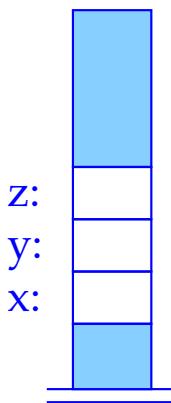


Variablen ordnen wir Speicherzellen in **S** zu:



Die Übersetzungsfunktionen benötigen als weiteres Argument eine Funktion  $\rho$ , die für jede Variable  $x$  die (Relativ-)Adresse von  $x$  liefert. Die Funktion  $\rho$  heißt **Adress-Umgebung** (Address Environment).

Variablen können auf zwei Weisen verwendet werden.

Beispiel:  $x = y + 1$

Für  $y$  sind wir am **Inhalt** der Zelle, für  $x$  an der **Adresse** interessiert.

L-Wert von  $x$  = Adresse von  $x$

R-Wert von  $x$  = Inhalt von  $x$

<code>code<sub>R</sub> e ρ</code>	liefert den Code zur Berechnung des R-Werts von $e$ in der Adress-Umgebung $ρ$
<code>code<sub>L</sub> e ρ</code>	analog für den L-Wert

Achtung:

Nicht jeder Ausdruck verfügt über einen L-Wert (Bsp.:  $x + 1$ ).

Wir definieren:

$$\begin{aligned}\text{code}_R (e_1 + e_2) \rho &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{add}\end{aligned}$$

... analog für die anderen binären Operatoren

$$\begin{aligned}\text{code}_R (-e) \rho &= \text{code}_R e \rho \\ &\quad \text{neg}\end{aligned}$$

... analog für andere unäre Operatoren

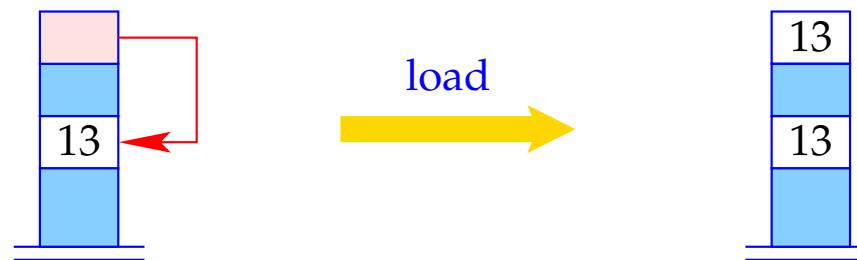
$$\begin{aligned}\text{code}_R q \rho &= \text{loadc} q \\ \text{code}_L x \rho &= \text{loadc} (\rho x)\end{aligned}$$

...

$$\text{code}_R x \rho = \text{code}_L x \rho$$

load

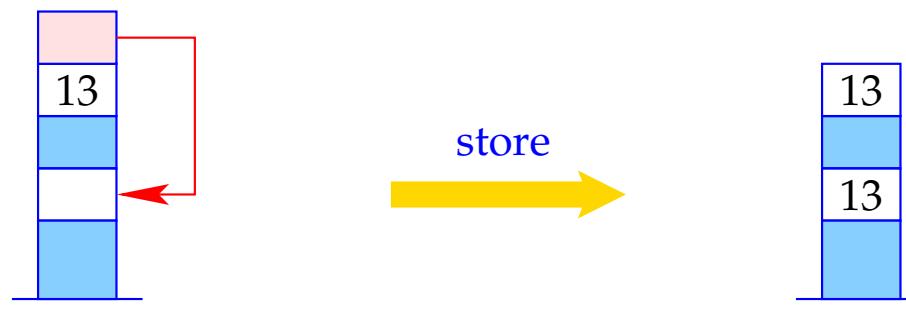
Die Instruktion **load** lädt den Wert der Speicherzelle, deren Adresse oben auf dem Stack liegt.



$$S[SP] = S[S[SP]];$$

$$\begin{aligned}
 \text{code}_R (x = e) \rho &= \text{code}_R e \rho \\
 &\quad \text{code}_L x \rho \\
 &\quad \text{store}
 \end{aligned}$$

