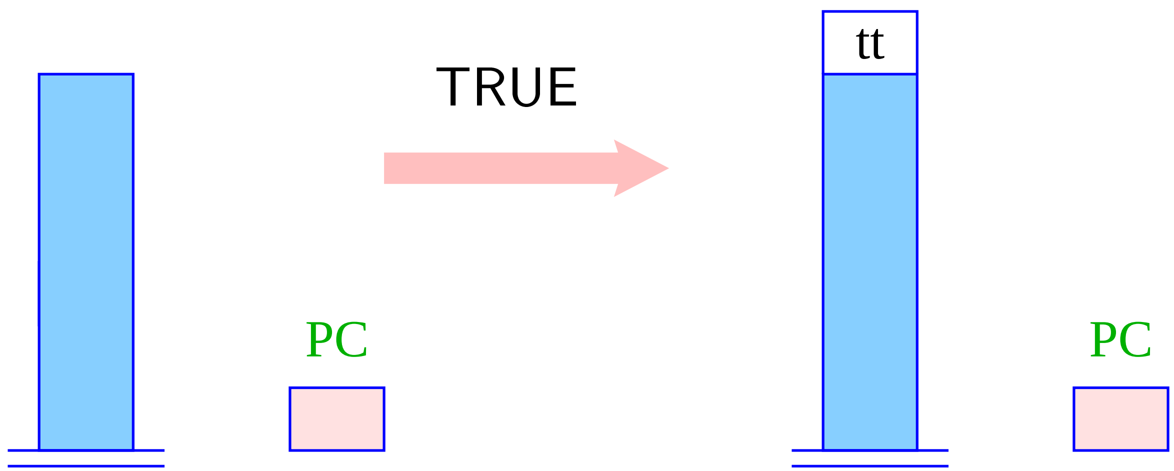

tt
ff
AND
PC
ff
PC

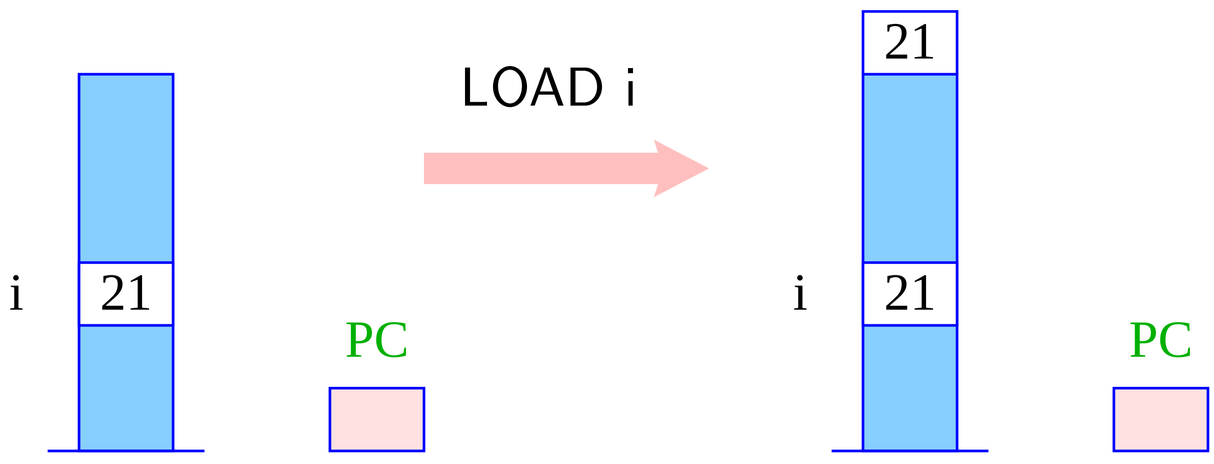

4
17
LESS
PC
ff
PC

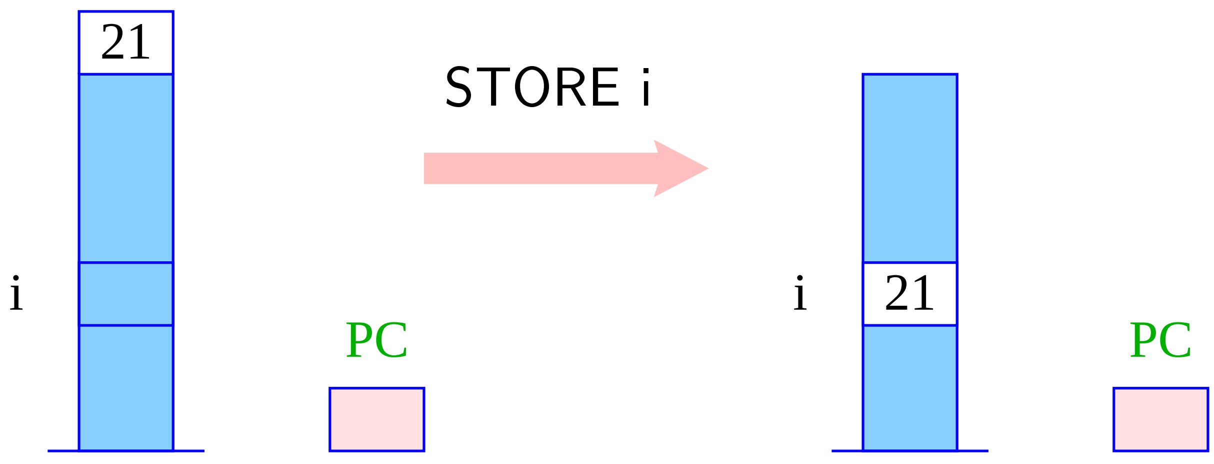
Laden und Speichern

- 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

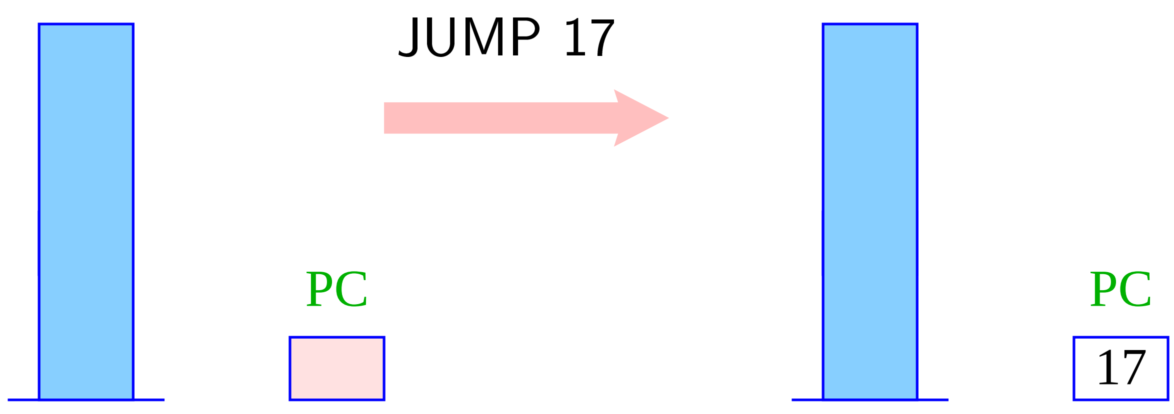

TRUE
tt
PC
PC

