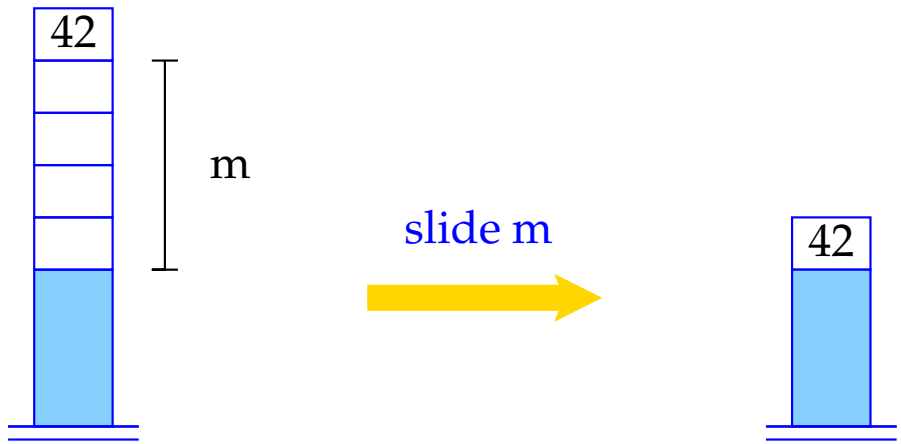
Der Befehl slide kopiert den Rückgabewert an die korrekte Stelle:




```
tmp = S[SP];
SP = SP-m;
S[SP] = tmp;
```

Entsprechend übersetzen wir eine Funktions-Definition:

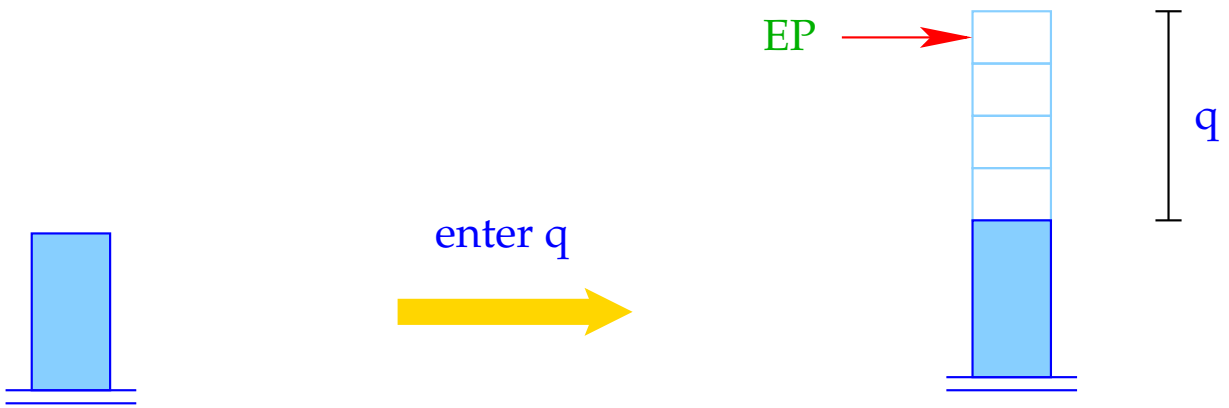
$$\text{code}\quad t\ \mathrm{f}\ (specs)\{V\_defs\quad ss\}\quad \rho\quad =$$

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

wobei

q	=	$max + k$	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.

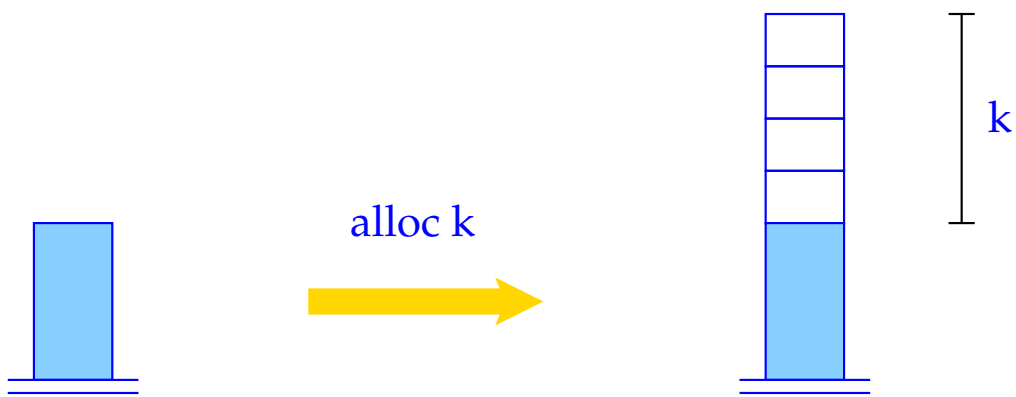


EP = SP + q;

if (EP $\geq$ NP)

