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 $t \equiv \textbf{struct}\,\{t_1\; c_1; \ldots t_k\; c_k;\}$. 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:

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

Beispiel:

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

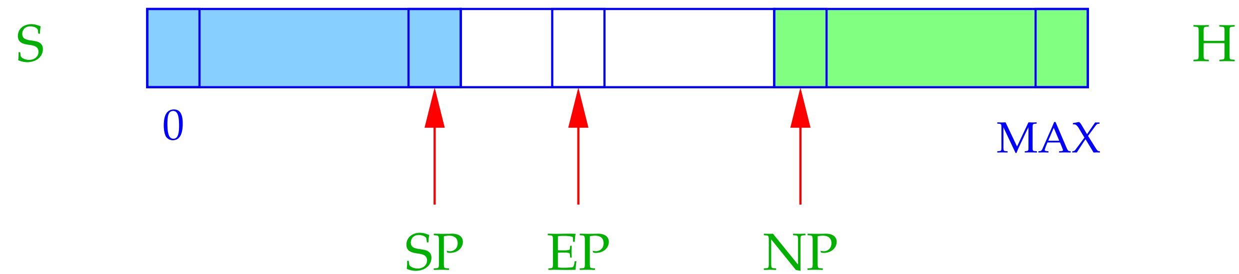
Dann ist

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

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 :-(.

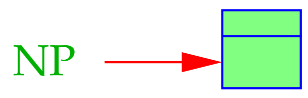
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

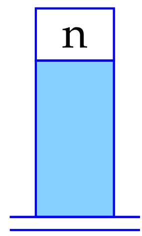
- Zeiger zu erzeugen, d.h. Zeiger auf Speicherzellen zu setzen; sowie
- Zeiger zu dereferenzieren, d. h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Es gibt zwei Arten, Zeiger zu erzeugen:

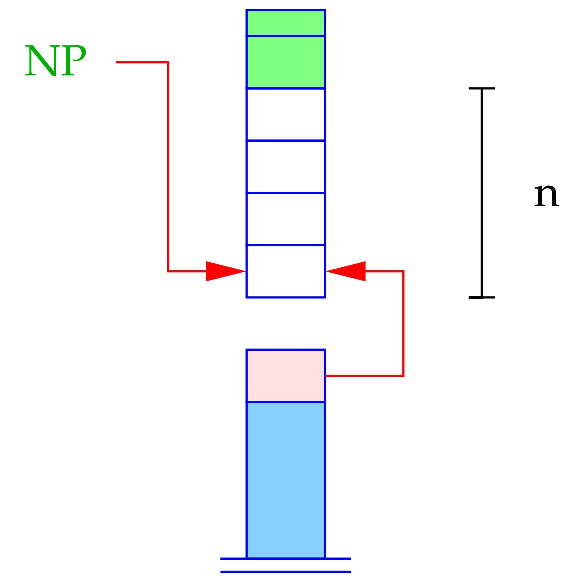
(1) Ein Aufruf von **malloc** liefert einen Zeiger auf eine Heap-Zelle:

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


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:

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

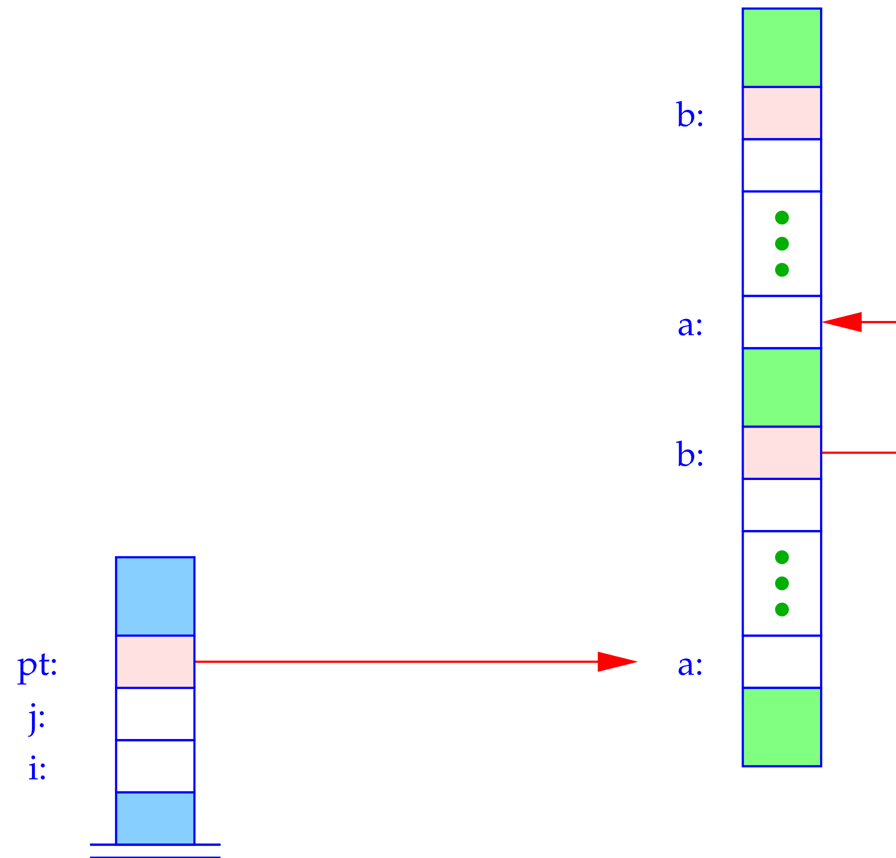
Beispiel: Betrachte für

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

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

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

$$\text{code}_L\ (e \to a)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{loadc}\ (\rho\, a)$$
$$\text{add}$$


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

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

$\text{code}_L\ e\ \rho$	=	$\text{code}_R\ ((pt \to b) \to a)\ \rho$	=	$\text{code}_R\ ((pt \to b) \to a)\ \rho$
		$\text{code}_R\ (i + 1)\ \rho$		loada 1
		loadc 1		loadc 1
		mul		add
		add		loadc 1
				mul
				add

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

$$\text{code}_R\ ((pt \to b) \to a)\ \rho \quad = \quad \text{code}_R\ (pt \to b)\ \rho \quad = \quad \text{loada } 3$$

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

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.

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

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

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

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

$$\text{code}_R\ (\text{malloc}\,(e))\ \rho \quad = \quad \text{code}_R\ e\ \rho$$

$$\text{new}$$

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

$$\text{code}_R\ (e_1 \Box e_2)\ \rho \quad = \quad \text{code}_R\ e_1\ \rho$$

$$\text{code}_R\ e_2\ \rho$$

$$\text{op} \qquad \text{op Befehl zu Operator `}\Box\text{'}$$

$$
\begin{array}{lcll}
\text{code}_\text{R}\ q\ \rho & = & \text{loadc q} & q \text{ Konstante} \\
\\
\text{code}_\text{R}\ (e_1 = e_2)\ \rho & = & \text{code}_\text{R}\ e_2\ \rho & \\
 & & \text{code}_\text{L}\ e_1\ \rho & \\
 & & \text{store} & \\
\\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{load} & \text{sonst}
\end{array}
$$

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:

$$\text{code}_L\ (*a)\ \rho \quad = \quad \text{code}_R\ a\ \rho \quad = \quad \text{code}_L\ a\ \rho \quad = \quad \text{loadc } 7$$

$$\text{code}\ s_1\ \rho \quad = \quad \text{loadc } 5$$

$$\text{loadc } 7$$

$$\text{store}$$

$$\text{pop}$$

Zur Übung übersetzen wir auch noch:

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

code $(s_2 s_3)\ \rho$ =

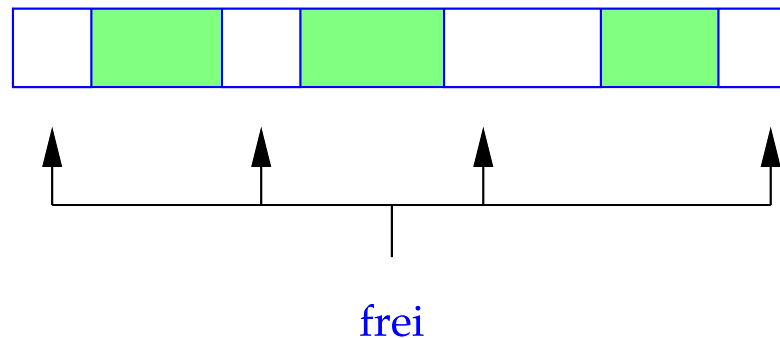
```
loadc 7
loadc 2
loadc 1      //  Skalierung
mul
add
loadc 17
store
pop          //  Ende von s2

loadc 5
loadc 17
load
loadc 3
loadc 1      //  Skalierung
mul
add
store
pop          //  Ende von s3
```

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.:

$$\text{code } \mathbf{free}\,(e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{pop}$$

  $\Longrightarrow$ 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:

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

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

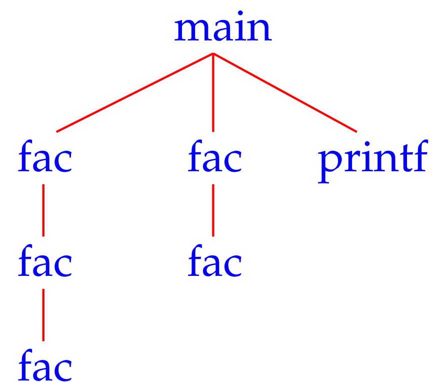
Beispiel:

```
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:

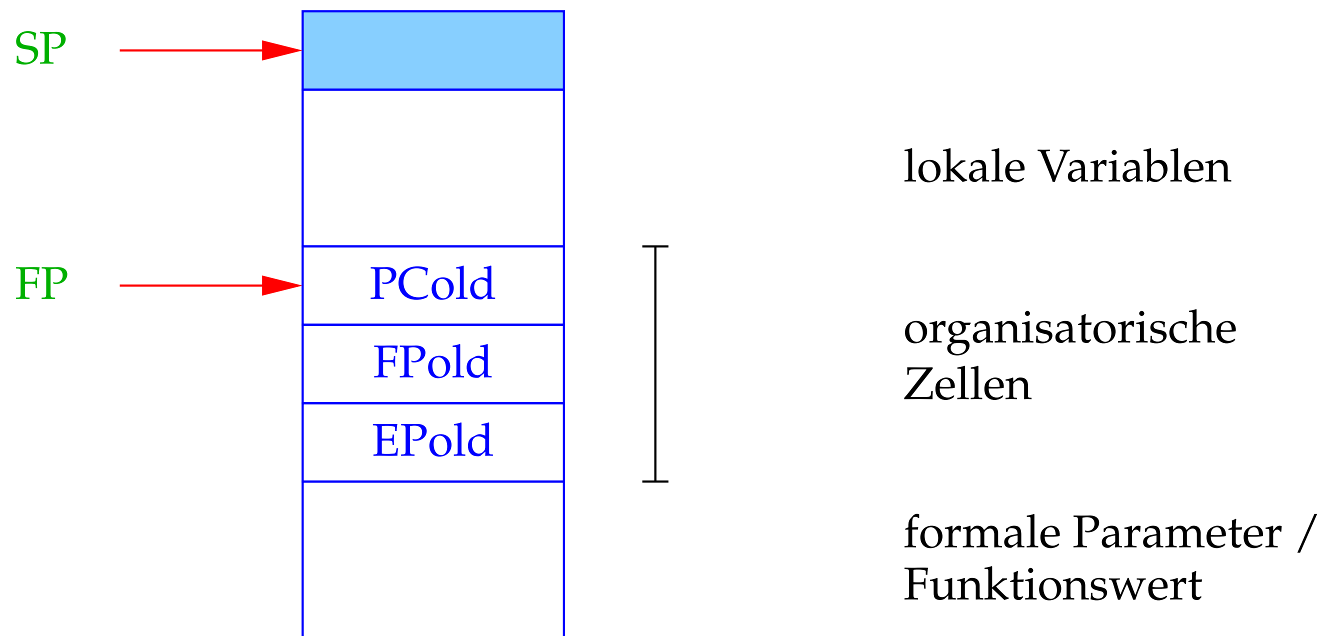
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

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.