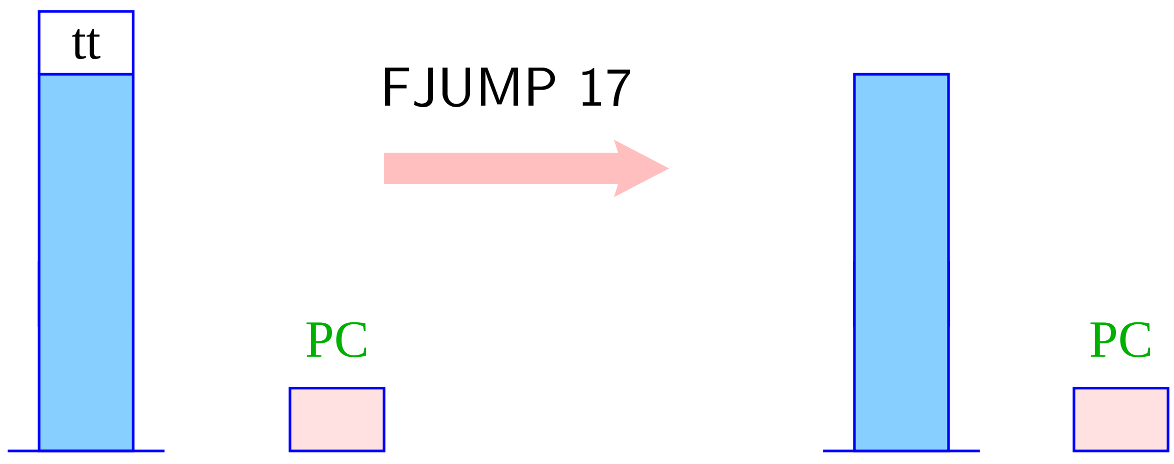

i
21
PC
LOAD i
21
i
21
PC

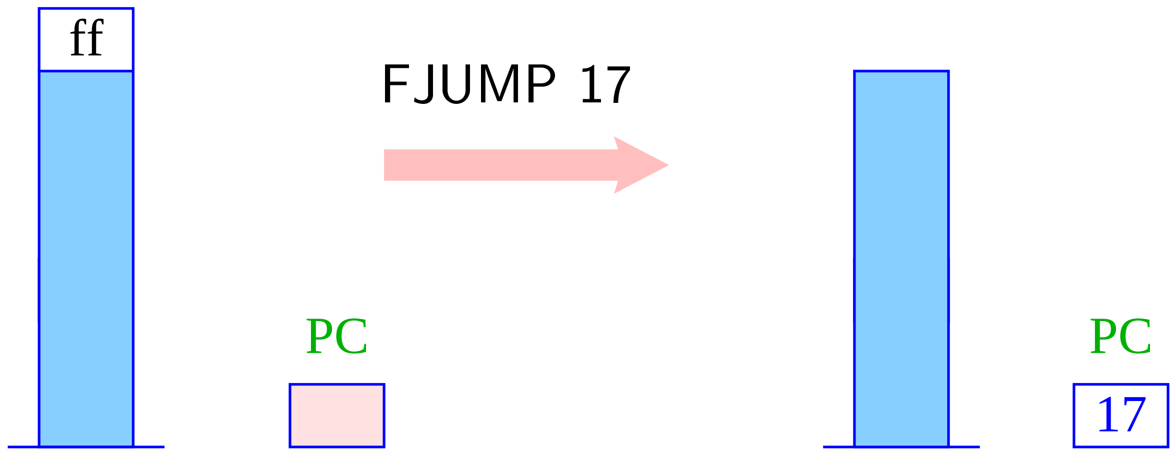

21
STORE i
i
PC
i
21
PC

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.

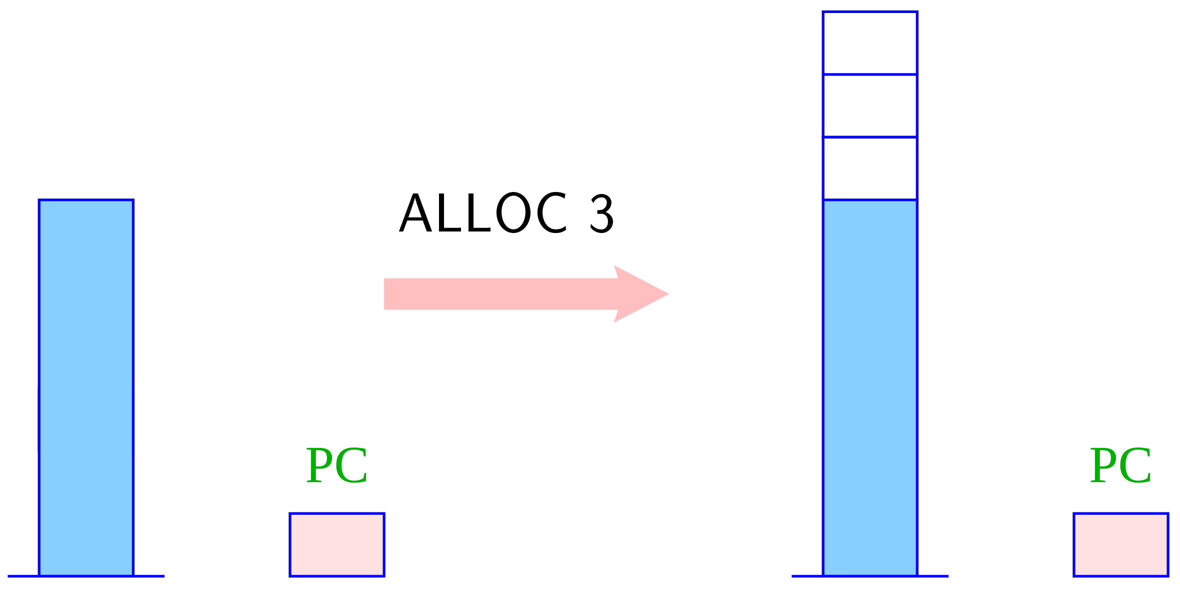

JUMP 17
PC
PC
17


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:

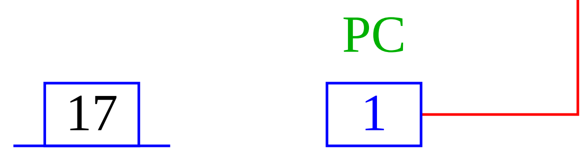
```
CONST 17
CONST 4
ADD
CONST 2
MUL
WRITE
HALT
```

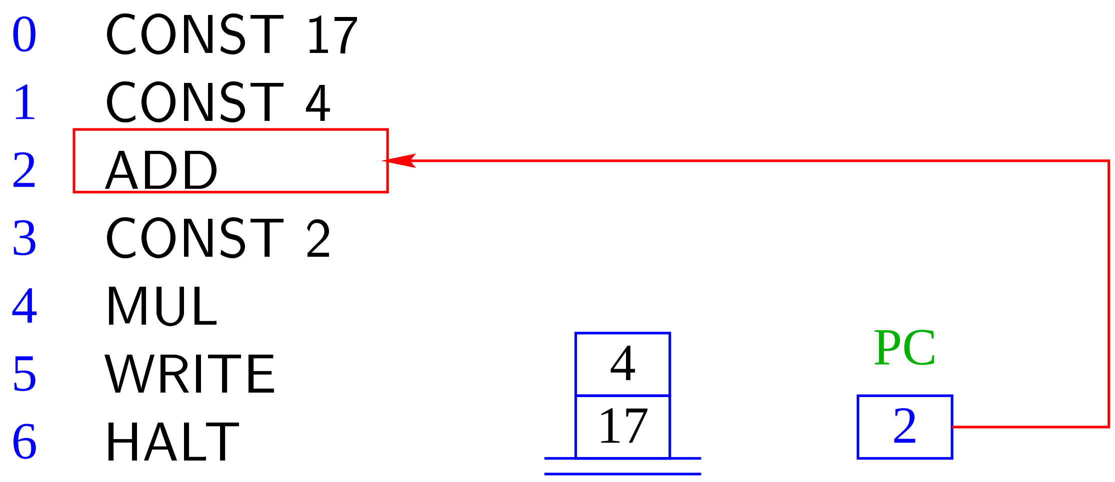
0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT

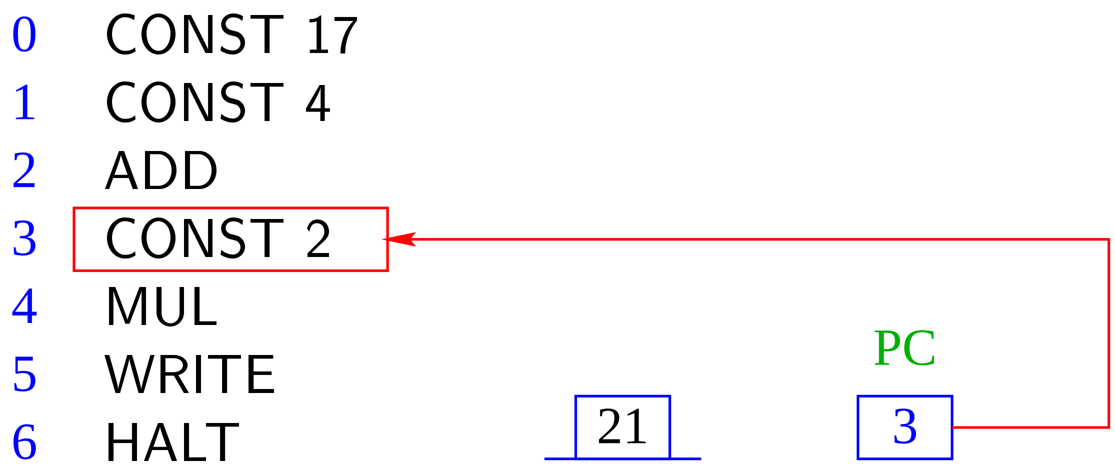
PC

0

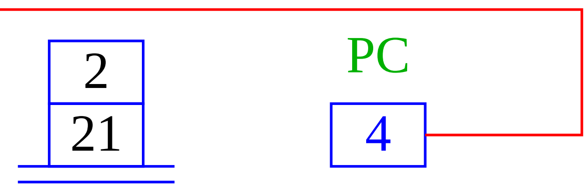
0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT



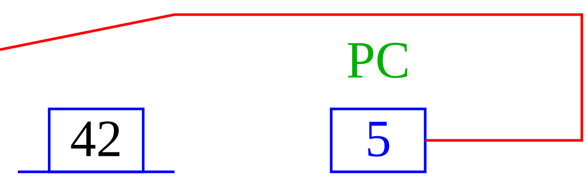

0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
4
17
PC
2


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

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



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



```
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.

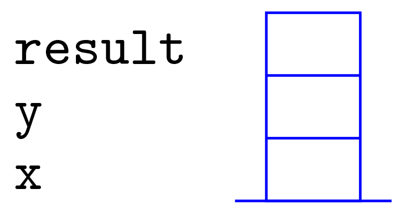
2.1 Übersetzung von Deklarationen

Betrachte Deklaration

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

Idee:

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



$\Longrightarrow$

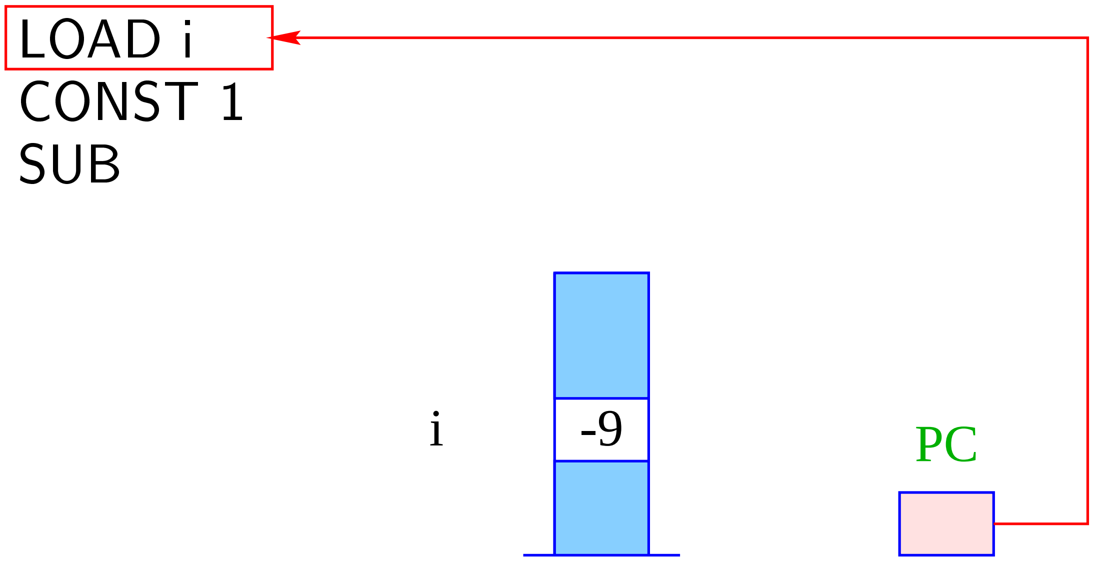
Übersetzung von `int` $x_0, \ldots, x_{n-1}$; = ALLOC n

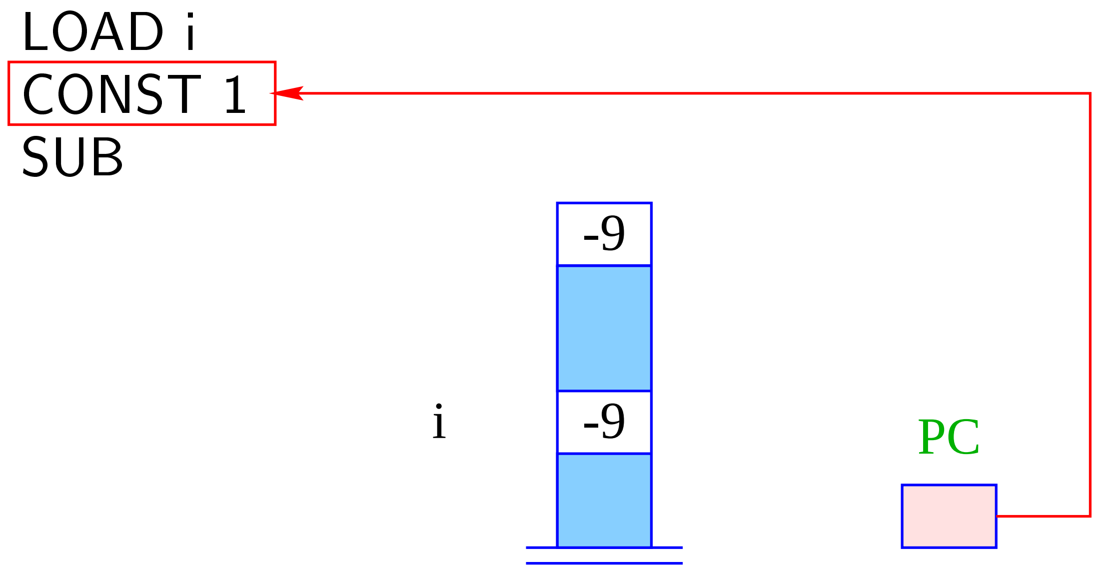
2.2 Übersetzung von Ausdrücken

Idee:

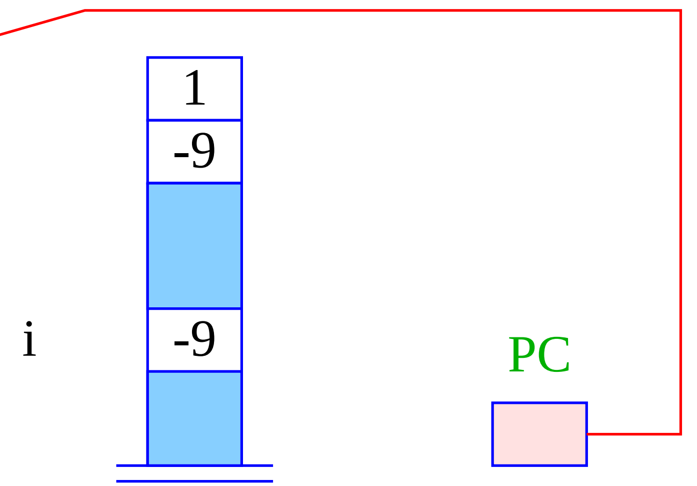
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

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

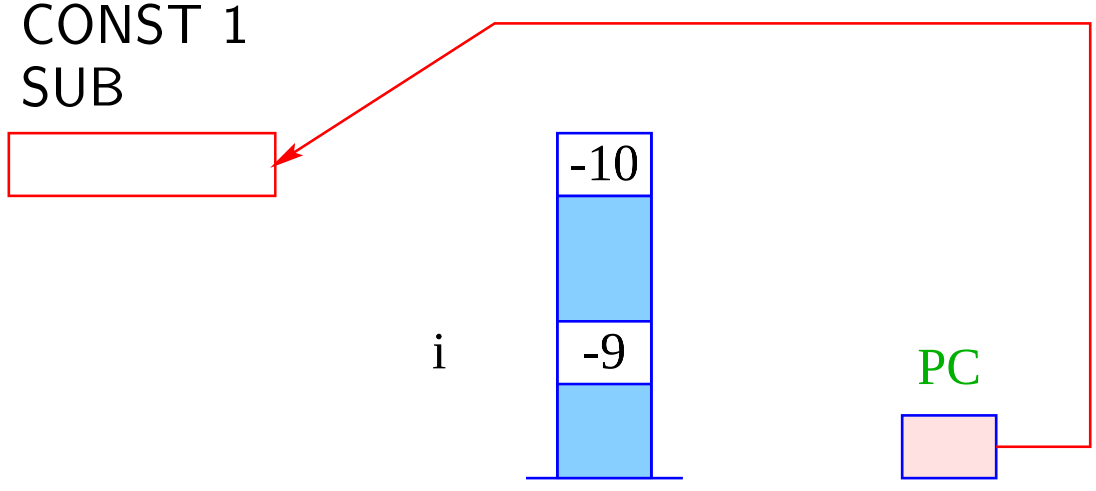

LOAD i
CONST 1
SUB
i
-9
PC


LOAD i
CONST 1
SUB
-9
i
-9
PC

LOAD i
CONST 1
$\boxed{\text{SUB}}$



