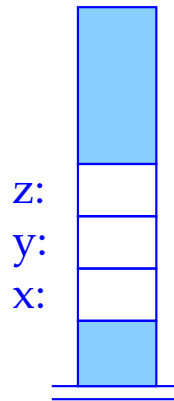


Variablen ordnen wir Speicherzellen in **S** zu:



Die Übersetzungsfunktionen benötigen als weiteres Argument eine Funktion ρ , die für jede Variable x die (Relativ-)Adresse von x liefert. Die Funktion ρ 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_R $e \rho$	liefert den Code zur Berechnung des R-Werts von e in der Adress-Umgebung ρ
code_L $e \rho$	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

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

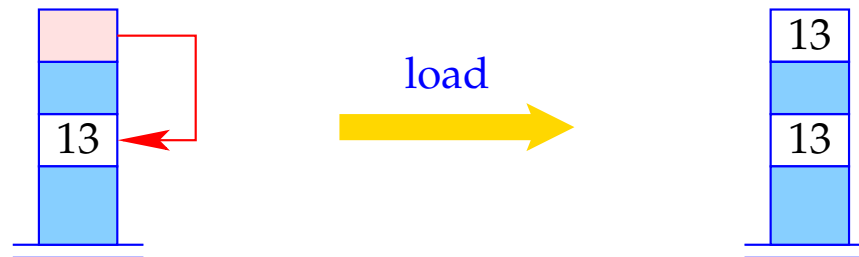
... analog für andere unäre Operatoren

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

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

load

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



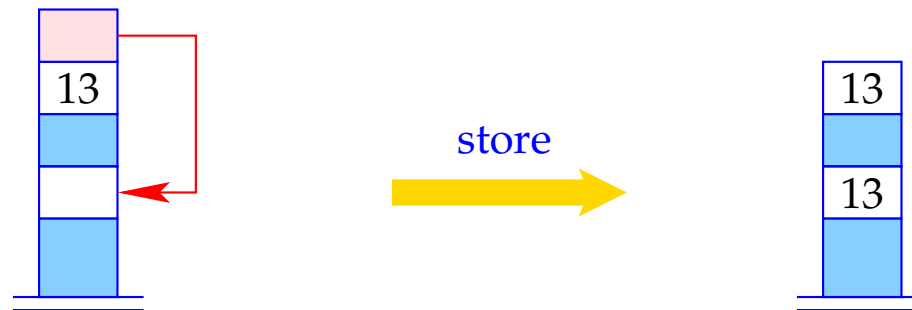
$S[SP] = S[S[SP]];$

$$\text{code}_R (x = e) \rho = \text{code}_R e \rho$$

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

store

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 $\text{code}_R e \rho$:

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

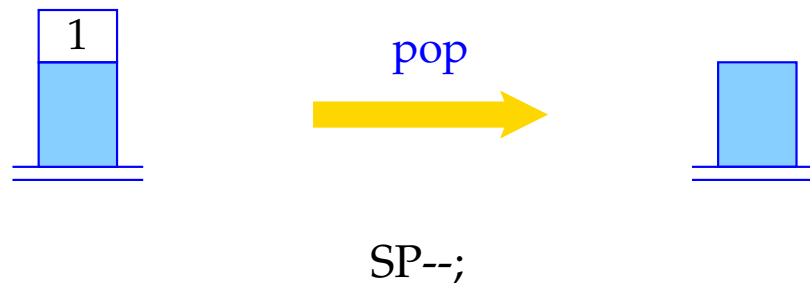
Ist e ein Ausdruck, dann ist $e;$ eine Anweisung (Statement).

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 ...

