

**Beispiel:** Code für  $e \equiv x = y - 1$  mit  $\rho = \{x \mapsto 4, y \mapsto 7\}$ .  
Dann liefert  $\text{code}_R e \rho$ :

loadc 7
load

loadc 1  
sub

loadc 4
store

## Optimierungen:

Einführung von Spezialbefehlen für häufige Befehlsfolgen, hier etwa:

loada q = loadc q  
load

storea q = loadc q  
store

### 3 Anweisungen und Anweisungsfolgen

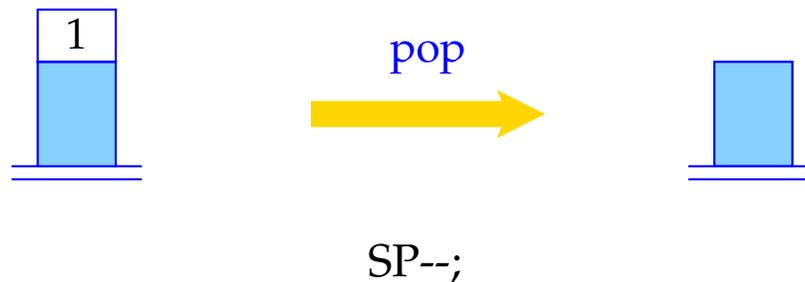
Ist  $e$  ein Ausdruck, dann ist  $e;$  eine Anweisung (Statement).

Anweisungen liefern keinen Wert zurück. Folglich muss der **SP** vor und nach der Ausführung des erzeugten Codes gleich sein:

$$\text{code } e; \rho = \text{code}_R e \rho$$

pop

Die Instruktion **pop** wirft das oberste Element des Kellers weg ...