```
LOAD i
CONST 1
SUB
```



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	LOAD 1	MUL
CONST 7	CONST 14	
ADD	SUB	

2.3 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!

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

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

Beispiel:

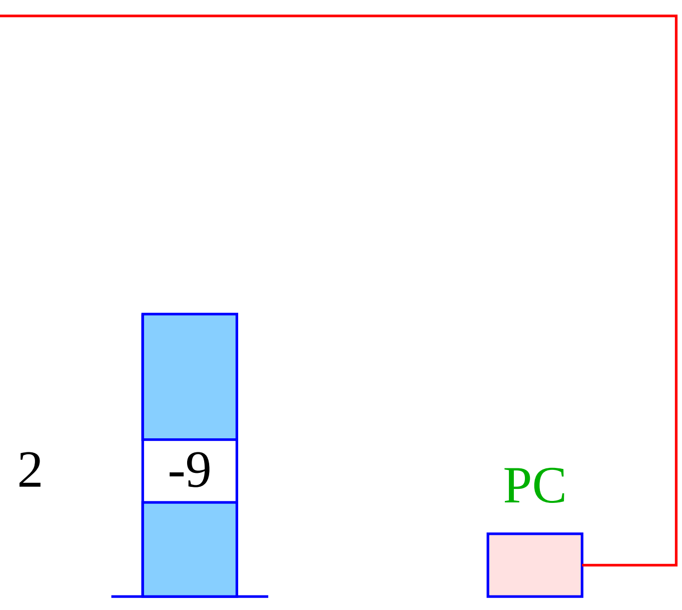
Für

