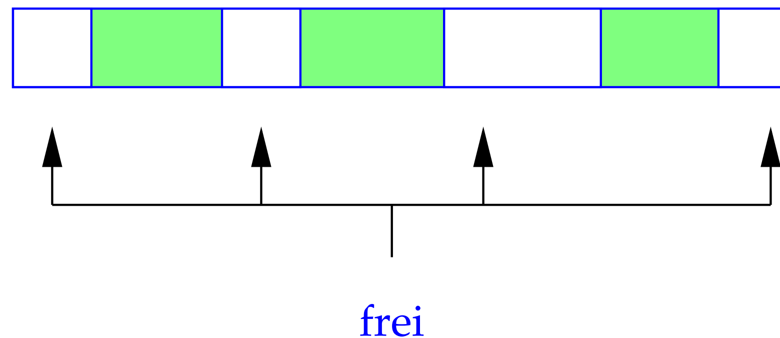
$\text{code}\ (s_2 s_3)\ \rho \quad = \quad$

```
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 s2         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.:

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

  $\Longrightarrow$ 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:

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

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

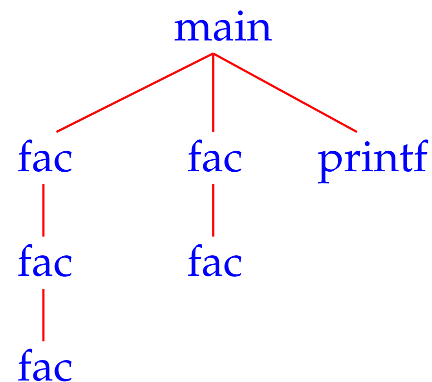
Beispiel:

```
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:

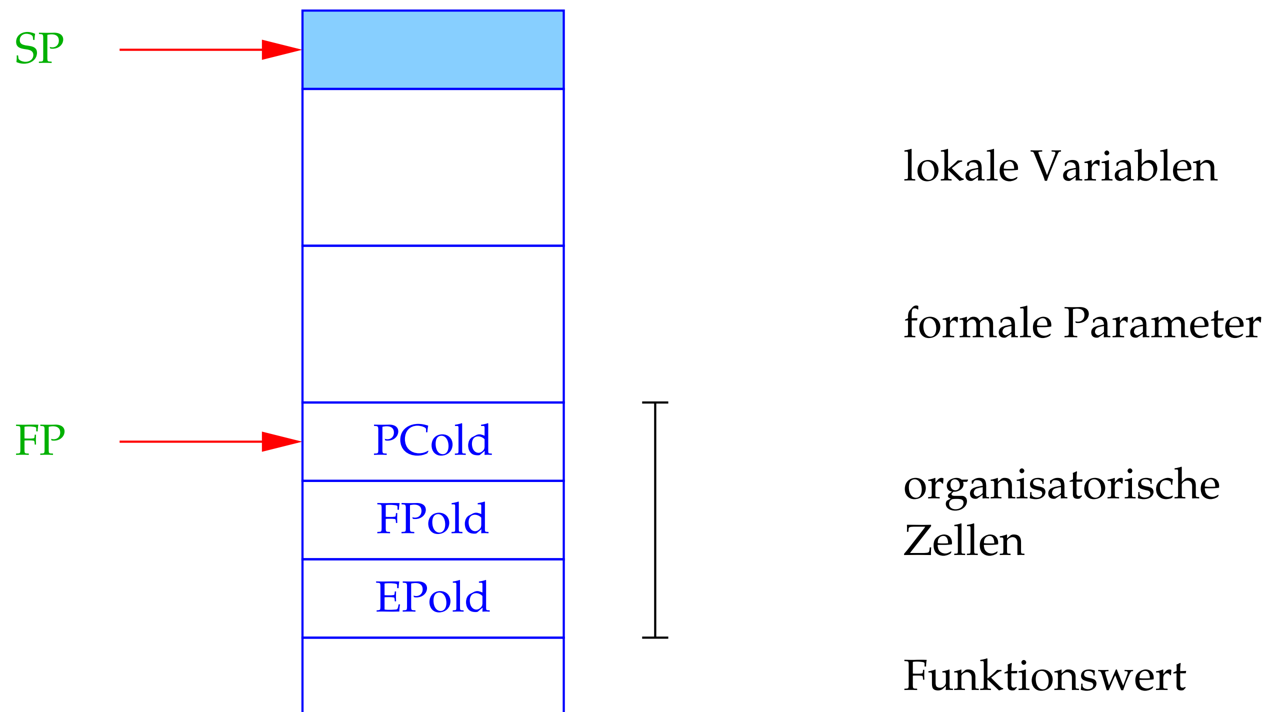
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

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.

- Die lokalen Variablen und formalen Parameter adressieren wir relativ zu FP.
- Bei einem Funktions-Aufruf muss der FP in eine organisatorische Zelle gerettet werden.
- Weiterhin müssen gerettet werden:
  - die Fortsetzungsadresse nach dem Aufruf;
  - der aktuelle EP.

Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

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.

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   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:

$$\begin{array}{lrcl} \rho_1 : & i & \mapsto & (L, 2) \\ & x & \mapsto & (L, 1) \\ & l & \mapsto & (G, 2) \\ & \text{ith} & \mapsto & (G, _\text{ith}) \\ & \text{main} & \mapsto & (G, _\text{main}) \\ & & \ldots & \end{array}$$

2 Innerhalb von main:

$$\begin{array}{rrcl} \rho_2 : & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & k & \mapsto & (L, 1) \\ & \text{ith} & \mapsto & (G, _\text{ith}) \\ & \text{main} & \mapsto & (G, _\text{main}) \\ & & \dots & \end{array}$$

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).

Aktionen beim Betreten von g:

1. Retten von FP, EP } mark
2. Bestimmung der aktuellen Parameter
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

(Schritte 1–5: stehen in f; Schritte 6–7: stehen in g)

Aktionen beim Verlassen von g:

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

Damit erhalten wir für einen Aufruf:

$$\begin{array}{lcl} \text{code}_\text{R}\; g(e_1, \ldots, e_n)\; \rho & = & \text{mark} \\ & & \text{code}_\text{R}\; e_1\; \rho \\ & & \ldots \\ & & \text{code}_\text{R}\; e_n\; \rho \\ & & \text{code}_\text{R}\; g\; \rho \\ & & \text{call m} \end{array}$$

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 ...