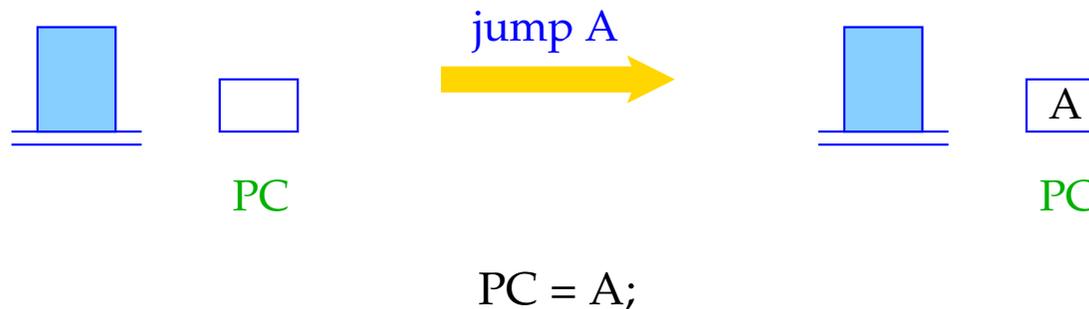


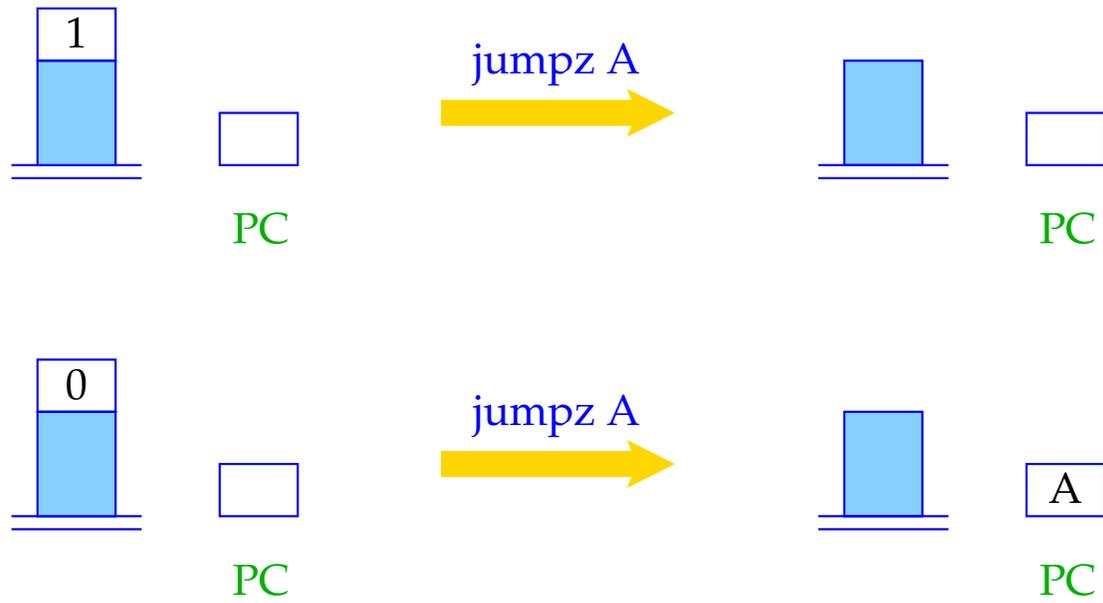
Der Code für eine Statement-Folge ist die Konkatenation des Codes for die einzelnen Statements in der Folge:

$$\begin{aligned} \text{code } (s \text{ } ss) \rho &= \text{code } s \rho \\ &\quad \text{code } ss \rho \\ \text{code } \varepsilon \rho &= \quad // \text{ leere Folge von Befehlen} \end{aligned}$$

## 4 Bedingte und iterative Anweisungen

Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:





```
if (S[SP] == 0) PC = A;  
SP--;
```

Der Übersichtlichkeit halber gestatten wir die Verwendung von **symbolischen Sprungzielen**. In einem zweiten Pass können diese dann durch absolute Code-Adressen ersetzt werden.

Statt absoluter Code-Adressen könnte man auch **relative** Adressen benutzen, d. h. Sprungziele relativ zum aktuellen **PC** angeben.

### **Vorteile:**

- **kleinere Adressen** reichen aus;
- der Code wird **relokierbar**, d. h. kann im Speicher unverändert hin und her geschoben werden.

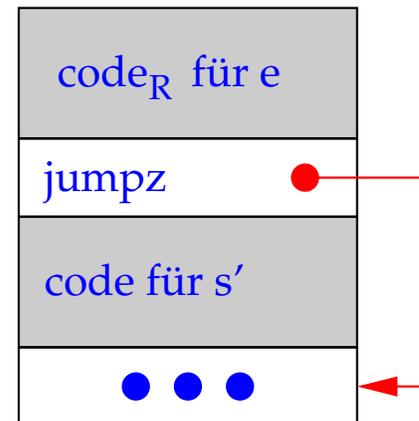
## 4.1 Bedingte Anweisung, einseitig

Betrachten wir zuerst  $s \equiv \mathbf{if} (e) s'$ .

Idee:

- Lege den Code zur Auswertung von  $e$  und  $s'$  hintereinander in den Code-Speicher;
- Dekoriere mit Sprung-Befehlen so, dass ein korrekter Kontroll-Fluss gewährleistet ist!

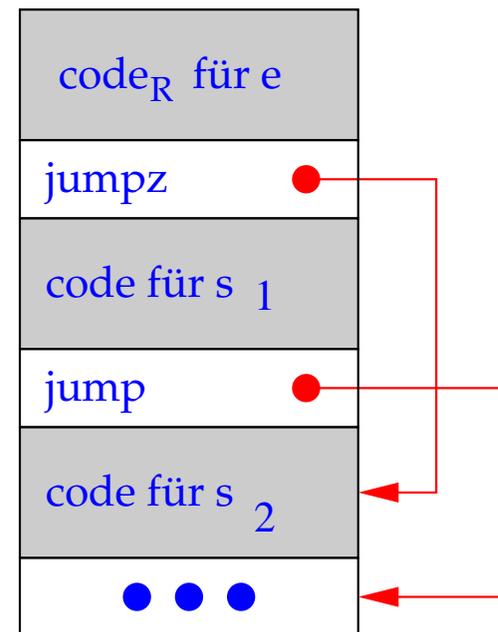
$\text{code } s \rho = \text{code}_R e \rho$   
 $\text{jumpz } A$   
 $\text{code } s' \rho$   
 $A : \dots$



