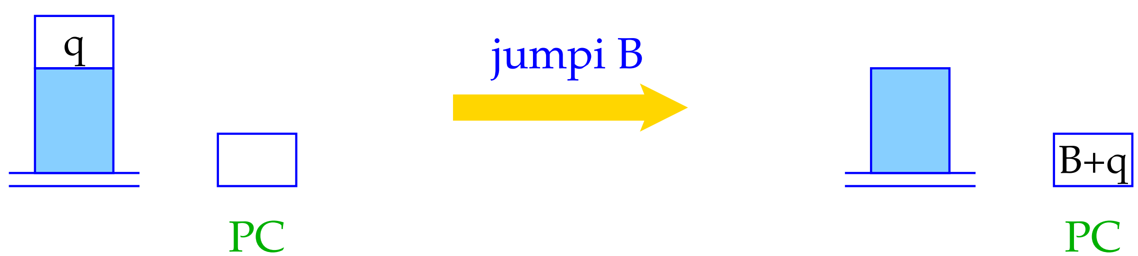
4.5 Das switch-Statement

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:

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

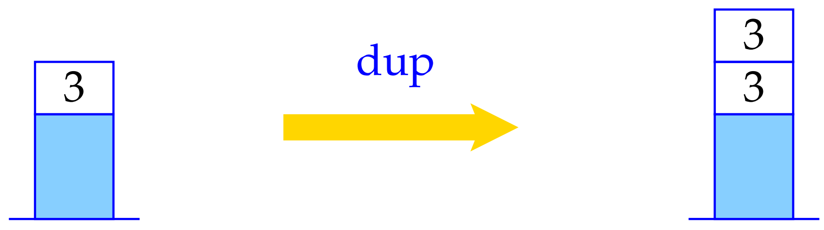
Dann ergibt sich für s die Instruktionsfolge:

code $s\ \rho$	$=$	$\text{code}_R\ e\ \rho$	C_0:	code $ss_0\ \rho$	B:	jump C_0
		check $0\ k$ B		jump D		$\ldots$
				$\ldots$		jump C_k
			C_k:	code $ss_k\ \rho$	D:	$\ldots$
				jump D		

- 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.

check 0 k B	=	dup	dup		jumpi B
		loadc 0	loadc k	A:	pop
		geq	leq		loadc k
		jumpz A	jumpz A		jumpi B

- 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 check durch jumpi B ersetzen :-)

5 Speicherbelegung für Variablen

Ziel:

Ordne jeder Variablen x statisch, d. h. zur Übersetzungszeit, eine feste (Relativ-)Adresse $\quad \rho\, x \quad$ zu!

Annahmen:

- Variablen von Basistypen wie **int**, ... erhalten eine Speicherzelle.
- Variablen werden in der Reihenfolge im Speicher abgelegt, wie sie deklariert werden, und zwar ab Adresse 1.

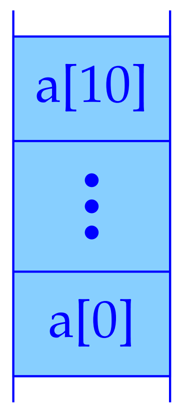
Folglich erhalten wir für die Deklaration $\quad d \equiv t_1\; x_1; \ldots\; t_k\; x_k; \quad$ (t_i einfach) die Adress-Umgebung ρ mit

$$\rho\, x_i = i, \quad i = 1, \ldots, k$$

5.1 Felder

Beispiel: **int** [11] a;

Das Feld a enthält 11 Elemente und benötigt darum 11 Zellen.
$\rho\, a$ ist die Adresse des Elements $a[0]$.



Notwendig ist eine Funktion sizeof (hier: $|\cdot|$), die den Platzbedarf eines Typs berechnet:

$$|t| = \begin{cases} 1 & \text{falls } t \text{ einfach} \\ k \cdot |t'| & \text{falls } t \equiv t'[k] \end{cases}$$

Dann ergibt sich für die Deklaration $\quad d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

$$\begin{array}{lll} \rho\, x_1 & = & 1 \\ \rho\, x_i & = & \rho\, x_{i-1} + |t_{i-1}| \qquad\qquad \text{für } i > 1 \end{array}$$

Weil $|\cdot|$ zur Übersetzungszeit berechnet werden kann, kann dann auch ρ zur Übersetzungszeit berechnet werden.