```
x = x + 1;
```

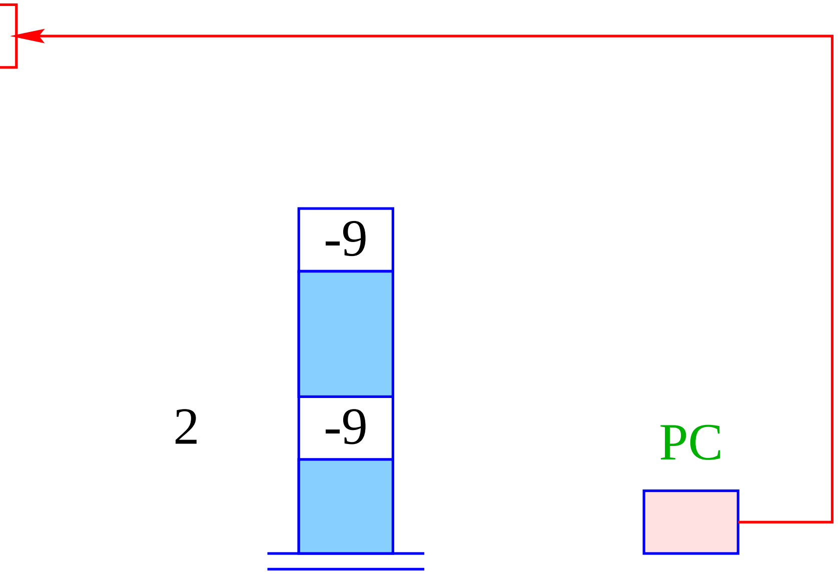
(x die 2. Variable) liefert das:

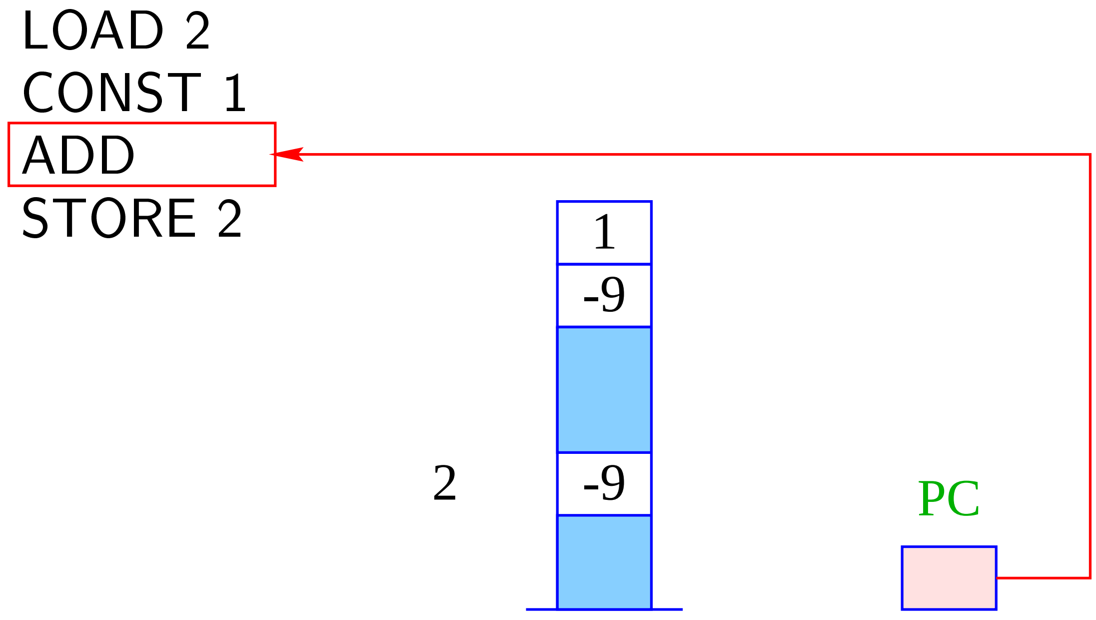
LOAD 2
CONST 1
ADD
STORE 2

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

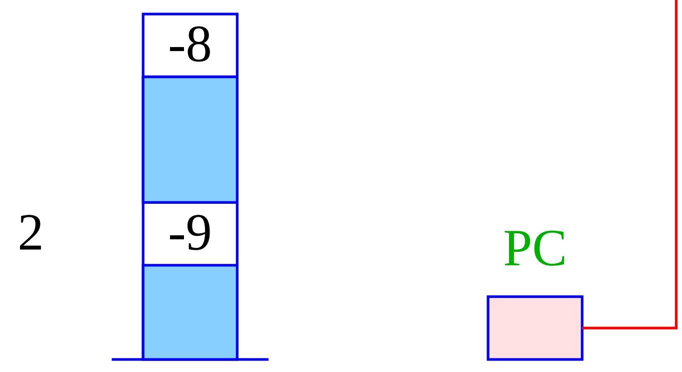


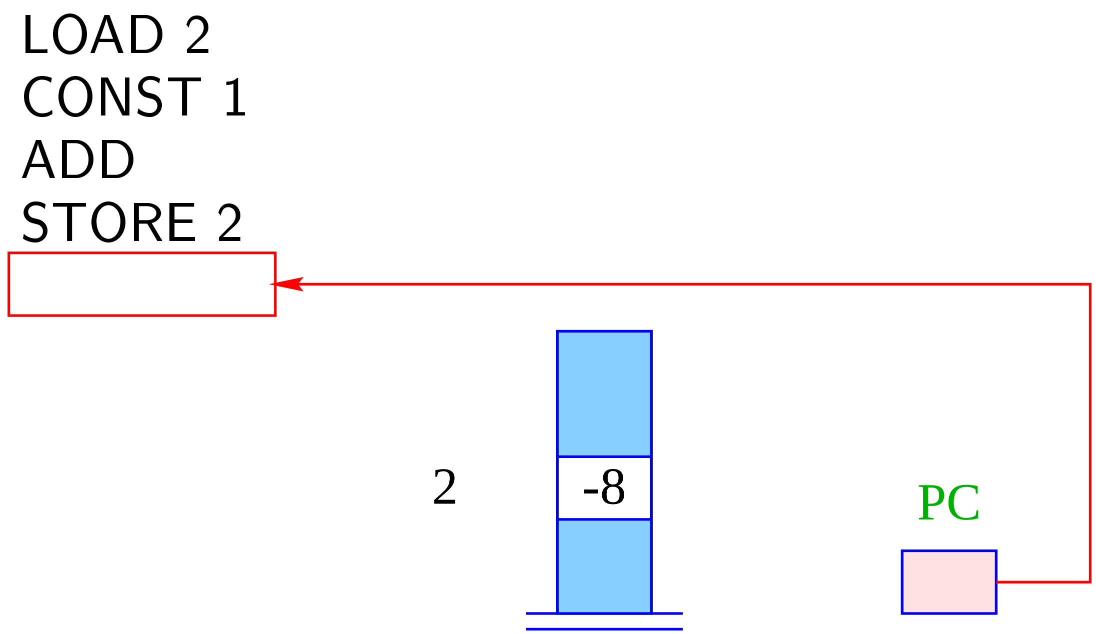
```
LOAD 2
CONST 1
ADD
STORE 2
```




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

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




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

2.4 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:	...	

- Falls die Bedingung sich zu `false` evaluiert, wird der `then`-Teil übersprungen (mithilfe von FJUMP A).
- Nach Abarbeitung des `then`-Teils muss hinter dem gesamten `if`-Statement fortgefahren werden. Dazu dient JUMP B.

Beispiel:

Für das Statement

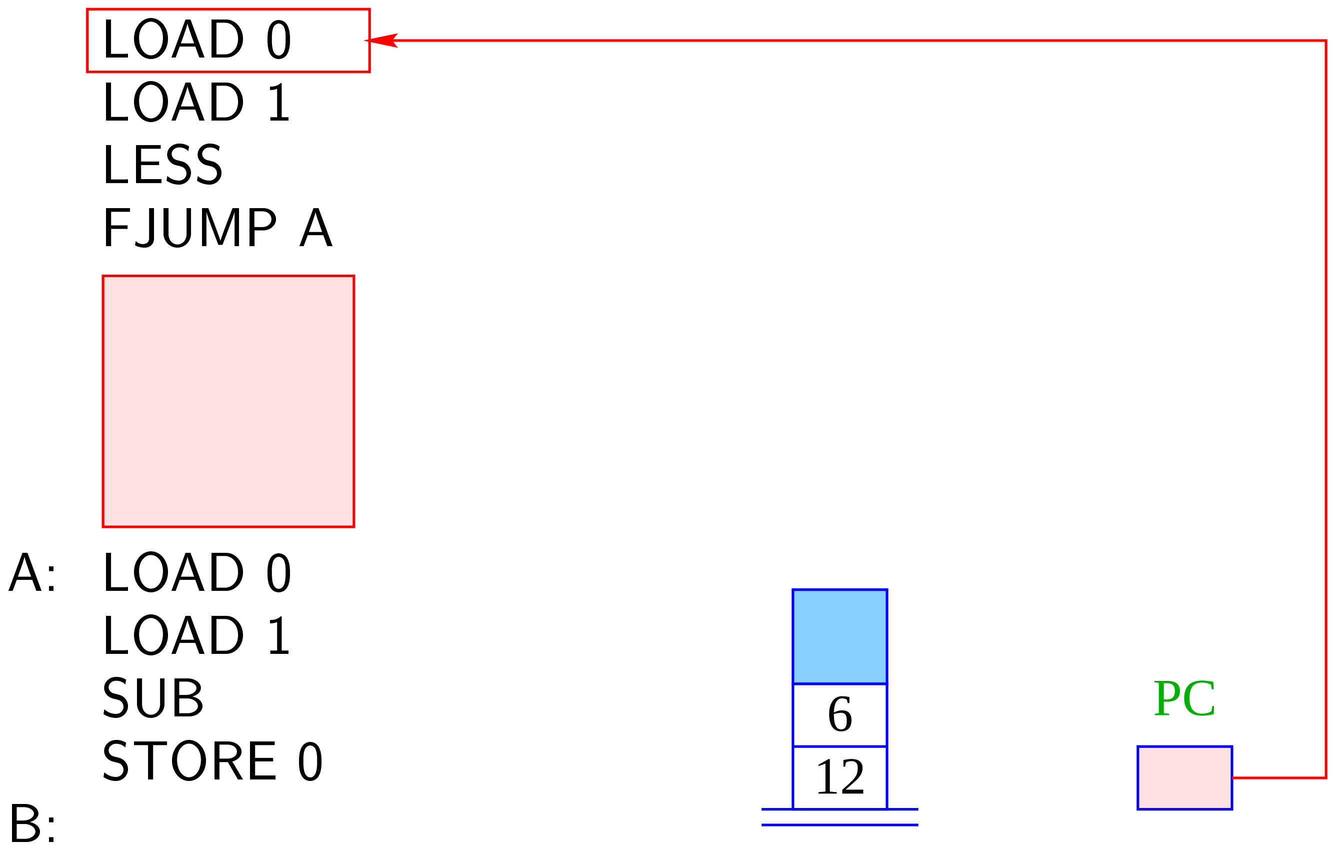
```
if (x < y)
    y = y - x;
else
    x = x - y;
```

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

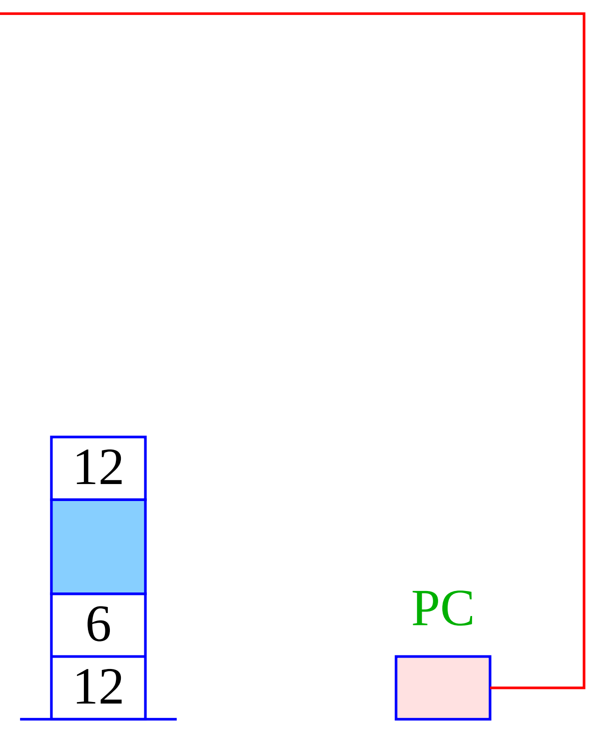
```
LOAD 0
LOAD 1
LESS
FJUMP A
```