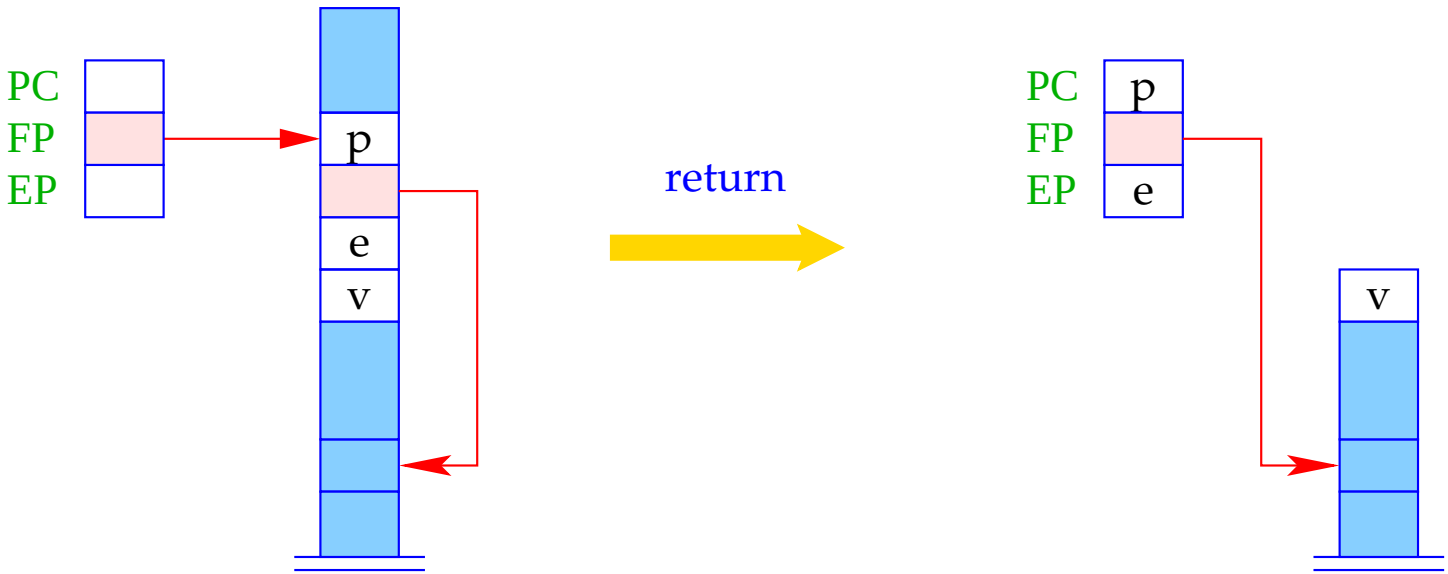
Error (“Stack Overflow”);

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

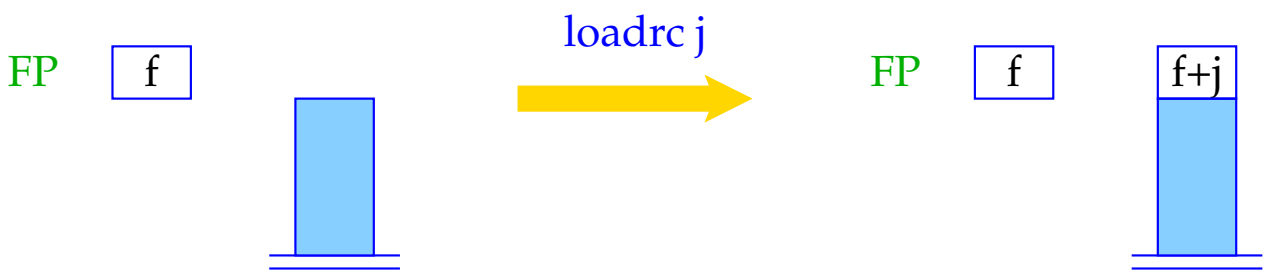
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

Darum modifizieren wir code_L für Variablen-Namen.

Für $\rho\ x = (tag, j)$ 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

Der Code für **return** e; entspricht einer Zuweisung an eine Variable mit Relativadresse -3.

$$\begin{array}{lcl} \text{code } \textbf{return}\, e;\ \rho & = & \text{code}_R\ e\ \rho \\ & & \text{storer -3} \\ & & \text{return} \end{array}$$

Beispiel: Für die Funktion

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

erzeugen wir:

```
_fac:   enter q          loadc 1          A:   loadr -3              mul
        alloc 0          storer -3             loadr -3              storer -3
        loadr -3         return                loadc 1               return
        loadc 0          jump B                sub              B:   return
        leq                                    mark
        jumpz A                                loadc _fac
                                               call
                                               slide 0
```

Dabei ist $\rho_{\text{fac}} : x \mapsto (L, -3)$ und $q = 1 + 1 + 3 = 5$.

10 Übersetzung ganzer Programme

Vor der Programmausführung gilt:

$$\mathrm{SP} = -1 \qquad \mathrm{FP} = \mathrm{EP} = 0 \qquad \mathrm{PC} = 0 \qquad \mathrm{NP} = \mathrm{MAX}$$

Sei $p \equiv V\_defs \quad F\_def_1 \ldots F\_def_n$, ein Programm, wobei F_def_i eine Funktion f_i definiert, von denen eine main heißt.

Der Code für das Programm p enthält:

- Code für die Funktions-Definitionen F_def_i;
- Code zum Anlegen der globalen Variablen;
- Code für den Aufruf von main();
- die Instruktion halt.

Dann definieren wir:

$$
\begin{array}{lll}
\text{code } p\ \emptyset & = & \text{enter } (\mathrm{k}+4) \\
 & & \text{alloc } (\mathrm{k}+1) \\
 & & \text{mark} \\
 & & \text{loadc \_main} \\
 & & \text{call} \\
 & & \text{slide k} \\
 & & \text{halt} \\
 & \_\mathrm{f}_1\text{:} & \text{code } F\_def_1\ \rho \\
 & & \vdots \\
 & \_\mathrm{f}_n\text{:} & \text{code } F\_def_n\ \rho
\end{array}
$$

wobei $\emptyset \quad \widehat{=}$ leere Adress-Umgebung;

$\rho \quad \widehat{=}$ globale Adress-Umgebung;

$\mathrm{k} \quad \widehat{=}$ Platz für globale Variablen