Aufgabe:

Erweitere code_L und code_R auf Ausdrücke mit indizierten Feldzugriffen.

Sei $\quad t[c]\; a; \quad$ die Deklaration eines Feldes $\quad a$.

Um die Anfangsadresse der Datenstruktur $\quad a[i] \quad$ zu bestimmen, müssen wir $\rho\, a + |t| * \textit{(R-Wert von i)}$ ausrechnen. Folglich:

$$
\begin{array}{lll}
\text{code}_\text{L}\; a[e]\; \rho & = & \text{loadc}\; (\rho\, a) \\
 & & \text{code}_\text{R}\; e\; \rho \\
 & & \text{loadc}\; |t| \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

… oder allgemeiner:

$$\begin{array}{lll} \mathrm{code_L}\ e_1[e_2]\ \rho & = & \mathrm{code_R}\ e_1\ \rho \\ & & \mathrm{code_R}\ e_2\ \rho \\ & & \mathrm{loadc}\ |t| \\ & & \mathrm{mul} \\ & & \mathrm{add} \end{array}$$

Bemerkung:

- In C ist ein Feld ein Zeiger. Ein deklariertes Feld a ist eine Zeiger-Konstante, deren R-Wert die Anfangsadresse des Feldes ist.
- Formal setzen wir für ein Feld e: $\mathrm{code_R}\ e\ \rho = \mathrm{code_L}\ e\ \rho$
- In C sind äquivalent (als L-Werte):

$$2[a] \qquad a[2] \qquad a+2$$

5.2 Strukturen

In Modula heißen Strukturen Records.

Vereinfachung:

Komponenten-Namen werden nicht anderweitig verwandt.
Alternativ könnte man zu jedem Struktur-Typ *st* eine separate Komponenten-Umgebung ρ_{st} verwalten :-)

Sei $\quad$ **struct** { **int** a; **int** b; } x; $\quad$ Teil einer Deklarationsliste.

- x erhält die erste freie Zelle des Platzes für die Struktur als Relativ-Adresse.
- Für die Komponenten vergeben wir Adressen relativ zum Anfang der Struktur, hier $a \mapsto 0$, $b \mapsto 1$.

Sei allgemein $\quad t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. Dann ist

$$
\begin{aligned}
|t| &= \sum_{i=1}^{k} |t_i| \\
\rho\, c_1 &= 0 \quad \text{und} \\
\rho\, c_i &= \rho\, c_{i-1} + |t_{i-1}| \quad \text{für } i > 1
\end{aligned}
$$

Damit erhalten wir:

$$
\begin{aligned}
\text{code}_\text{L}\ (e.c)\ \rho \quad = \quad & \text{code}_\text{L}\ e\ \rho \\
& \text{loadc}\ (\rho\, c) \\
& \text{add}
\end{aligned}
$$

Beispiel:

Sei **struct** { **int** a; **int** b; } x; mit $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

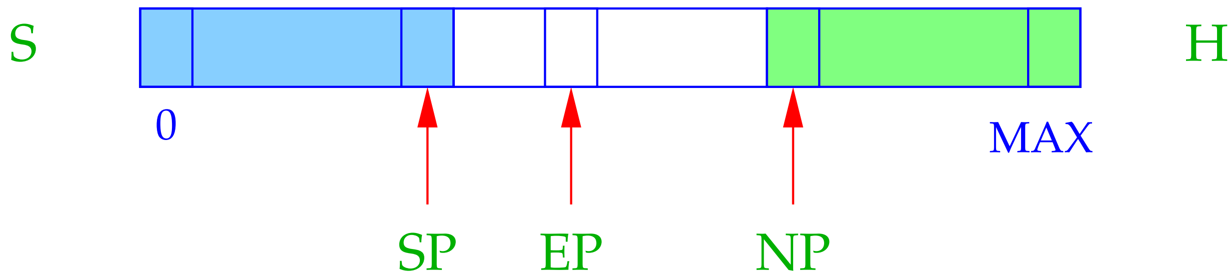
Dann ist

$$\text{code}_\text{L}\ (x.b)\ \rho \quad = \quad \begin{array}{l}\texttt{loadc 13}\\ \texttt{loadc 1}\\ \texttt{add}\end{array}$$

6 Zeiger und dynamische Speicherverwaltung

Zeiger (Pointer) gestatten den Zugriff auf anonyme, dynamisch erzeugte Datenelemente, deren Lebenszeit nicht dem LIFO-Prinzip unterworfen ist.

