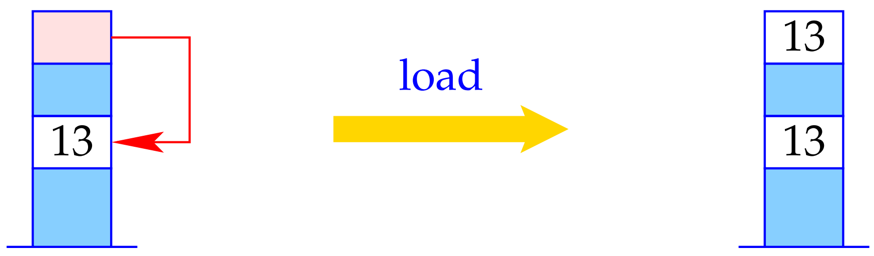
Wir definieren:

$$
\begin{array}{lcl}
\text{code}_\text{R}\ (e_1 + e_2)\ \rho & = & \text{code}_\text{R}\ e_1\ \rho \\
 & & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{add} \\
 & & \quad \text{... analog für die anderen binären Operatoren} \\
\text{code}_\text{R}\ (-e)\ \rho & = & \text{code}_\text{R}\ e\ \rho \\
 & & \text{neg} \\
 & & \quad \text{... analog für andere unäre Operatoren} \\
\text{code}_\text{R}\ q\ \rho & = & \text{loadc q} \\
\text{code}_\text{L}\ x\ \rho & = & \text{loadc}\ (\rho\, x) \\
 & \ldots &
\end{array}
$$

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

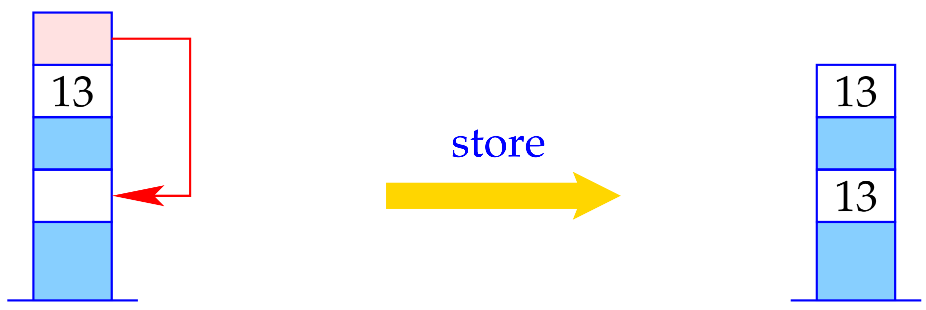
Die Instruktion load lädt den Wert der Speicherzelle, deren Adresse oben auf dem Stack liegt.



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

$$\text{code}_\text{R}\ (x = e)\ \rho \quad = \quad \begin{array}{l} \text{code}_\text{R}\ e\ \rho \\ \text{code}_\text{L}\ x\ \rho \\ \text{store} \end{array}$$

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 $\text{code}_\text{R}\ e\ \rho$:

loadc 7	loadc 1	loadc 4
load	sub	store

Optimierungen:

Einführung von Spezialbefehlen für häufige Befehlsfolgen, hier etwa:

loada q	=	loadc q
		load
storea q	=	loadc q
		store

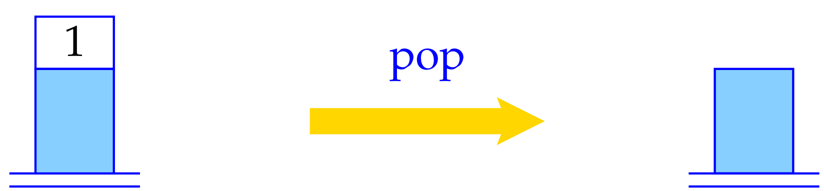
3 Anweisungen und Anweisungsfolgen

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:

$$\begin{array}{lll} \text{code}\ e;\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{pop} \end{array}$$

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



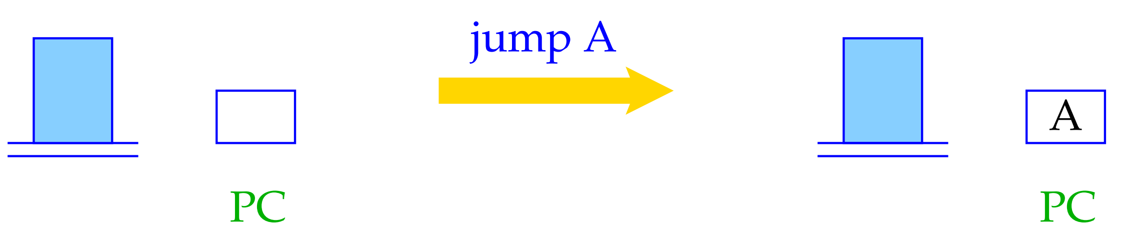
SP--;

