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 **struct** { **int** a; **int** b; } x; 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

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

Damit erhalten wir:

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

Beispiel:

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

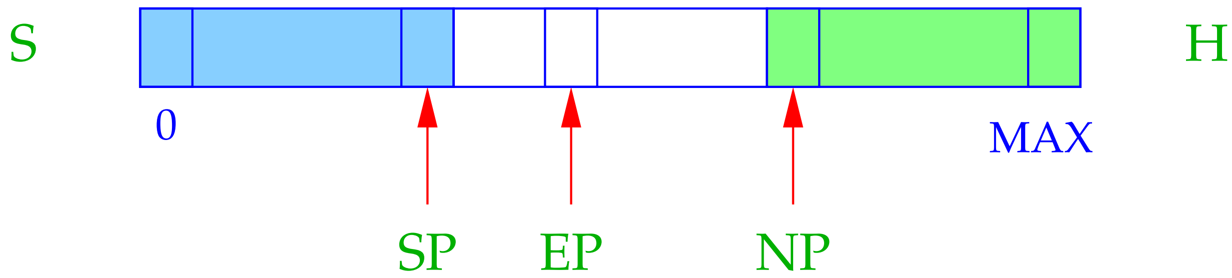
Dann ist

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

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

EP $\widehat{=}$ 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 :-(.

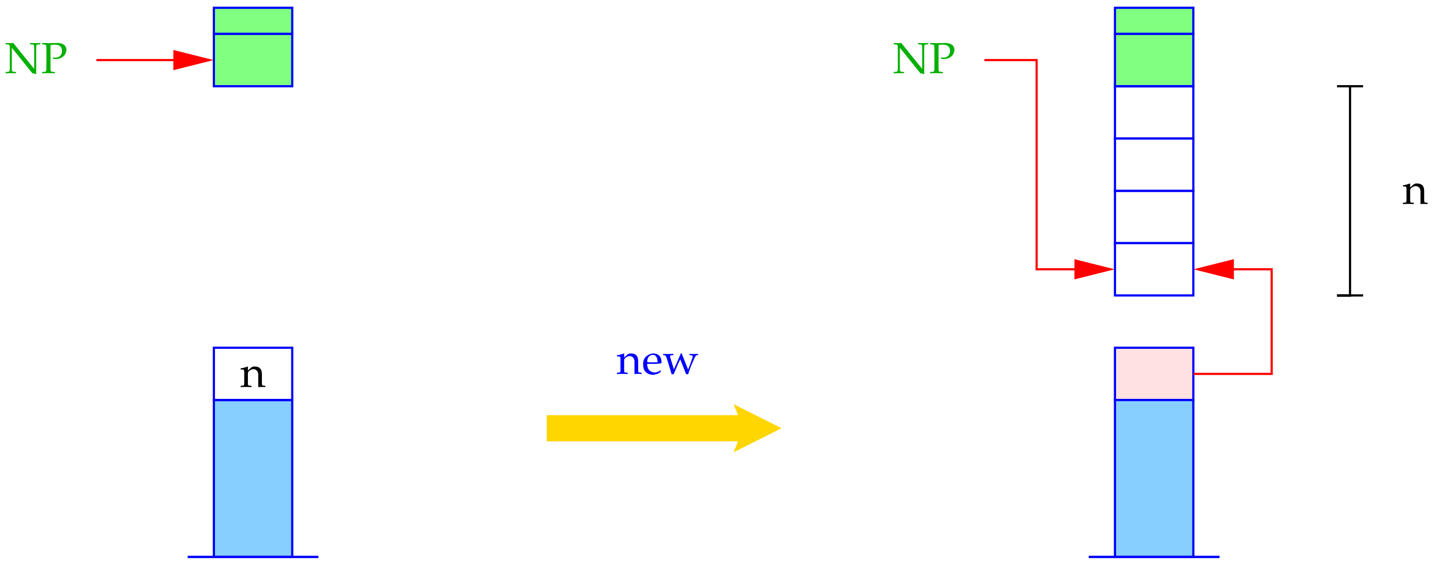
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- 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:

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


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:

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

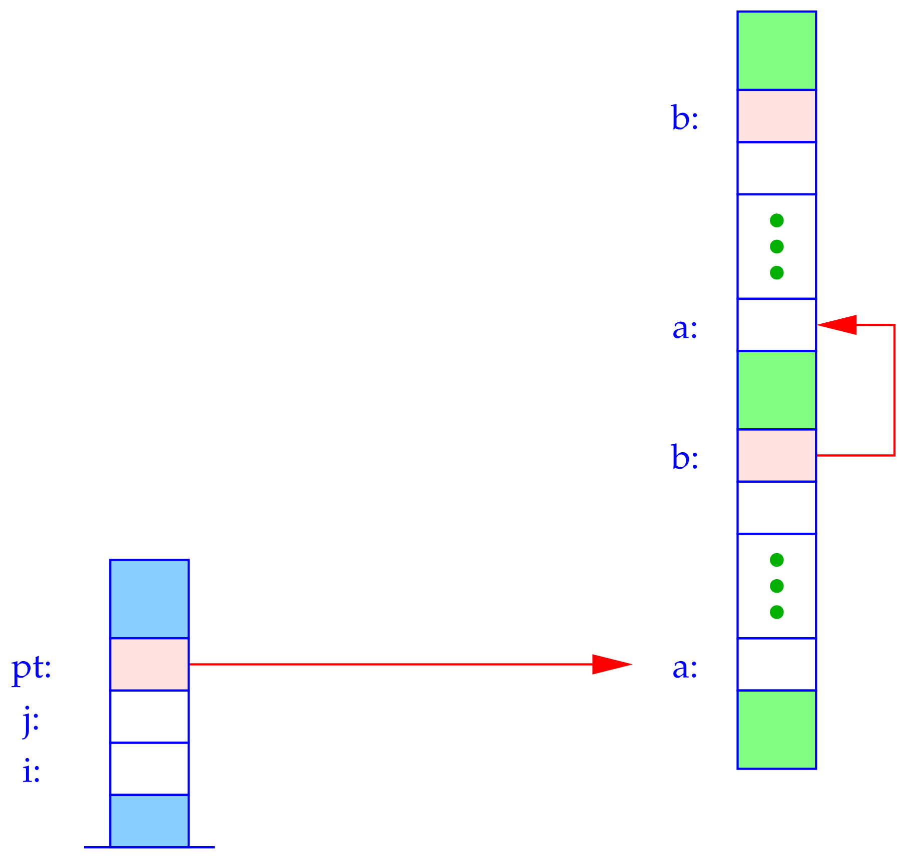
Beispiel: Betrachte für

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

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

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

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


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

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

$$
\begin{array}{lllll}
\mathrm{code_L}\ e\ \rho & = & \mathrm{code_R}\ ((pt \rightarrow b) \rightarrow a)\ \rho & = & \mathrm{code_R}\ ((pt \rightarrow b) \rightarrow a)\ \rho \\
 & & \mathrm{code_R}\ (i+1)\ \rho & & \mathtt{loada}\ 1 \\
 & & \mathtt{loadc}\ 1 & & \mathtt{loadc}\ 1 \\
 & & \mathtt{mul} & & \mathtt{add} \\
 & & \mathtt{add} & & \mathtt{loadc}\ 1 \\
 & & & & \mathtt{mul} \\
 & & & & \mathtt{add}
\end{array}
$$

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

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

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.

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

$$\mathrm{code}_L\ (*e)\ \rho \quad = \quad \mathrm{code}_R\ e\ \rho$$

$$\mathrm{code}_L\ x\ \rho \quad = \quad \mathrm{loadc}\ (\rho\, x)$$

$$\mathrm{code}_R\ (\&e)\ \rho \quad = \quad \mathrm{code}_L\ e\ \rho$$

$$\begin{array}{lcl} \mathrm{code}_R\ (\mathrm{malloc}\,(e))\ \rho & = & \mathrm{code}_R\ e\ \rho \\ & & \mathrm{new} \end{array}$$

$$\mathrm{code}_R\ e\ \rho \quad = \quad \mathrm{code}_L\ e\ \rho \qquad \text{falls } e \text{ ein Feld ist}$$

$$\begin{array}{lcll} \mathrm{code}_R\ (e_1 \,\Box\, e_2)\ \rho & = & \mathrm{code}_R\ e_1\ \rho & \\ & & \mathrm{code}_R\ e_2\ \rho & \\ & & \mathrm{op} & \mathrm{op}\ \text{Befehl zu Operator '}\Box\text{'} \end{array}$$

