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 Rückgabe werts 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.

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 $(tag, a) \in \{G, L\} \times \mathbb{Z}$ 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:

$\boxed{0}$

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

$\boxed{1}$

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

$\boxed{2}$

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

Vorkommende Adress-Umgebungen in dem Programm:

$\boxed{0}$ Vor den Funktions-Definitionen:

$$\begin{array}{llll} \rho_0 : & i & \mapsto & (G,1) \\ & l & \mapsto & (G,2) \\ & & \ldots & \end{array}$$

$\boxed{1}$ Innerhalb von ith:

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

Achtung:

- Die aktuellen 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 aktuellen 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:

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

$\boxed{2}$ 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		stehen in f
4. Setzen des neuen FP		
5. Retten von PC und Sprung an den Anfang von g	call	
6. Setzen des neuen EP	enter	stehen in g
7. Allokieren der lokalen Variablen	alloc	

Aktionen bei Beenden des Aufrufs:

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

return

slide

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

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

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 ...

- 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

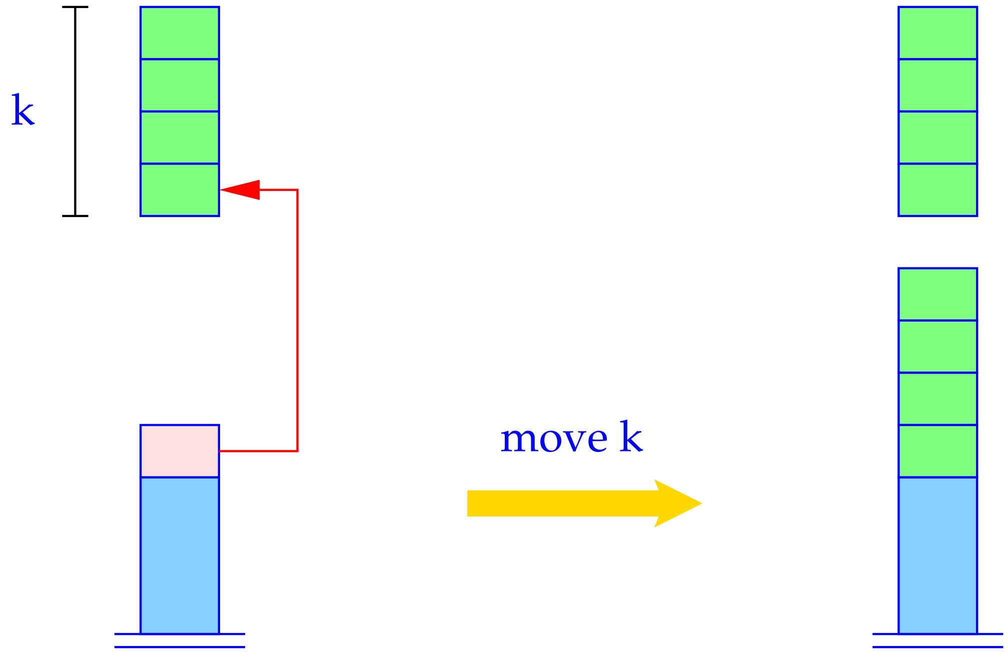
$$(*g)() \qquad \text{und} \qquad g()$$

  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:

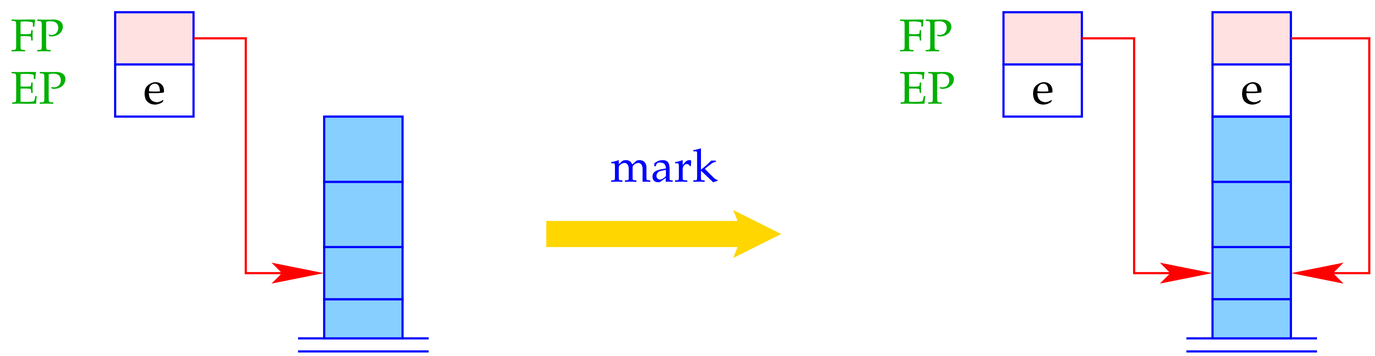


for (i = k-1; i≥0; i--)

