

# Die Übersetzung funktionaler Programmiersprachen

## 11 Die Sprache PuF

Wir betrachten hier nur die Mini-Sprache PuF (“Pure Functions”). Insbesondere verzichten wir (vorerst) auf:

- Seiteneffekte;
- Datenstrukturen;

Ein Programm ist ein Ausdruck  $e$  der Form:

$$\begin{aligned}
e ::= & b \mid x \mid (\square_1 e) \mid (e_1 \square_2 e_2) \\
& \mid (\mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_3) \\
& \mid (e' \ e_0 \ \dots \ e_{k-1}) \mid (\mathbf{fn} \ x_0, \dots, x_{k-1} \Rightarrow e) \\
& \mid (\mathbf{let} \ x_1 = e_1; \dots; x_n = e_n \ \mathbf{in} \ e_0) \\
& \mid (\mathbf{letrec} \ x_1 = e_1; \dots; x_n = e_n \ \mathbf{in} \ e_0)
\end{aligned}$$

Ein Ausdruck ist somit:

- ein Basiswert, eine Variable, eine Operator-Anwendung oder ein bedingter Ausdruck;
- eine Funktions-**Anwendung**;
- eine Funktion – d.h. aus einem Funktionsrumpf entstanden mithilfe von **Abstraktion** der formalen Parameter;
- ein **let**-Ausdruck, der **lokal Variablen-Definitionen** einführt, oder
- ein **letrec**-Ausdruck, der lokal **rekursive** Variablen-Definitionen einführt.

Als Basis-Typen erlauben wir der Einfachkeit halber nur **int**.

## Beispiel:

Die folgende allseits bekannte Funktion berechnet die Fakultät:

```
fac    =    fn x ⇒ if x ≤ 1 then 1
                else x · fac (x - 1)
```

Wie üblich, setzen wir nur da Klammern, wo sie zum Verständnis erforderlich sind :-)

## Achtung:

Wir unterscheiden zwei Arten der Parameter-Übergabe:

**CBV:** **Call-by-Value** – die aktuellen Parameter werden ausgewertet **bevor** der Rumpf der Funktion ausgewertet wird (genau wie bei **C ...**);

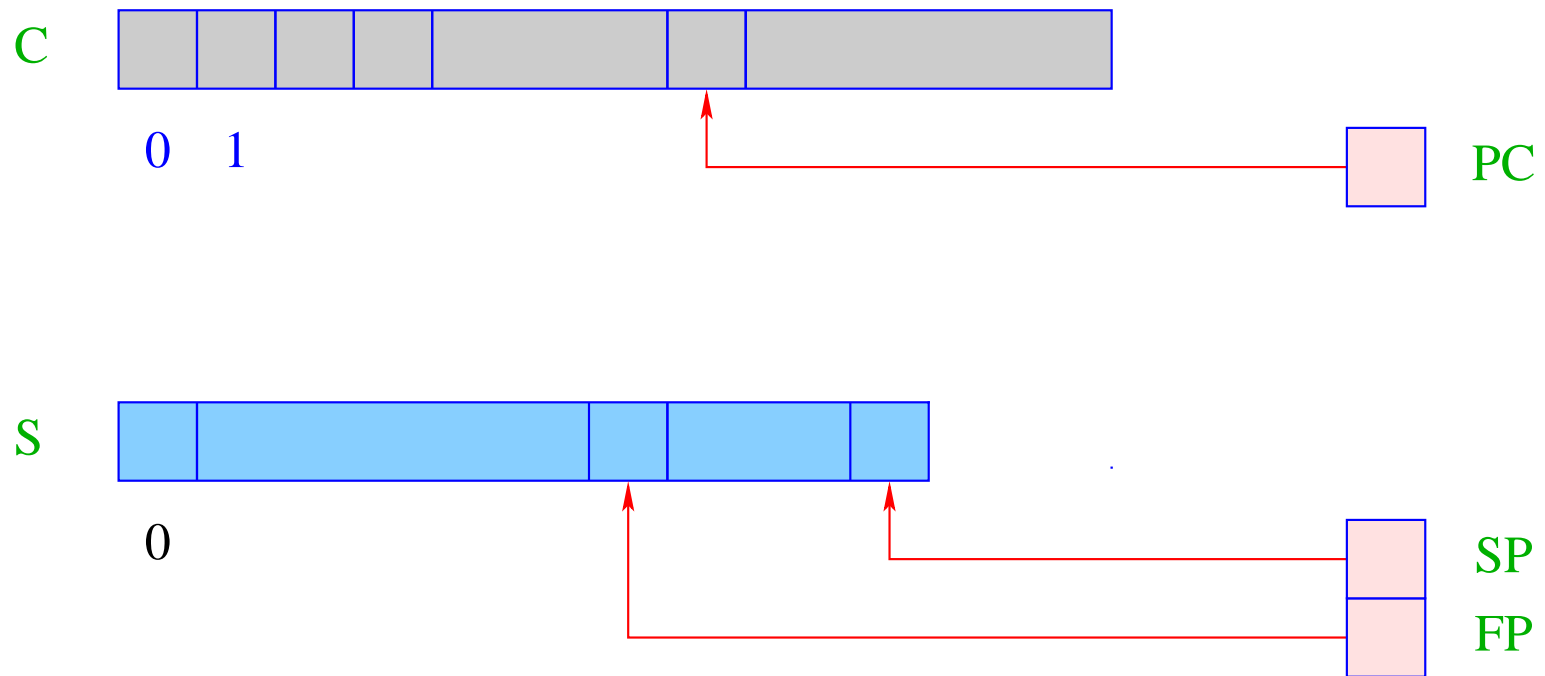
**CBN:** **Call-by-Need** – die aktuellen Parameter werden erst ausgewertet, wenn ihr Wert benötigt wird  $\implies$  spart **manchmal** Arbeit :-)