Die Instruktion **store** schreibt den Inhalt der zweitobersten Speicherzelle in die Speicherzelle, deren Adresse oben auf dem Keller steht, lässt den geschriebenen Wert aber oben auf dem Keller liegen :-)



$S[S[SP]] = S[SP-1];$   
 $SP--;$

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

loadc 7  
load

loadc 1  
sub

loadc 4  
store

**Optimierungen:**

Einführung von Spezialbefehlen für häufige Befehlsfolgen, hier etwa:

$$\begin{array}{rcl} \text{loada q} & = & \text{loadc q} \\ & & \text{load} \\ \text{storea q} & = & \text{loadc q} \\ & & \text{store} \end{array}$$

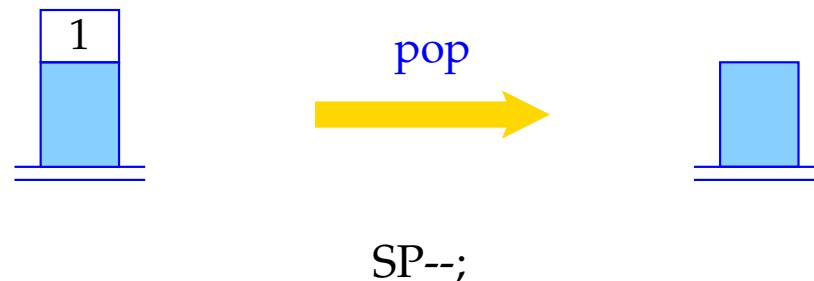
### 3 Anweisungen und Anweisungsfolgen

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:

$$\begin{aligned}\text{code } e; \rho &= \text{code}_R e \rho \\ &\quad \text{pop}\end{aligned}$$

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

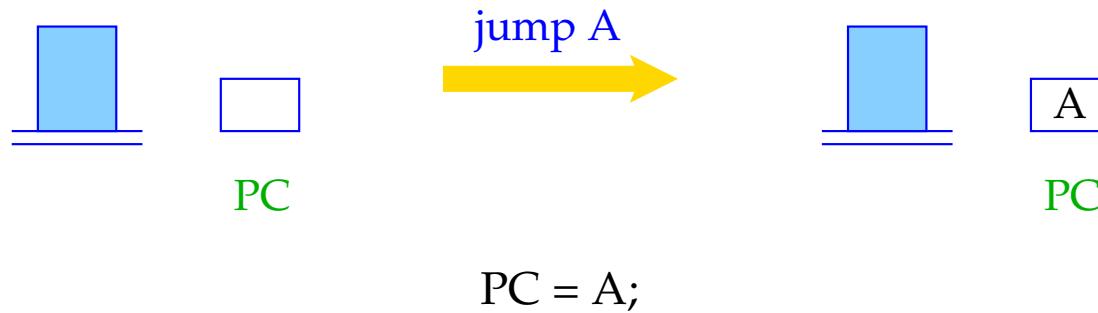


Der Code für eine Statement-Folge ist die Konkatenation des Codes for die einzelnen Statements in der Folge:

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

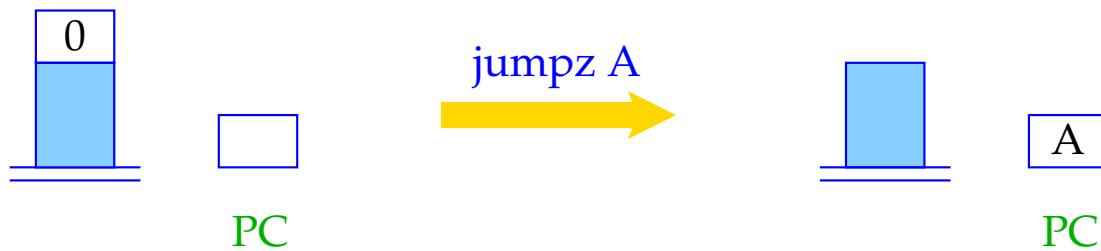
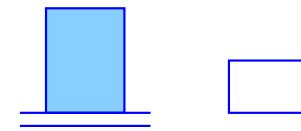
## 4 Bedingte und iterative Anweisungen

Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:

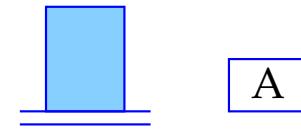




jumpz A



jumpz A



PC

A

PC

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.