    S[SP+i] = S[S[SP]+i];

SP = SP+k–1;

Der Befehl mark legt Platz für Rückgabewert und organisatorische Zellen an und rettet FP und EP.

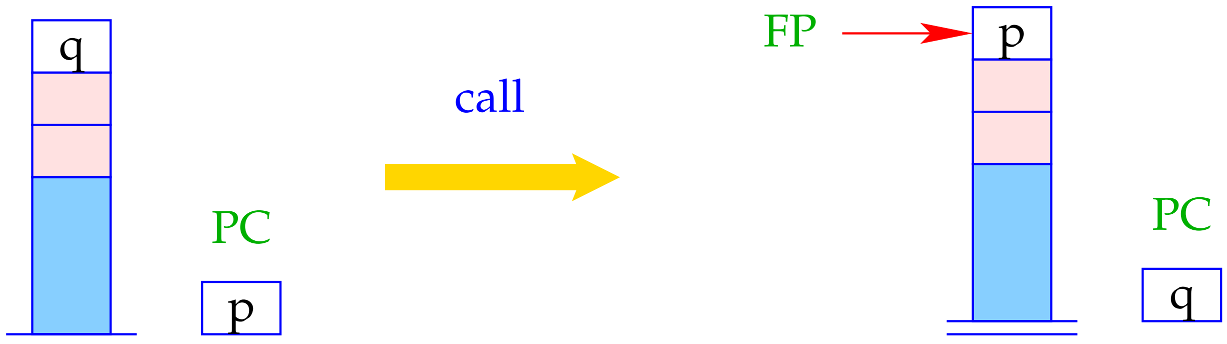


S[SP+1] = EP;

S[SP+2] = FP;

SP = SP + 2;

Der Befehl call rettet die Fortsetzungs-Adresse und setzt FP und PC auf die aktuellen Werte.



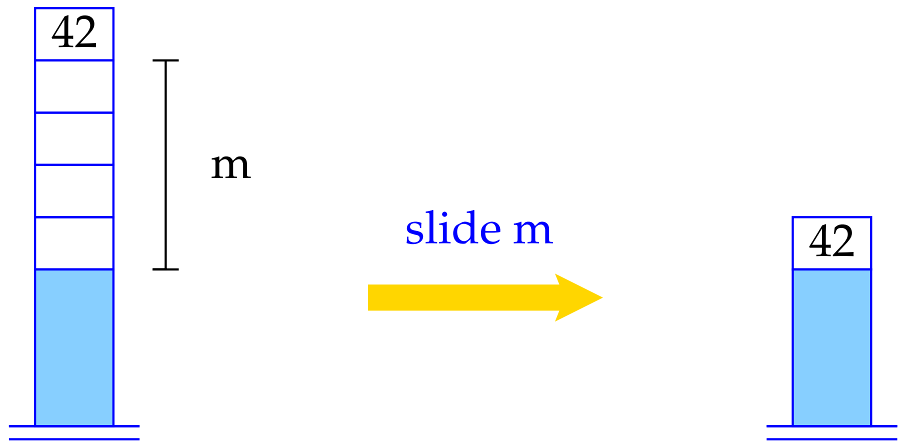
tmp = S[SP];

S[SP] = PC;

FP = SP;

PC = tmp;

Der Befehl slide kopiert den Rückgabewert an die korrekte Stelle:



tmp = S[SP];

SP = SP-m;

S[SP] = tmp;