## 4.2 Zweiseitiges if

Betrachte nun  $s \equiv \mathbf{if} (e) s_1 \mathbf{else} s_2$ . Die gleiche Strategie liefert:

$\mathbf{code} s \rho = \mathbf{code}_R e \rho$   
 $\mathbf{jumpz} A$   
 $\mathbf{code} s_1 \rho$   
 $\mathbf{jump} B$   
A :  $\mathbf{code} s_2 \rho$   
B : ...



Beispiel:

Sei  $\rho = \{x \mapsto 4, y \mapsto 7\}$  und

$s \equiv$  **if** ( $x > y$ ) (i)  
     $x = x - y;$  (ii)  
    **else**  $y = y - x;$  (iii)

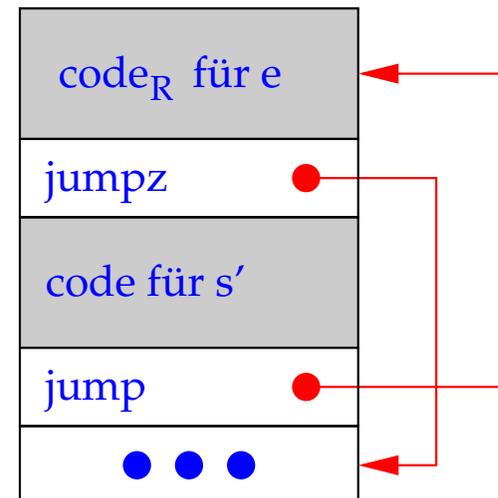
Dann liefert **code**  $s \rho$ :

loada 4	loada 4	A: loada 7
loada 7	loada 7	loada 4
gr	sub	sub
jumpz A	storea 4	storea 7
	pop	pop
	jump B	B: ...
<span style="color: red;">(i)</span>	<span style="color: red;">(ii)</span>	<span style="color: red;">(iii)</span>

### 4.3 while-Schleifen

Betrachte schließlich die Schleife  $s \equiv \mathbf{while} (e) s'$ . Dafür erzeugen wir:

$\mathit{code} s \rho =$   
A :  $\mathit{code}_R e \rho$   
     $\mathit{jumpz} B$   
     $\mathit{code} s' \rho$   
     $\mathit{jump} A$   
B : ...



Beispiel:

Sei  $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$  und  $s$  das Statement:

**while**  $(a > 0) \{c = c + 1; a = a - b; \}$

Dann liefert `s`  $\rho$  die Folge:

A:	loada 7	loada 9	loada 7	B: ...
	loadc 0	loadc 1	loada 8	
	gr	add	sub	
	jumpz B	storea 9	storea 7	
		pop	pop	
			jump A	

## 4.4 for-Schleifen

Die **for**-Schleife  $s \equiv \mathbf{for} (e_1; e_2; e_3) s'$  ist äquivalent zu der Statementfolge  $e_1; \mathbf{while} (e_2) \{s' e_3; \}$  – sofern  $s'$  keine **continue**-Anweisung enthält.