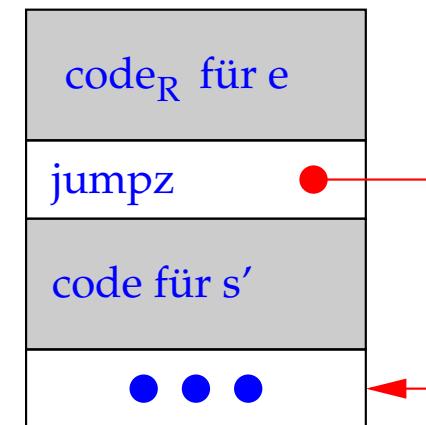
## 4.1 Bedingte Anweisung, einseitig

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!

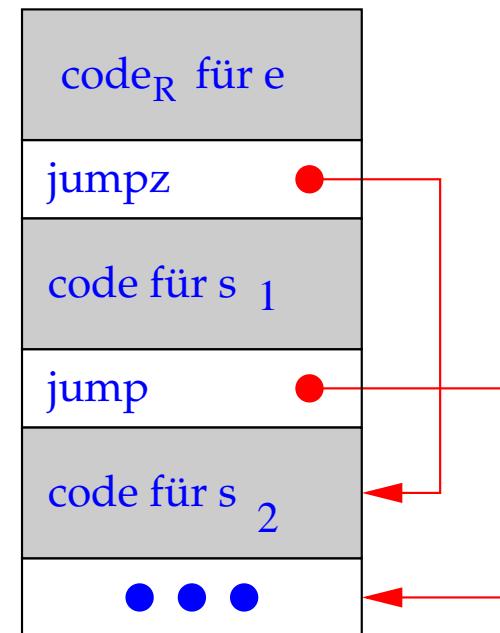
$\text{code } s \rho = \text{code}_R e \rho$   
jumpz A  
 $\text{code } s' \rho$   
A : ...



## 4.2 Zweiseitiges if

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

```
code s ρ    =   codeR e ρ  
                jumpz A  
                code s1 ρ  
                jump B  
A : code s2 ρ  
B : ...
```



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

$$s \equiv \mathbf{if} (x > y) \quad (i)$$

$$x = x - y; \quad (ii)$$

$$\mathbf{else} \ y = y - x; \quad (iii)$$

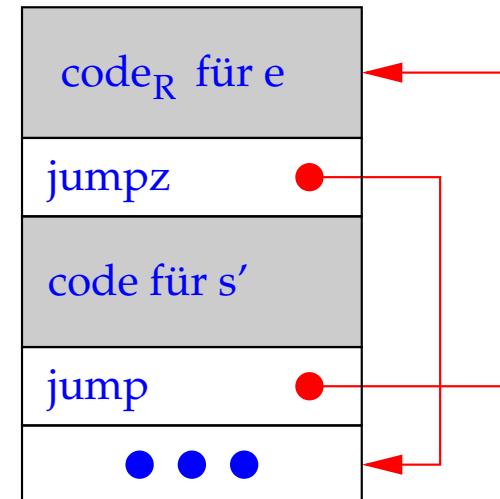
Dann liefert code  $s \rho$ :

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:	...
(i)	(ii)		(iii)

## 4.3 while-Schleifen

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

$\text{code } s \rho =$   
A :  $\text{code}_R e \rho$   
 $\text{jumpz } B$   
 $\text{code } s' \rho$   
 $\text{jump } A$   
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 code  $s \rho$  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  $s \equiv \text{for } (e_1; e_2; e_3) s'$  ist äquivalent zu der Statementfolge  $e_1; \text{while } (e_2) \{s' e_3; \}$  – 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 : ...
```