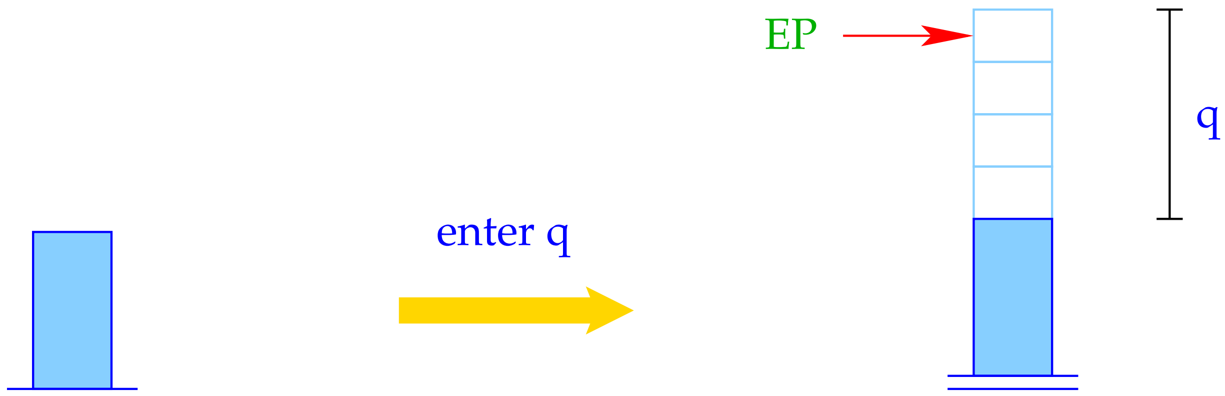
Entsprechend übersetzen wir eine Funktions-Definition:

$$\text{code}\quad t\ \mathrm{f}\ (specs)\{V\_defs\quad ss\}\quad \rho\quad =$$

```
_f:   enter q        //  setzen des EP
      alloc k        //  Anlegen der lokalen Variablen
      code ss ρf
      return         //  Verlassen der Funktion
```

wobei	q	=	$max + \mathrm{k}$ wobei
	max	=	maximale Länge des lokalen Kellers
	k	=	Platz für die lokalen Variablen
	ρ_f	=	Adress-Umgebung für f
			// *berücksichtigt specs, V_defs* und ρ

Der Befehl enter q setzt den EP auf den neuen Wert. Steht nicht mehr genügend Platz zur Verfügung, wird die Programm-Ausführung abgebrochen.

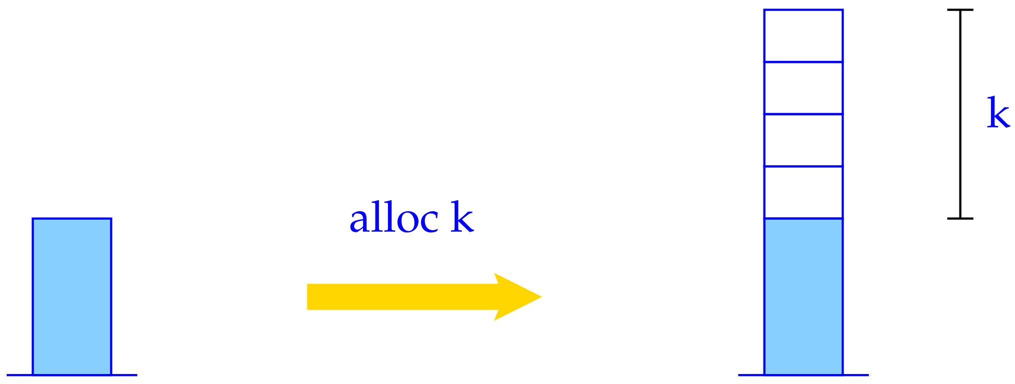


EP = SP + q;

if (EP $\geq$ NP)