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

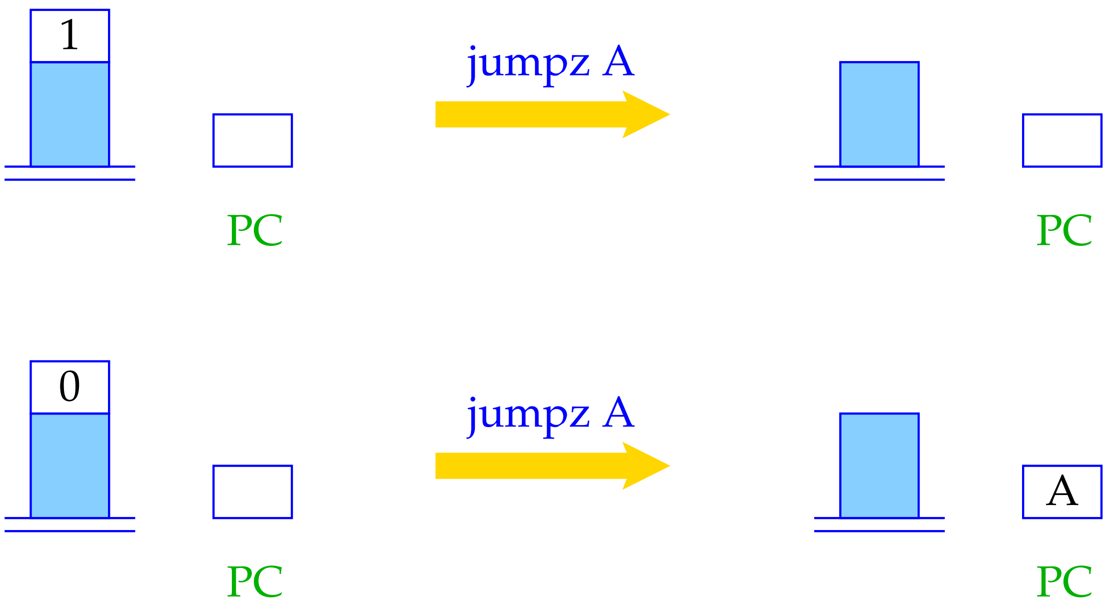
$$\begin{array}{lcl} \text{code}\ (s\ ss)\ \rho & = & \text{code}\ s\ \rho \\ & & \text{code}\ ss\ \rho \\ \text{code}\ \varepsilon\ \rho & = & \qquad // \quad \textit{leere Folge von Befehlen} \end{array}$$

4 Bedingte und iterative Anweisungen

Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:



PC = A;



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.

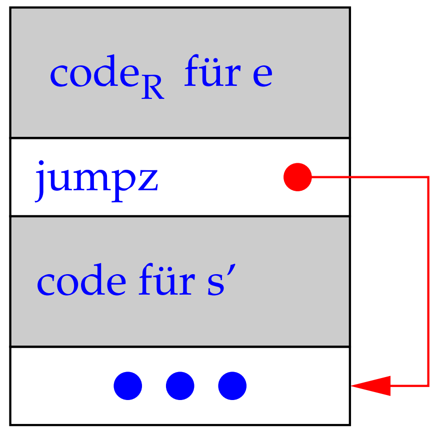
4.1 Bedingte Anweisung, einseitig

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

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!

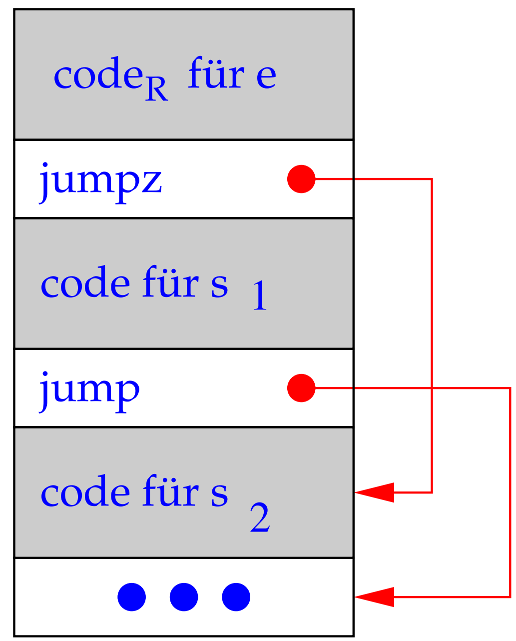
$$
\begin{aligned}
\text{code}\ s\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\
& \text{jumpz A} \\
& \text{code}\ s'\ \rho \\
\text{A}: \quad & \ldots
\end{aligned}
$$