$$
\begin{array}{llll}
\mathrm{code_R}\ q\ \rho & = & \mathrm{loadc\ q} & q \text{ Konstante} \\[2em]
\mathrm{code_R}\ (e_1 = e_2)\ \rho & = & \mathrm{code_R}\ e_2\ \rho & \\
 & & \mathrm{code_L}\ e_1\ \rho & \\
 & & \mathrm{store} & \\[2em]
\mathrm{code_R}\ e\ \rho & = & \mathrm{code_L}\ e\ \rho & \\
 & & \mathrm{load} & \text{sonst}
\end{array}
$$

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

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

Dann ist:

$$
\begin{array}{lllllll}
\mathrm{code_L}\ (*a)\ \rho & = & \mathrm{code_R}\ a\ \rho & = & \mathrm{code_L}\ a\ \rho & = & \mathrm{loadc}\ 7 \\
\mathrm{code}\ s_1\ \rho & = & \mathrm{loadc}\ 5 & & & & \\
 & & \mathrm{loadc}\ 7 & & & & \\
 & & \mathrm{store} & & & & \\
 & & \mathrm{pop} & & & &
\end{array}
$$

Zur Übung übersetzen wir auch noch:

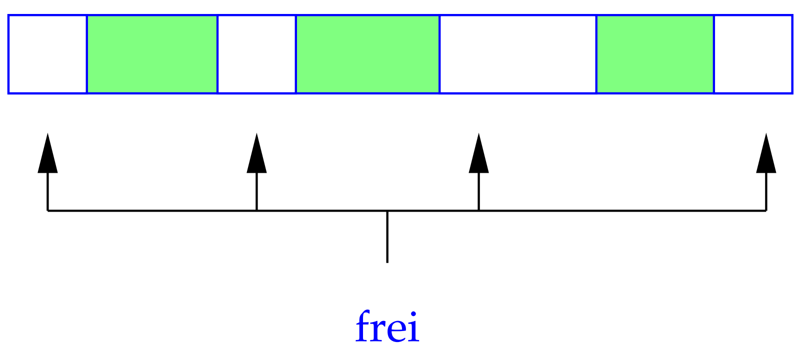
$$s_2 \equiv b = \&a[2]; \qquad \text{und} \qquad s_3 \equiv *(b+3) = 5;$$

code $(s_2 s_3)\ \rho$	$=$	loadc 7			loadc 5		
		loadc 2			loadc 17		
		loadc 1	//	*Skalierung*	load		
		mul			loadc 3		
		add			loadc 1	//	*Skalierung*
		loadc 17			mul		
		store			add		
		pop	//	*Ende von* s_2	store		
					pop	//	*Ende von* s_3

8 Freigabe von Speicherplatz

Probleme:

- Der freigegebene Speicherbereich wird noch von anderen Zeigern referenziert (dangling references).
- Nach einiger Freigabe könnte der Speicher etwa so aussehen (fragmentation):



Mögliche Auswege:

- Nimm an, der Programmierer weiß, was er tut. Verwalte dann die freien Abschnitte (etwa sortiert nach Größe) in einer speziellen Datenstruktur;

  $\Longrightarrow$ **malloc** wird teuer :-(

- Tue nichts, d.h.:

$$\begin{array}{lll} \text{code } \textbf{free}\,(e);\ \rho & = & \text{code}_{\text{R}}\ e\ \rho \\ & & \text{pop} \end{array}$$

  $\Longrightarrow$ einfach und (i.a.) effizient :-)

- Benutze eine automatische, evtl. “konservative” Garbage-Collection, die gelegentlich sicher nicht mehr benötigten Heap-Platz einsammelt und dann **malloc** zur Verfügung stellt.

9 Funktionen

Die Definition einer Funktion besteht aus

- einem Namen, mit dem sie aufgerufen werden kann;
- einer Spezifikation der formalen Parameter;
- evtl. einem Ergebnistyp;
- einem Anweisungsteil.

In C gilt:

$$\text{code}_R\ f\ \rho \quad = \quad \text{load c\_}f \quad = \quad \text{Anfangsadresse des Codes für } f$$

$\Longrightarrow$ Auch Funktions-Namen müssen in der Adress-Umgebung verwaltet werden!