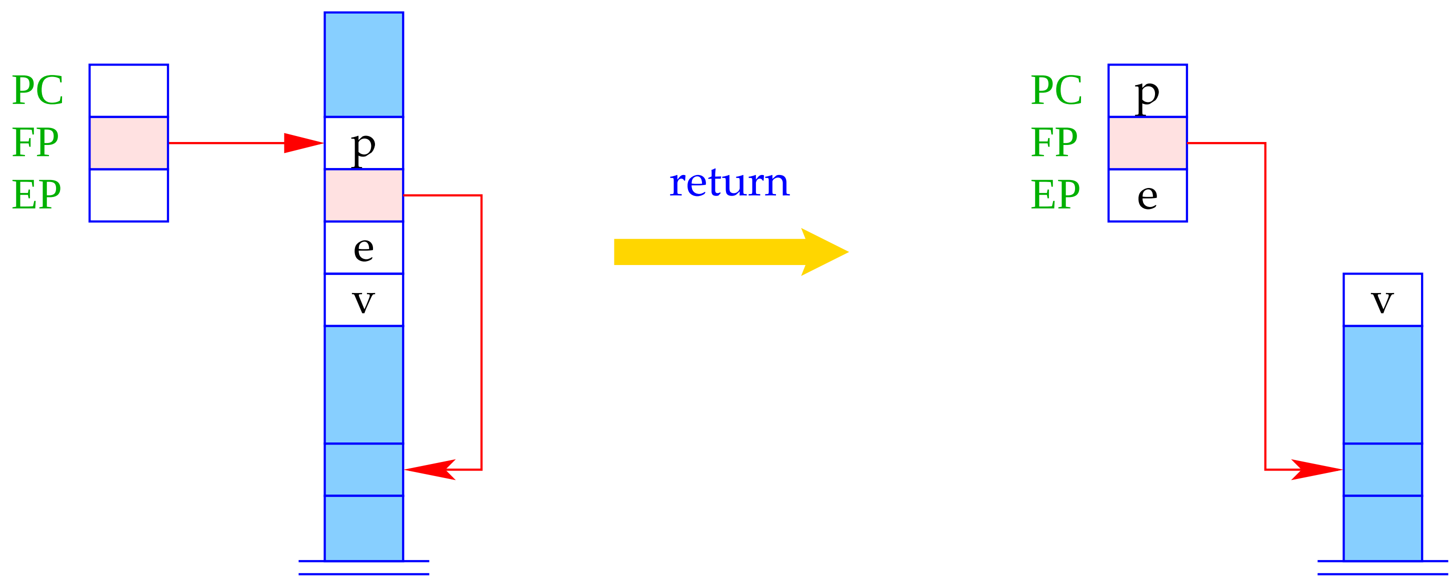
    Error (“Stack Overflow”);

Der Befehl alloc k reserviert auf dem Keller Platz für die lokalen Variablen.



SP = SP + k;

Der Befehl return gibt den aktuellen Keller-Rahmen auf. D.h. er restauriert die Register PC, EP und FP und hinterlässt oben auf dem Keller den Rückgabe-Wert.



PC = S[FP]; EP = S[FP-2];

if (EP $\geq$ NP) Error (“Stack Overflow”);

SP = FP-3; FP = S[SP+2];

9.4 Zugriff auf Variablen, formale Parameter und Rückgabe von Werten

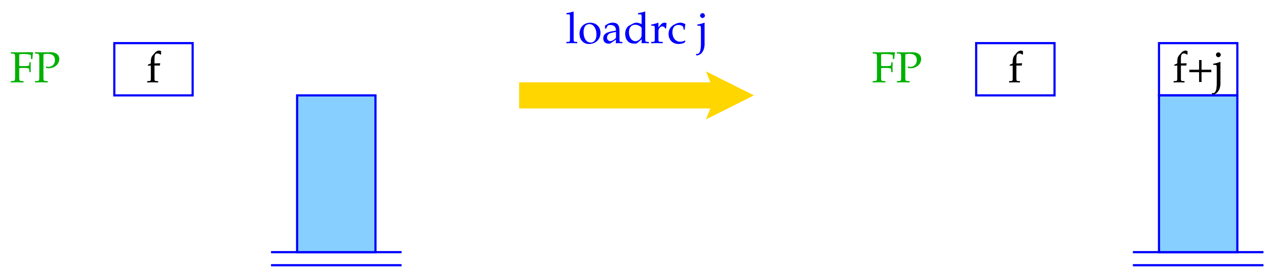
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

Darum modifizieren wir code_L für Variablen-Namen.

Für $\quad \rho\, x = (tag, j) \quad$ definieren wir

$$\text{code}_\text{L}\; x\; \rho = \begin{cases} \text{loadc j} & tag = G \\ \text{loadrc j} & tag = L \end{cases}$$

Der Befehl loadrc j berechnet die Summe von FP und j.



SP++;

S[SP] = FP+j;

Als Optimierung führt man analog zu loada j und storea j die Befehle loadr j und storer j ein:

```
loadr j   =   loadrc j
              load

storer j  =   loadrc j;
              store
```