```
LOAD 1
LOAD 0
SUB
STORE 1
JUMP B
```

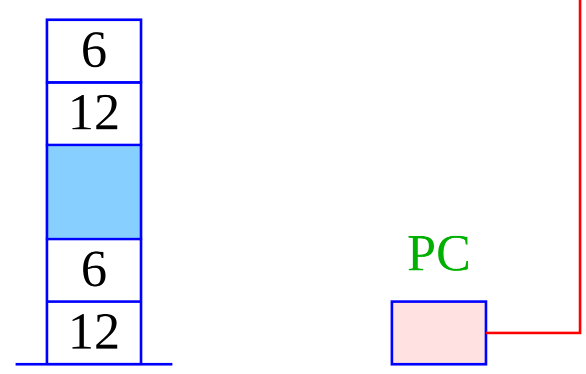
```
A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:  ...
```


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

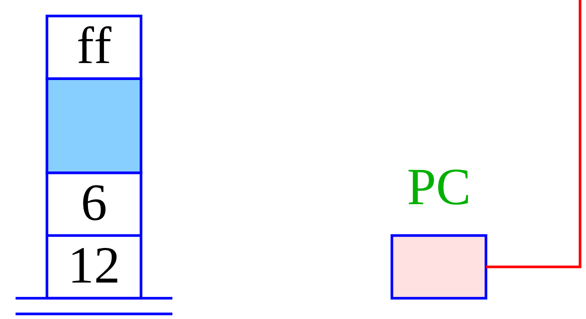
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



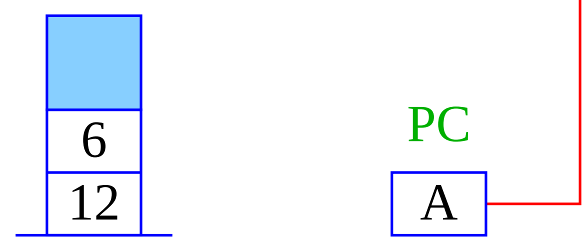
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



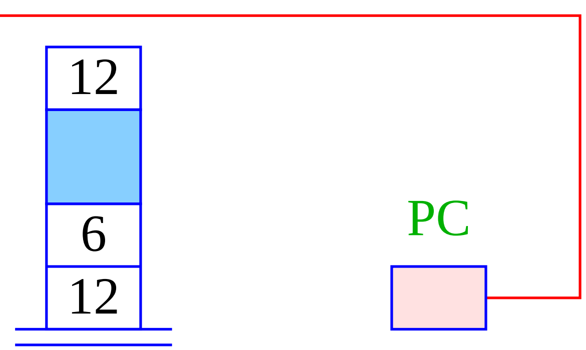
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



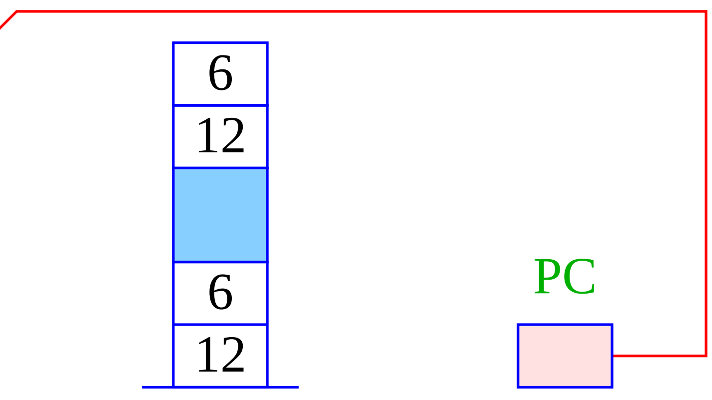
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



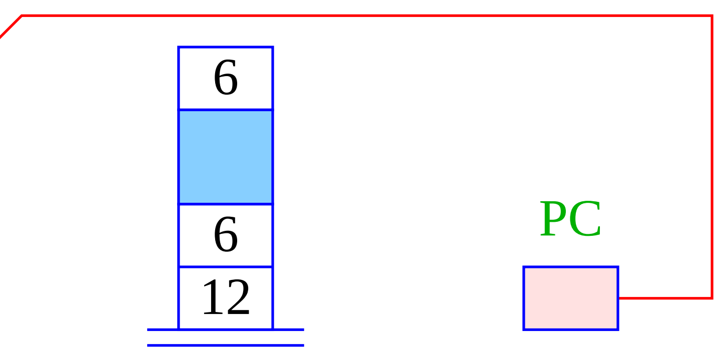
```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



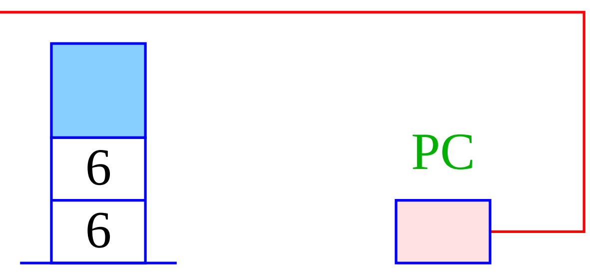
```
      LOAD 0
      LOAD 1
      LESS
      FJUMP A

A:    LOAD 0
      LOAD 1
      SUB
      STORE 0
B:
```



```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



```
    LOAD 0
    LOAD 1
    LESS
    FJUMP A

A:  LOAD 0
    LOAD 1
    SUB
    STORE 0
B:
```



2.5 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:	...	

- Falls die Bedingung sich zu `false` evaluiert, wird die Schleife verlassen (mithilfe von FJUMP B).
- Nach Abarbeitung des Rumpfs muss auf jeden Fall 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:  ...
```

2.6 Übersetzung von Statement-Folgen

Idee:

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

Folglich:

Übersetzung von	stmt_1 ... 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

2.7 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 ...