4.2 Zweiseitiges if

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

$$
\begin{aligned}
\text{code}\ s\ \rho \quad &= \quad \text{code}_\text{R}\ e\ \rho \\
& \quad\quad \text{jumpz A} \\
& \quad\quad \text{code}\ s_1\ \rho \\
& \quad\quad \text{jump B} \\
\text{A}: & \quad\quad \text{code}\ s_2\ \rho \\
\text{B}: & \quad\quad \ldots
\end{aligned}
$$



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

$$
\begin{array}{llll}
s & \equiv & \textbf{if}\,(x > y) & (i) \\
 & & \quad x = x - y; & (ii) \\
 & & \textbf{else}\; y = y - x; & (iii)
\end{array}
$$

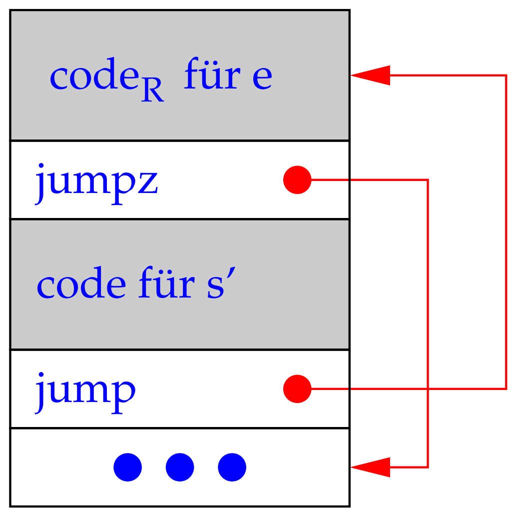
Dann liefert code $s\ \rho$:

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 $s \equiv \textbf{while}\ (e)\ s'$. Dafür erzeugen wir:

$$
\begin{array}{lll}
\text{code}\ s\ \rho & = & \\
 & \text{A}: & \text{code}_\text{R}\ e\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$



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

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

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

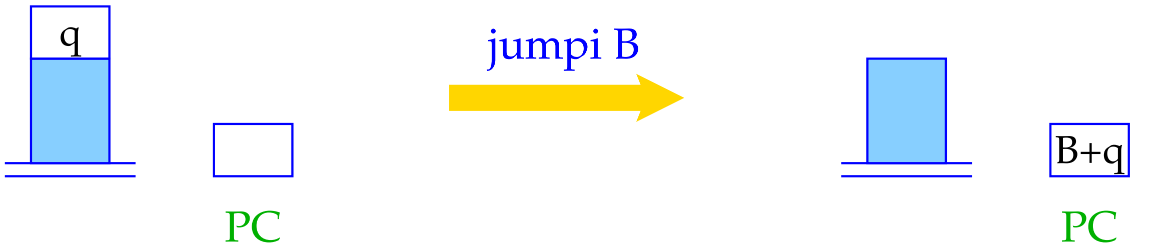
Die **for**-Schleife $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ ist äquivalent zu der Statementfolge $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – sofern s' keine **continue**-Anweisung enthält.
Darum übersetzen wir:

$$
\begin{array}{rcl}
\text{code}\ s\ \rho & = & \text{code}_\text{R}\ e_1 \\
 & & \text{pop} \\
 & \text{A}: & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{code}_\text{R}\ e_3\ \rho \\
 & & \text{pop} \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$

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*-tem 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:

$$
\begin{array}{lll}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
 & & \qquad \textbf{case}\ 0\!:\quad ss_0\ \textbf{break;} \\
 & & \qquad \textbf{case}\ 1\!:\quad ss_1\ \textbf{break;} \\
 & & \qquad\qquad \vdots \\
 & & \qquad \textbf{case}\ k-1\!:\quad ss_{k-1}\ \textbf{break;} \\
 & & \qquad \textbf{default:}\quad ss_k \\
 & & \}
\end{array}
$$

Dann ergibt sich für s die Instruktionsfolge:

$$
\begin{array}{llllllll}
\text{code } s\ \rho & = & \text{code}_{\text{R}}\ e\ \rho & & \text{C}_0: & \text{code } ss_0\ \rho & \text{B}: & \text{jump C}_0 \\
 & & \text{check } 0\ k\ \text{B} & & & \text{jump D} & & \dots \\
 & & & & & \dots & & \text{jump C}_k \\
 & & & & \text{C}_k: & \text{code } ss_k\ \rho & \text{D}: & \dots \\
 & & & & & \text{jump D} & &
\end{array}
$$

- Das Macro check $0\ k$ B überprüft, ob der R-Wert von e im Intervall $[0, k]$ liegt, und führt einen indizierten Sprung in die Tabelle B aus.
- Die Sprungtabelle enthält direkte Sprünge zu den jeweiligen Alternativen.
- Am Ende jeder Alternative steht ein Sprung hinter das **switch**-Statement.