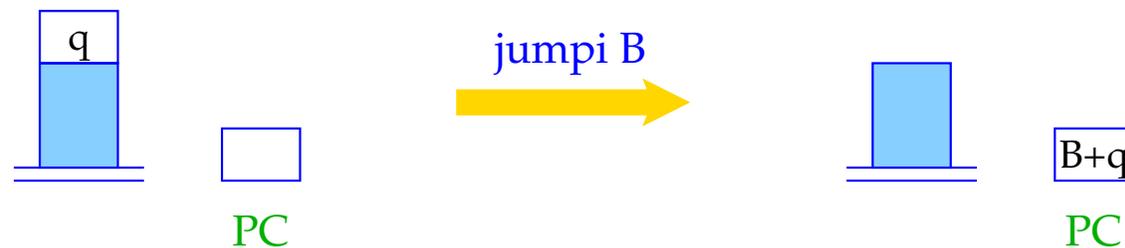
Darum übersetzen wir:

```
code s ρ = codeR e1
           pop
           A : codeR e2 ρ
               jumpz B
               code s' ρ
               codeR e3 ρ
               pop
               jump A
           B : ...
```

## 4.5 Das switch-Statement

### Idee:

- Unterstütze Mehrfachverzweigung in **konstanter Zeit!**
- Benutze **Sprungtabelle**, die an der  $i$ -ten Stelle den Sprung an den Anfang der  $i$ -ten Alternative enthält.
- Eine Möglichkeit zur Realisierung besteht in der Einführung von **indizierten Sprüngen**.



$PC = B + S[SP];$

$SP--;$

## Vereinfachung:

Wir betrachten nur **switch**-Statements der folgenden Form:

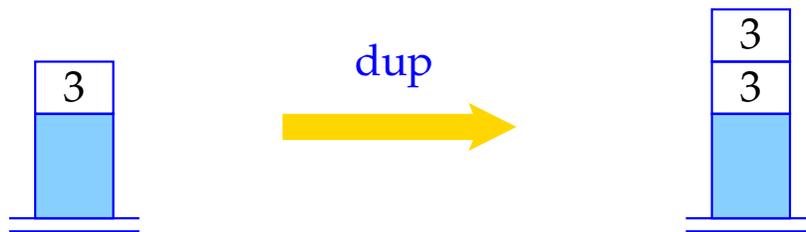
$$s \equiv \text{switch } (e) \{$$
$$\quad \text{case } 0: \quad ss_0 \text{ break;}$$
$$\quad \text{case } 1: \quad ss_1 \text{ break;}$$
$$\quad \quad \quad \vdots$$
$$\quad \text{case } k - 1: \quad ss_{k-1} \text{ break;}$$
$$\quad \text{default: } \quad ss_k$$
$$\quad \quad \quad \}$$

Dann ergibt sich für  $s$  die Instruktionsfolge:



<code>check 0 k B</code>	=	<code>dup</code>	<code>dup</code>	<code>jumpi B</code>
		<code>loadc 0</code>	<code>loadc k</code>	A: <code>pop</code>
		<code>geq</code>	<code>leq</code>	<code>loadc k</code>
		<code>jumpz A</code>	<code>jumpz A</code>	<code>jumpi B</code>

- Weil der R-Wert von  $e$  noch zur Indizierung benötigt wird, muss er vor jedem Vergleich kopiert werden.
- Dazu dient der Befehl `dup`.
- Ist der R-Wert von  $e$  kleiner als 0 oder größer als  $k$ , ersetzen wir ihn vor dem indizierten Sprung durch  $k$ .



```
S[SP+1] = S[SP];
```

```
SP++;
```

## Achtung:

- Die Sprung-Tabelle könnte genauso gut direkt hinter dem Macro `check` liegen. Dadurch spart man ein paar unbedingte Sprünge, muss aber evt. das `switch`-Statement zweimal durchsuchen.
- Beginnt die Tabelle mit  $u$  statt mit 0, müssen wir den R-Wert von  $e$  um  $u$  vermindern, bevor wir ihn als Index benutzen.
- Sind sämtliche möglichen Werte von  $e$  `sicher` im Intervall  $[0, k]$ , können wir auf `check` verzichten.