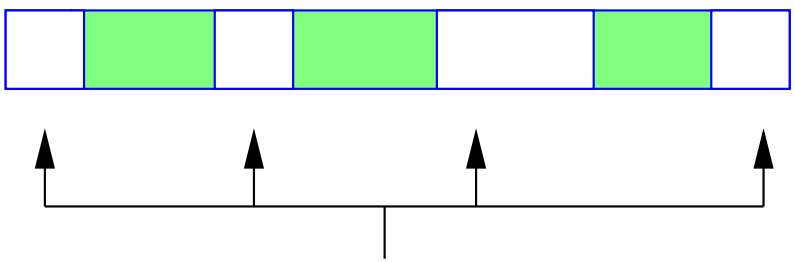
8 Freigabe von Speicherplatz

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.:

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

  $\Longrightarrow$ 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:

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

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

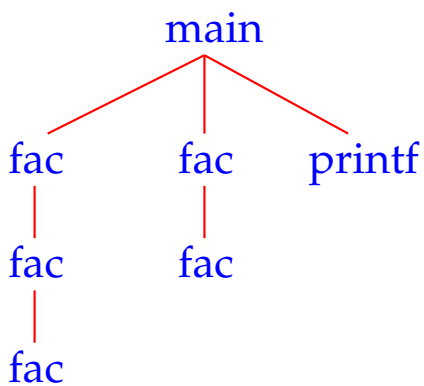
Beispiel:

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

```
int 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:

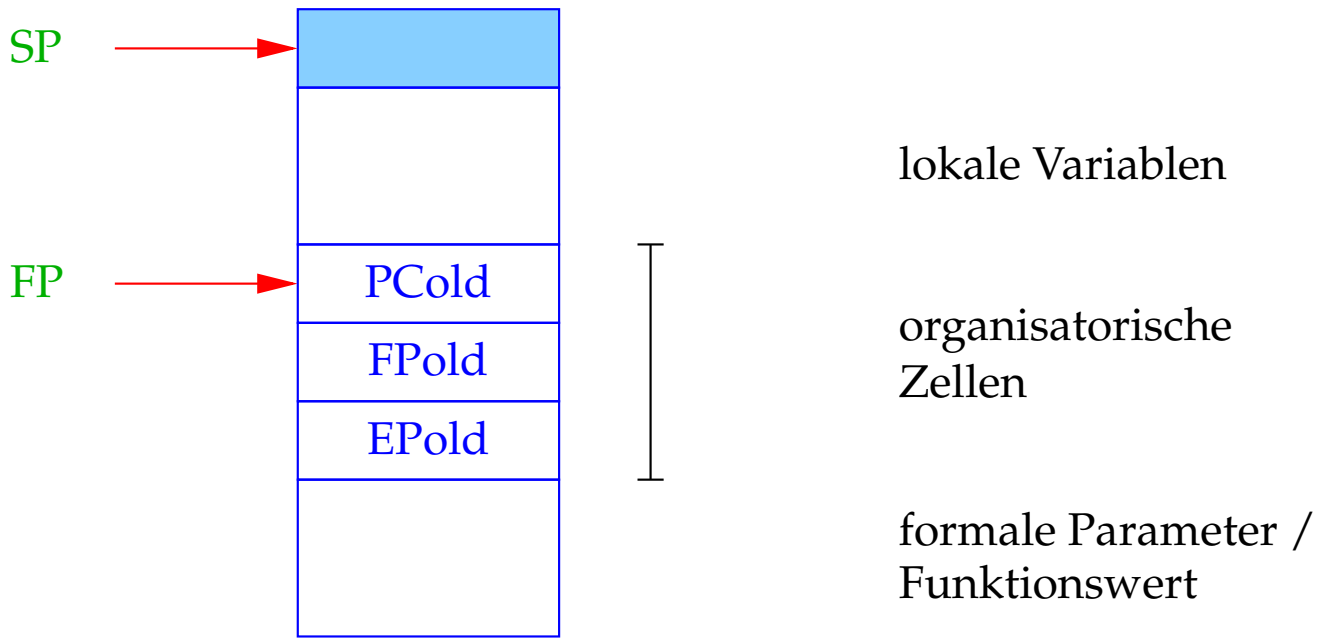
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

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.

Achtung:

- Die lokalen Variablen erhalten Relativadressen $+1, +2, \ldots$.
- Die formalen Parameter liegen unterhalb der organisatorischen Zellen und haben deshalb negative Adressen relativ zu FP :-)
- Diese Organisation ist besonders geeignet für Funktionsaufrufe mit variabler Argument-Anzahl wie z.B. `printf`.
- Den Speicherbereich für die Parameter recyclen wir zur Speicherung des Rückgabewerts der Funktion :-))

  Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

Achtung:

- Die lokalen Variablen erhalten Relativadressen $+1, +2, \ldots$.
- Die formalen Parameter liegen unterhalb der organisatorischen Zellen und haben deshalb negative Adressen relativ zu FP :-)
- Diese Organisation ist besonders geeignet für Funktionsaufrufe mit variabler Argument-Anzahl wie z.B. `printf`.
- Den Speicherbereich für die Parameter recyclen wir zur Speicherung des Rückgabewerts der Funktion :-))

  Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

Die organisatorischen Zellen speichern die Register-Inhalte, die nach dem Funktions-Aufruf restauriert werden müssen. Bei einem Funktions-Aufruf muss der FP in eine organisatorische Zelle gerettet werden.

Achtung:

- Die lokalen Variablen erhalten Relativadressen $+1, +2, \ldots$.
- Die formalen Parameter liegen unterhalb der organisatorischen Zellen und haben deshalb negative Adressen relativ zu FP :-)
- Diese Organisation ist besonders geeignet für Funktionsaufrufe mit variabler Argument-Anzahl wie z.B. `printf`.
- Den Speicherbereich für die Parameter recyclen wir zur Speicherung des Rückgabewerts der Funktion :-))

  Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

Die organisatorischen Zellen speichern die Register-Inhalte, die nach dem Funktions-Aufruf restauriert werden müssen. Bei einem Funktions-Aufruf muss der FP in eine organisatorische Zelle gerettet werden.

Unsere Übersetzungsaufgaben für Funktionen:

- Erzeuge Code für den Rumpf!
- Erzeuge Code für Aufrufe!

9.2 Bestimmung der Adress-Umgebung

Wir müssen zwei Arten von Variablen unterscheiden:

1. globale/externe, die außerhalb von Funktionen definiert werden;
2. lokale/interne/automatische (inklusive formale Parameter), die innerhalb von Funktionen definiert werden.

$$\Longrightarrow$$

Die Adress-Umgebung ρ ordnet den Namen Paare $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ zu.

Achtung:

- Tatsächlich gibt es i.a. weitere verfeinerte Abstufungen der Sichtbarkeit von Variablen.
- Bei der Übersetzung eines Programms gibt es i.a. für verschiedene Programmteile verschiedene Adress-Umgebungen!

Beispiel:

0