$\Longrightarrow$ Wir benötigen eine weitere potentiell beliebig große Datenstruktur H – den Heap (bzw. die Halde):



NP $\hat{=}$ New Pointer; zeigt auf unterste belegte Haldenzelle.

EP $\hat{=}$ Extreme Pointer; zeigt auf die Zelle, auf die der SP maximal zeigen kann (innerhalb der aktuellen Funktion).

Idee dabei:

- Chaos entsteht, wenn Stack und Heap sich überschneiden (Stack Overflow).
- Eine Überschneidung kann bei jeder Erhöhung von SP, bzw. jeder Erniedrigung des NP eintreten.
- EP erspart uns die Überprüfungen auf Überschneidung bei den Stackoperationen :-)
- Die Überprüfungen bei Heap-Allokationen bleiben erhalten :-(.

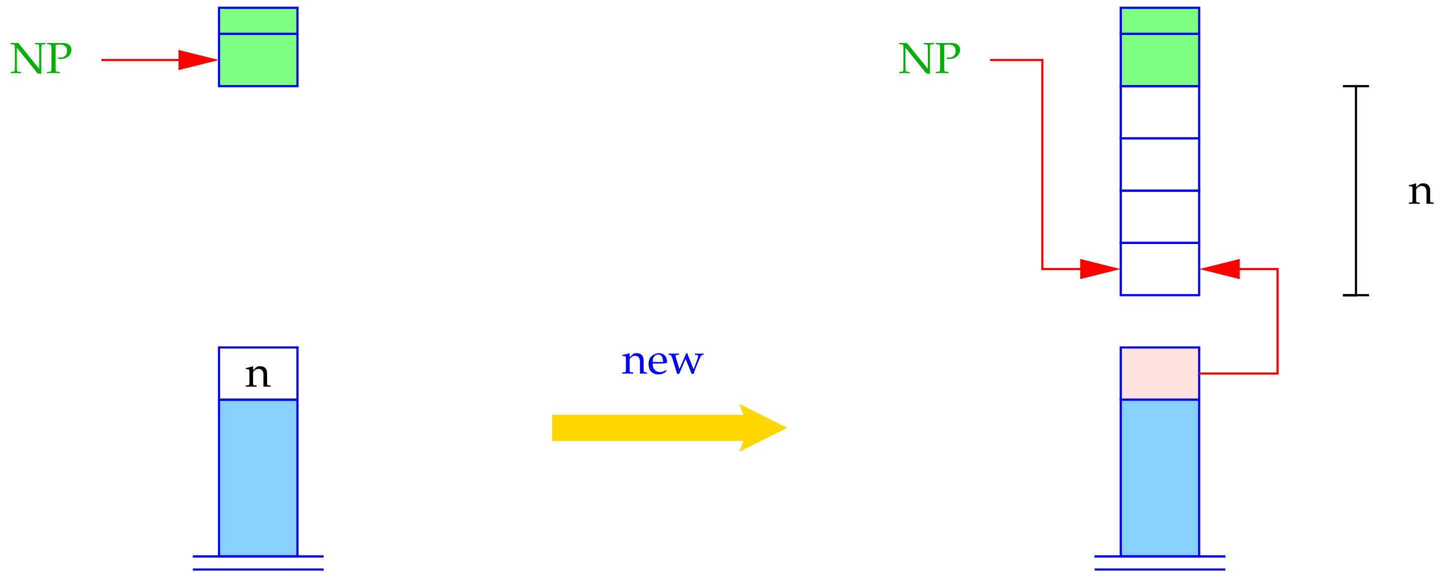
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- Zeiger zu erzeugen, d.h. Zeiger auf Speicherzellen zu setzen; sowie
- Zeiger zu dereferenzieren, d. h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Es gibt zwei Arten, Zeiger zu erzeugen:

(1) Ein Aufruf von **malloc** liefert einen Zeiger auf eine Heap-Zelle:

$$\begin{array}{lll} \text{code}_\text{R}\ \mathbf{malloc}\,(e)\ \rho & = & \text{code}_\text{R}\ e\ \rho \\ & & \text{new} \end{array}$$


NP
n
new
NP
n

```
if (NP - S[SP] ≤ EP)
        S[SP] = NULL;
else {
        NP = NP - S[SP];
        S[SP] = NP;
}
```

- NULL ist eine spezielle Zeigerkonstante (etwa 0 :-)
- Im Falle einer Kollision von Stack und Heap wird der NULL-Zeiger zurückgeliefert.