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) :

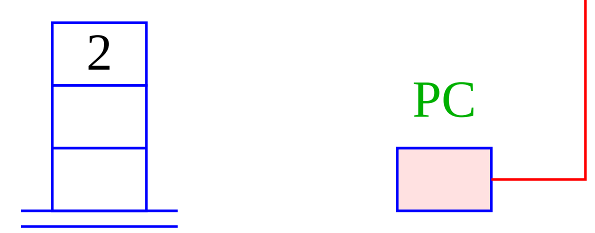
```
ALLOC 2          A:  CONST 1          LOAD 1
READ                 LOAD 0           LOAD 0
STORE 0              LESS             MUL
CONST 1              FJUMP B          STORE 1
STORE 1

          LOAD 0       B:  LOAD 1
          CONST 1          WRITE
          SUB              HALT
          STORE 0
          JUMP A
```

```
    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

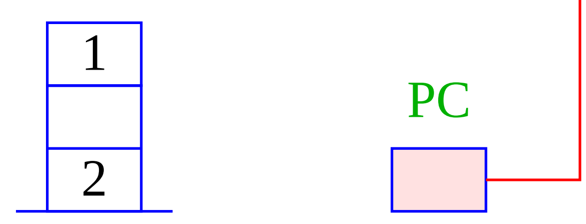
```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      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
```



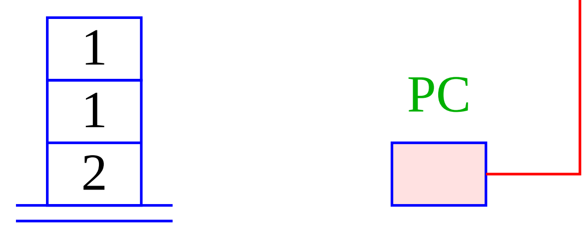
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    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
```

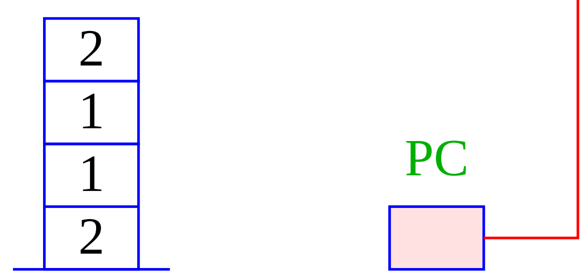


```
      ALLOC 2
      READ
      STORE 0
      CONST 1
      STORE 1
A:    CONST 1
      LOAD 0
      LESS
      FJUMP B

      JUMP A
B:    LOAD 1
      WRITE
      HALT
```

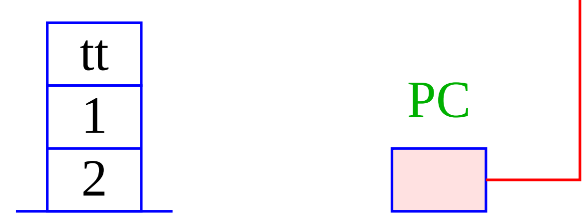
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



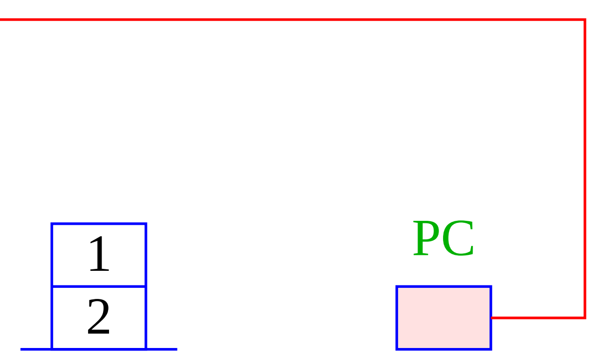
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    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
```



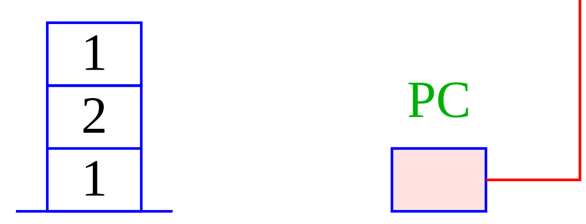
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    JUMP A
B:  LOAD 1
    WRITE
    HALT
```



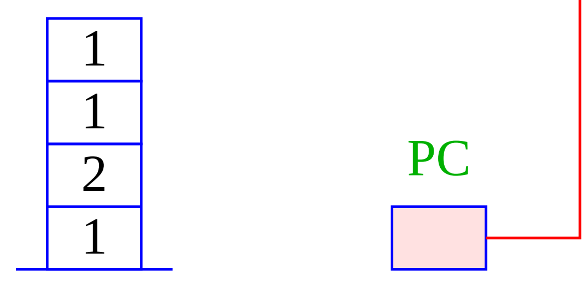
```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    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
```

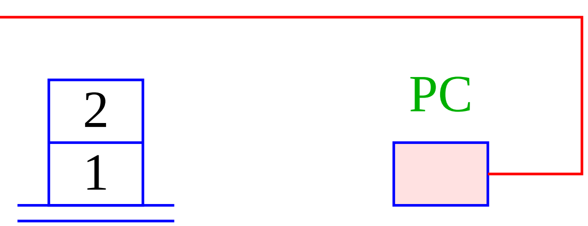


```
    ALLOC 2
    READ
    STORE 0
    CONST 1
    STORE 1
A:  CONST 1
    LOAD 0
    LESS
    FJUMP B

    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
```

```
     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.

$\Longrightarrow$ Compilerbau