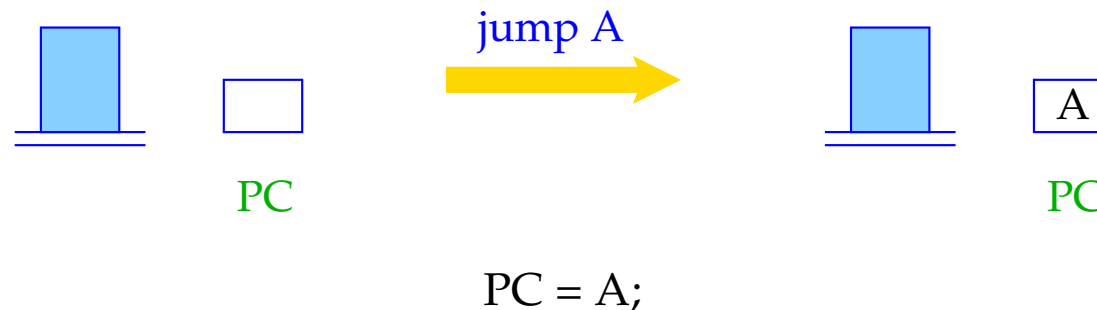


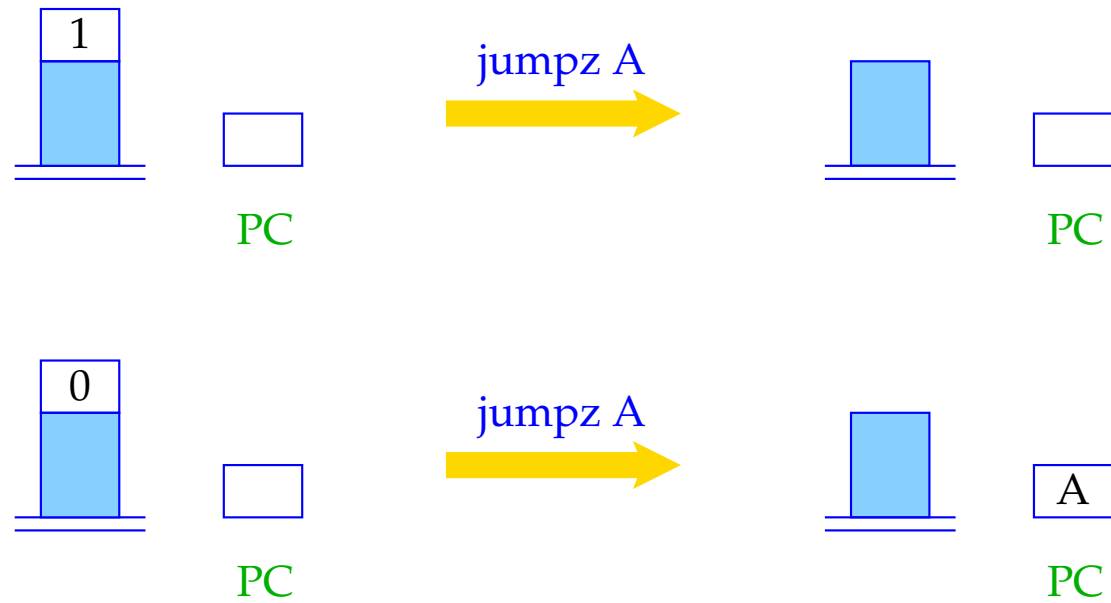
Der Code für eine Statement-Folge ist die Konkatenation des Codes for die einzelnen Statements in der Folge:

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

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

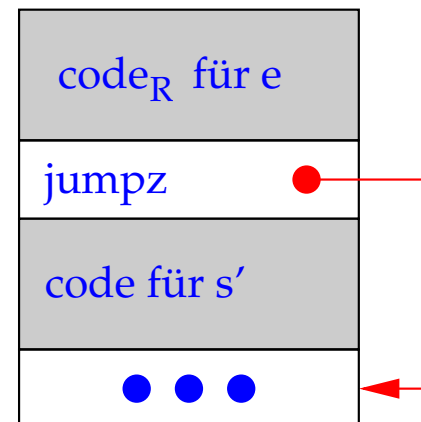
Betrachten wir zuerst $s \equiv \text{if } (e) s'$.

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!

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

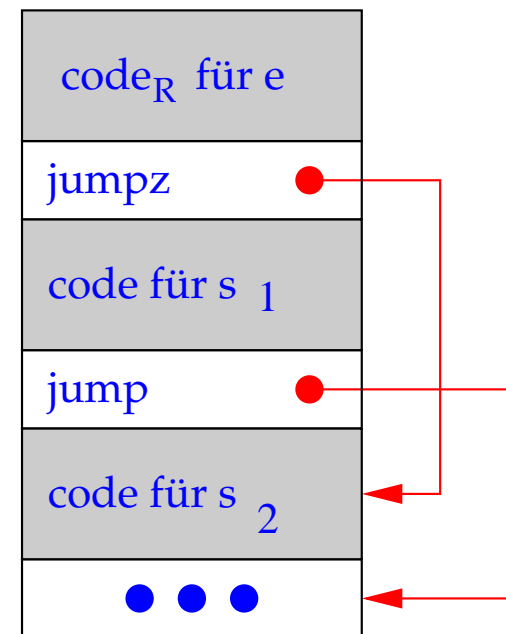
$\text{jumpz } A$
 $\text{code } s' \rho$
 $A : \dots$



4.2 Zweiseitiges if

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

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



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 7

gr

jumpz A

(i)

loada 4

loada 7

sub

storea 4

pop

jump B

(ii)

A: loada 7

loada 4

sub

storea 7

pop

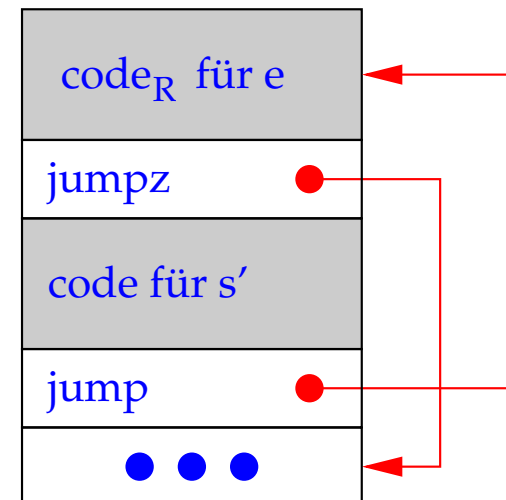
B: ...

(iii)

4.3 while-Schleifen

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

$\text{code } s \rho =$
A : $\text{code}_R e \rho$
 jumpz B
 $\text{code } s' \rho$
 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 $s \equiv \mathbf{for} (e_1; e_2; e_3) s'$ 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 :  ...
```