(2) Die Anwendung des Adressoperators $\&$ liefert einen Zeiger auf eine Variable, d. h. deren Adresse ($\widehat{=}$ L-Wert). Deshalb:

$$\text{code}_\text{R}\ (\&e)\ \rho = \text{code}_\text{L}\ e\ \rho$$

Dereferenzieren von Zeigern:

Die Anwendung des Operators $*$ auf den Ausdruck e liefert den Inhalt der Speicherzelle, deren Adresse der R-Wert von e ist:

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

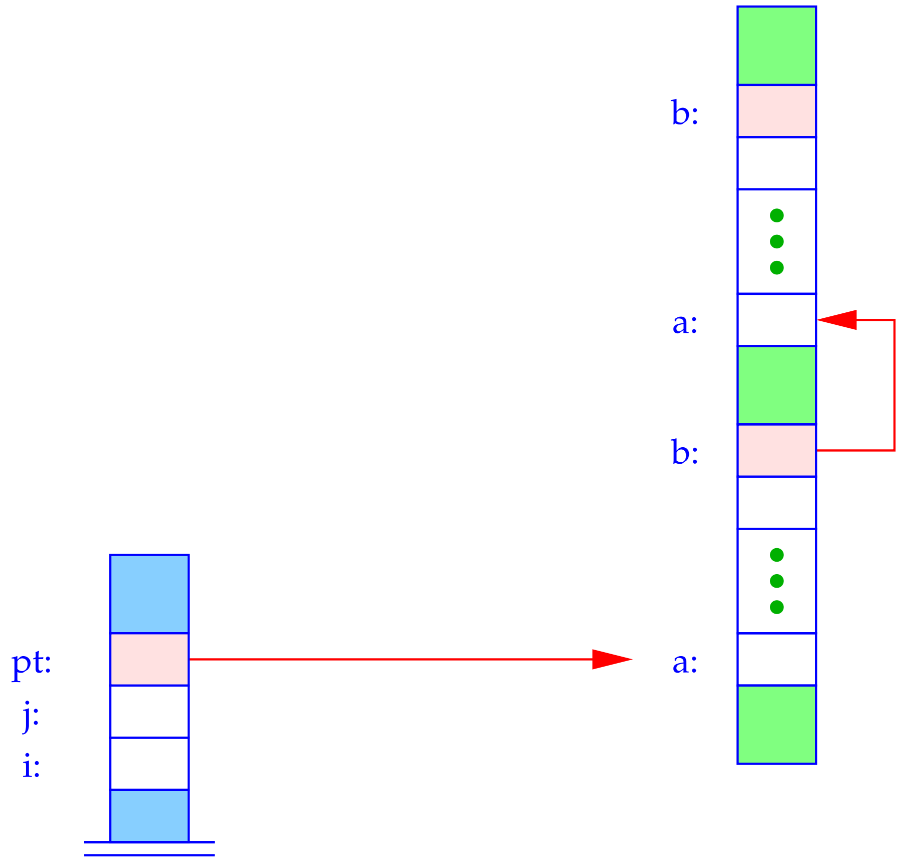
Beispiel: Betrachte für

$$\textbf{struct } t \; \{ \textbf{ int } a[7]; \textbf{ struct } t *b; \};$$
$$\textbf{int } i, j;$$
$$\textbf{struct } t *pt;$$

den Ausdruck $e \equiv ((pt \to b) \to a)[i+1]$

Wegen $e \to a \equiv (*e).a$ gilt:

$$\begin{aligned} \text{code}_\text{L} \; (e \to a) \; \rho &= \text{code}_\text{R} \; e \; \rho \\ &\quad \text{loadc } (\rho\, a) \\ &\quad \text{add} \end{aligned}$$


b:
a:
b:
pt:
j:
i:
a:

Sei $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$. Dann ist:

$$
\begin{array}{lclcl}
\text{code}_\text{L}\ e\ \rho & = & \text{code}_\text{R}\ ((pt \to b) \to a)\ \rho & = & \text{code}_\text{R}\ ((pt \to b) \to a)\ \rho \\
& & \text{code}_\text{R}\ (i+1)\ \rho & & \text{loada } 1 \\
& & \text{loadc } 1 & & \text{loadc } 1 \\
& & \text{mul} & & \text{add} \\
& & \text{add} & & \text{loadc } 1 \\
& & & & \text{mul} \\
& & & & \text{add}
\end{array}
$$

Für Felder ist der R-Wert gleich dem L-Wert. Deshalb erhalten wir:

$$\text{code}_R\ ((pt \to b) \to a)\ \rho \quad = \quad \text{code}_R\ (pt \to b)\ \rho \quad = \quad \begin{array}{l}\text{loada } 3\\ \text{loadc } 7\\ \text{add}\\ \text{load}\\ \text{loadc } 0\\ \text{add}\end{array}$$

where the middle expression continues:

$$\begin{array}{l}\text{code}_R\ (pt \to b)\ \rho\\ \text{loadc } 0\\ \text{add}\end{array}$$

Damit ergibt sich insgesamt die Folge:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add

7 Zusammenfassung

Stellen wir noch einmal die Schemata zur Übersetzung von Ausdrücken zusammen.

$$
\begin{array}{lcll}
\text{code}_\text{L}\ (e_1[e_2])\ \rho & = & \text{code}_\text{R}\ e_1\ \rho & \\
 & & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{loadc}\ |t| & \\
 & & \text{mul} & \\
 & & \text{add} & \text{sofern } e_1 \text{ Typ } t\,[\,] \text{ hat} \\
 & & & \\
\text{code}_\text{L}\ (e.a)\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{loadc}\ (\rho\, a) & \\
 & & \text{add} &
\end{array}
$$

$$\text{code}_\text{L}\ (*e)\ \rho \quad = \quad \text{code}_\text{R}\ e\ \rho$$

$$\text{code}_\text{L}\ x\ \rho \quad = \quad \text{loadc}\ (\rho\, x)$$

$$\text{code}_\text{R}\ (\&e)\ \rho \quad = \quad \text{code}_\text{L}\ e\ \rho$$

$$\begin{aligned}\text{code}_\text{R}\ (\text{malloc}\,(e))\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\ & \text{new}\end{aligned}$$

$$\text{code}_\text{R}\ e\ \rho \quad = \quad \text{code}_\text{L}\ e\ \rho \qquad\qquad \text{falls } e \text{ ein Feld ist}$$

$$\begin{aligned}\text{code}_\text{R}\ (e_1 \,\Box\, e_2)\ \rho \quad = \quad & \text{code}_\text{R}\ e_1\ \rho \\ & \text{code}_\text{R}\ e_2\ \rho \\ & \text{op} \qquad\qquad \text{op Befehl zu Operator '}\Box\text{'}\end{aligned}$$

$$\begin{array}{lcll}
\mathrm{code_R}\ q\ \rho & = & \mathtt{loadc\ q} & q \text{ Konstante} \\
\\
\mathrm{code_R}\ (e_1 = e_2)\ \rho & = & \mathrm{code_R}\ e_2\ \rho & \\
 & & \mathrm{code_L}\ e_1\ \rho & \\
 & & \mathtt{store} & \\
\\
\mathrm{code_R}\ e\ \rho & = & \mathrm{code_L}\ e\ \rho & \\
 & & \mathtt{load} & \text{sonst}
\end{array}$$

Beispiel: $\quad$ **int** $a[10], *b;$ $\quad$ mit $\rho = \{a \mapsto 7, b \mapsto 17\}$.

Betrachte das Statement: $\quad s_1 \equiv *a = 5;$

Dann ist:

$$
\begin{array}{lcl}
\text{code}_\text{L}\ (*a)\ \rho & = & \text{code}_\text{R}\ a\ \rho \quad = \quad \text{code}_\text{L}\ a\ \rho \quad = \quad \text{loadc } 7 \\
\text{code}\ s_1\ \rho & = & \text{loadc } 5 \\
 & & \text{loadc } 7 \\
 & & \text{store} \\
 & & \text{pop}
\end{array}
$$

Zur Übung übersetzen wir auch noch:

$$s_2 \equiv b = (\&a) + 2; \qquad \text{und} \qquad s_3 \equiv *(b+3) = 5;$$