```
int i;
struct list {
    int info;
    struct list * next;
} * l;
```

1

```
int ith (struct list * x, int i) {
    if (i ≤ 1) return x →info;
    else return ith (x →next, i − 1);
}
```

2

```
int main () {
    int k;
    scanf ("%d", &i);
    scanlist (&l);
    printf ("\n\t%d\n", ith (l,i));
}
```

Vorkommende Adress-Umgebungen in dem Programm:

0 Außerhalb der Funktions-Definitionen:

$$\begin{array}{lrcl} \rho_0 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, _\text{ith}) \\ & \text{main} & \mapsto & (G, _\text{main}) \\ & & \ldots & \end{array}$$

1 Innerhalb von ith:

$$\begin{array}{lrcl} \rho_1 : & i & \mapsto & (L, -4) \\ & x & \mapsto & (L, -3) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, _\text{ith}) \\ & \text{main} & \mapsto & (G, _\text{main}) \\ & & \ldots & \end{array}$$

Achtung:

- Die formalen Parameter werden von rechts nach links ausgewertet !!
- Der erste Parameter liegt direkt unterhalb der organisatorischen Zellen :-)
- Für einen Prototypen $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$ setzen wir:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)$$

Achtung:

- Die formalen Parameter werden von rechts nach links ausgewertet !!
- Der erste Parameter liegt direkt unterhalb der organisatorischen Zellen :-)
- Für einen Prototypen $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$ setzen wir:

$$x_1 \mapsto (L, -2 - |\tau_1|) \qquad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|)$$

2 Innerhalb von main:

$$\begin{array}{lrcl} \rho_2 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & k & \mapsto & (L, 1) \\ & \text{ith} & \mapsto & (G, \text{_ith}) \\ & \text{main} & \mapsto & (G, \text{_main}) \\ & & \ldots & \end{array}$$

9.3 Betreten und Verlassen von Funktionen

Sei f die aktuelle Funktion, d. h. der Caller, und f rufe die Funktion g auf, d. h. den Callee.

Der Code für den Aufruf muss auf den Caller und den Callee verteilt werden.

Die Aufteilung kann nur so erfolgen, dass der Teil, der von Informationen des Callers abhängt, auch dort erzeugt wird (analog für den Callee).

Achtung:

Den Platz für die aktuellen Parameter kennt nur der Caller ...

Aktionen beim Betreten von g:

1. Bestimmung der aktuellen Parameter
2. Retten von FP, EP } mark
3. Bestimmung der Anfangsadresse von g
4. Setzen des neuen FP
5. Retten von PC und Sprung an den Anfang von g } call
6. Setzen des neuen EP } enter
7. Allokieren der lokalen Variablen } alloc

} stehen in f

} stehen in g

Aktionen bei Beenden des Aufrufs:

1. Abspeichern des Rückgabewerts
2. Rücksetzen der Register FP, EP, SP
3. Rücksprung in den Code von f, d. h. Restauration des PC

} return

4. Aufräumen des Stack

} slide

Damit erhalten wir für einen Aufruf für eine Funktion mit mindestens einem Paremeter und einem Rückgabewert:

$$
\begin{aligned}
\text{code}_\text{R}\ g(e_1, \ldots, e_n)\ \rho \quad = \quad & \text{code}_\text{R}\ e_n\ \rho \\
& \ldots \\
& \text{code}_\text{R}\ e_1\ \rho \\
& \text{mark} \\
& \text{code}_\text{R}\ g\ \rho \\
& \text{call} \\
& \text{slide}\ (\text{m} - 1)
\end{aligned}
$$

wobei m der Platz für die aktuellen Parameter ist .

Beachte:

- Von jedem Ausdruck, der als aktueller Parameter auftritt, wird jeweils der R-Wert berechnet $\Longrightarrow$ Call-by-Value-Parameter-Übergabe.
- Die Funktion g kann auch ein Ausdruck sein, dessen R-Wert die Anfangs-Adresse der aufzurufenden Funktion liefert ...
- Ähnlich deklarierten Feldern, werden Funktions-Namen als konstante Zeiger auf Funktionen aufgefasst. Dabei ist der R-Wert dieses Zeigers gleich der Anfangs-Adresse der Funktion.
- Achtung! Für eine Variable **int** $(*g)()$; sind die beiden Aufrufe

  $$(*g)() \qquad \text{und} \qquad g()$$

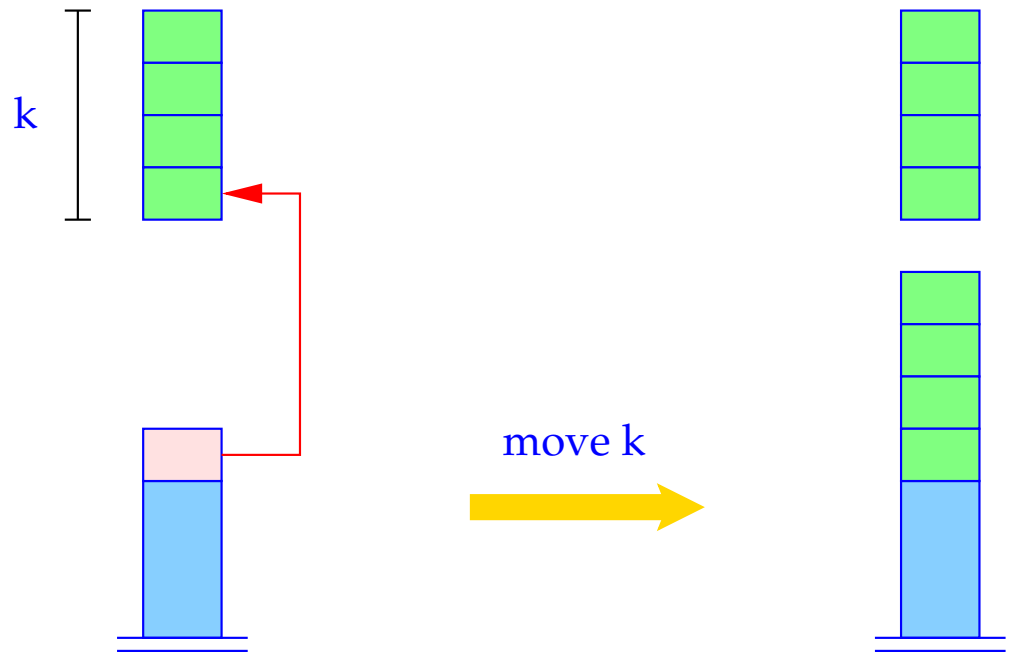
  äquivalent! Per Normalisierung, muss man sich hier vorstellen, werden Dereferenzierungen eines Funktions-Zeigers ignoriert :-)

- Bei der Parameter-Übergabe von Strukturen werden diese kopiert.

Folglich:

$$
\begin{array}{lcll}
\text{code}_\text{R}\ f\ \rho & = & \text{loadc}\ (\rho\, f) & f \text{ ein Funktions-Name} \\
\text{code}_\text{R}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & e \text{ ein Funktions-Zeiger} \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{move k} & e \text{ eine Struktur der Größe } k
\end{array}
$$

Dabei ist:




```
for (i = k-1; i≥0; i--)
        S[SP+i] = S[S[SP]+i];
SP = SP+k–1;
```