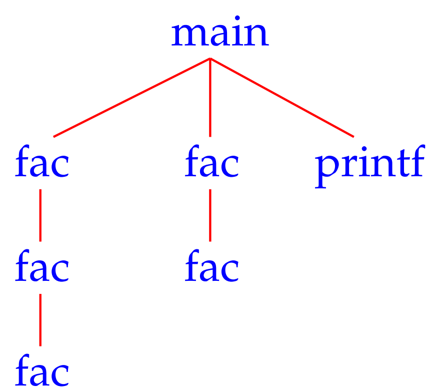
Beispiel:

```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x ∗ fac(x − 1);
}
```

```
main () {
    int n;
    n = fac(2) + fac(1);
    printf ("%d", n);
}
```

Zu einem Ausführungszeitpunkt können mehrere Instanzen (Aufrufe) der gleichen Funktion aktiv sein, d. h. begonnen, aber noch nicht beendet sein.

Der Rekursionsbaum im Beispiel:



Wir schließen:

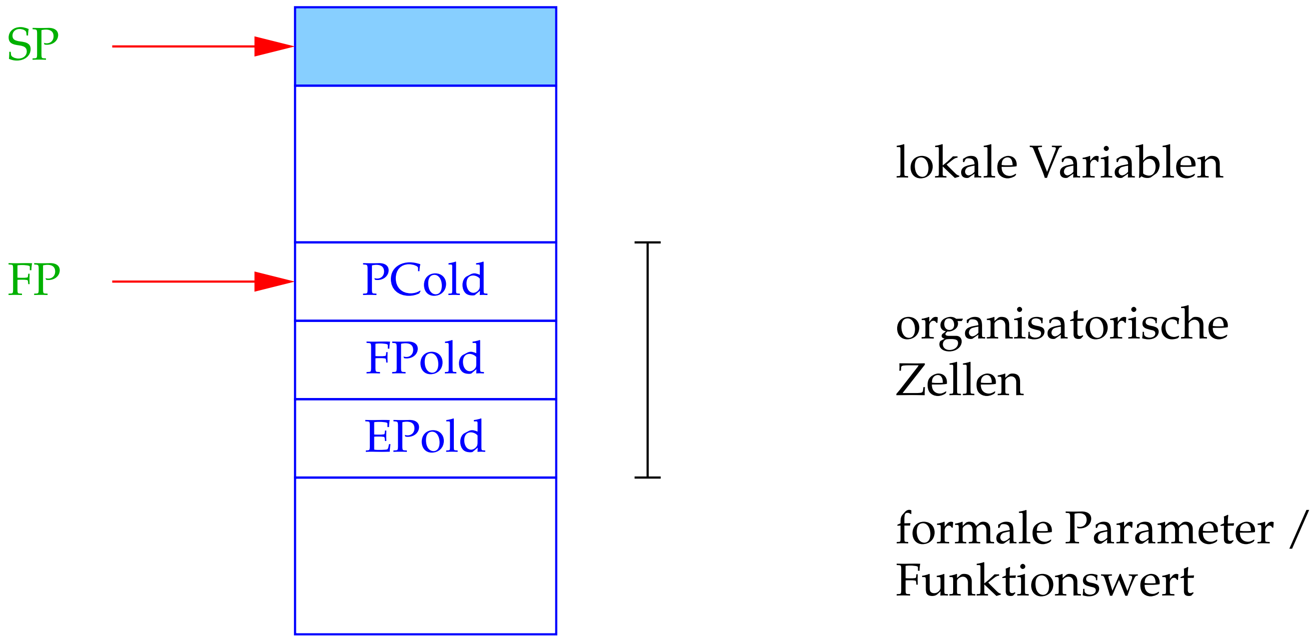
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee:

Lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.

In sequentiellen Programmiersprachen können diese Speicherbereiche auf dem Keller verwaltet werden. Deshalb heißen sie auch Keller-Rahmen (oder Stack Frame).

9.1 Speicherorganisation für Funktionen



FP $\widehat{=}$ Frame Pointer; zeigt auf die letzte organisatorische Zelle und wird zur Adressierung der formalen Parameter und lokalen Variablen benutzt.