## Beispiel:

```
let  fac  =  ... ;  
     foo  =  fn x, y => x  
in   foo 1 (fac 1000)
```

- Die Funktion `foo` greift nur auf ihr erstes Argument zu.
- Die Auswertung des zweiten Arguments wird bei **CBN** vermieden **:-)**
- Weil wir bei **CBN** nicht sicher sein können, ob der Wert einer Variablen bereits ermittelt wurde oder nicht, müssen wir **vor** jedem Variablen-Zugriff überprüfen, ob der Wert bereits vorliegt **:-)**
- Liegt der Wert noch nicht vor, muss seine Berechnung angestoßen werden.

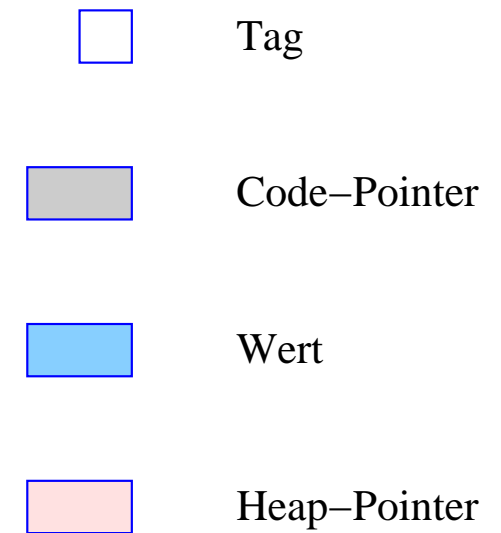
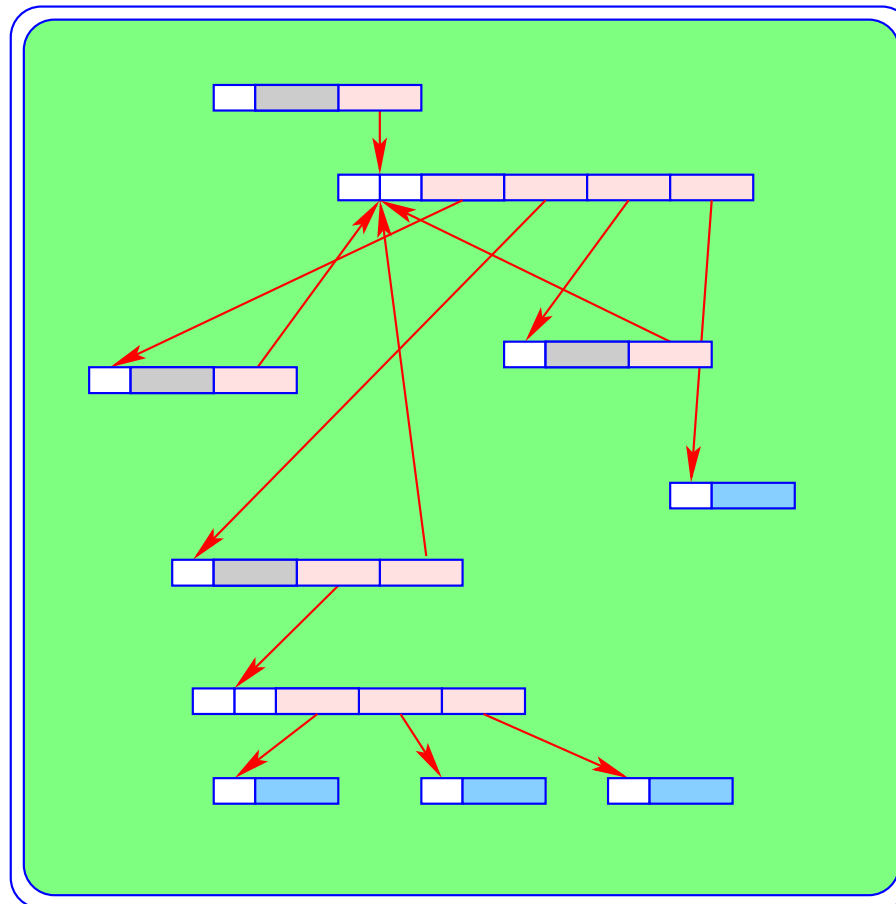
## 12 Architektur der MaMa:



... das sind die uns bereits bekannten Datenstrukturen:

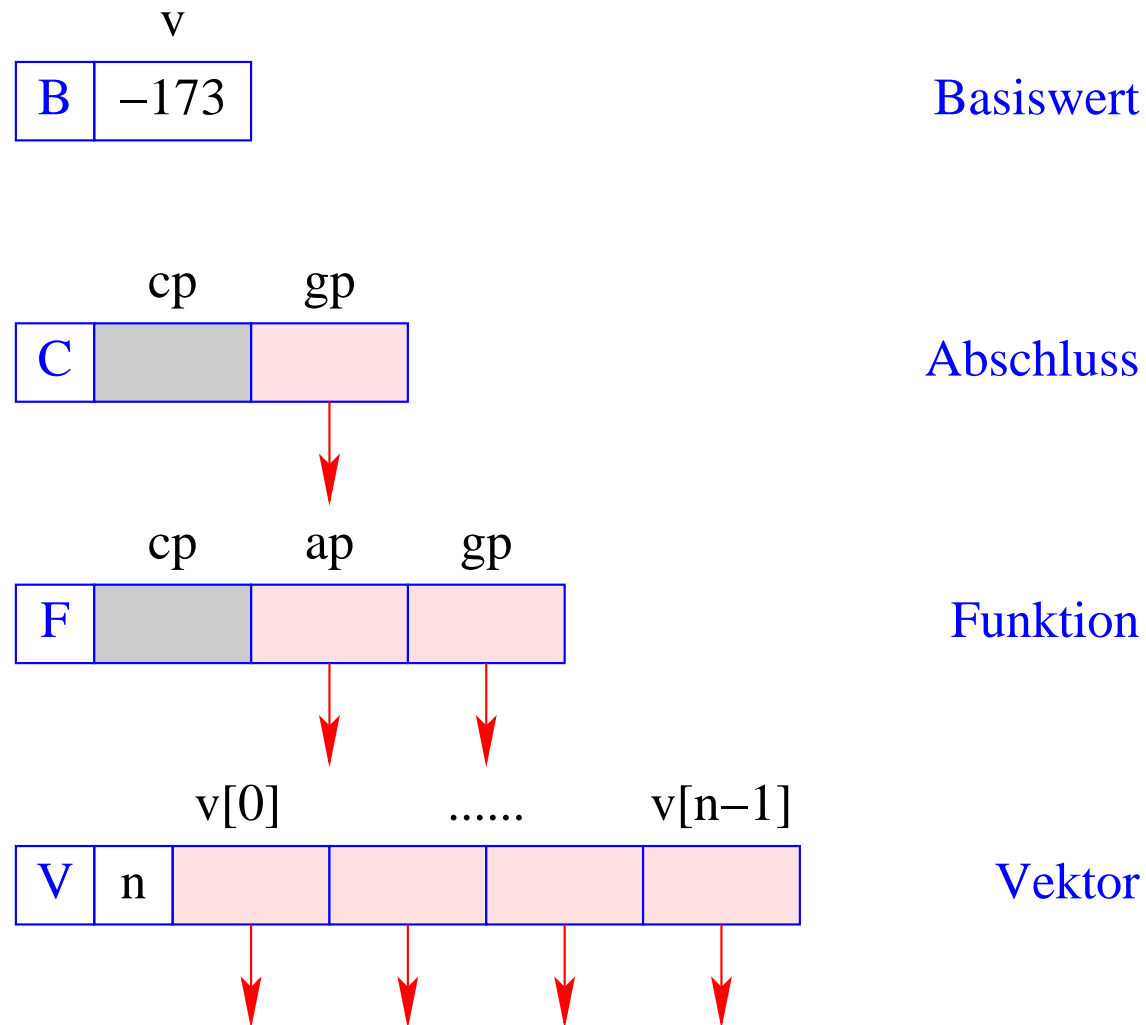
- C** = Code-Speicher – enthält MaMa-Programm;  
jede Zelle enthält einen Befehl;
- PC** = Program Counter – zeigt auf nächsten auszuführenden Befehl;
- S** = Runtime-Stack;  
jede Zelle kann einen Basis-Wert oder eine Adresse aufnehmen;
- SP** = Stack-Pointer – zeigt auf oberste belegte Zelle;
- FP** = Frame-Pointer – zeigt auf den aktuellen Kellerrahmen.

Weiterhin benötigen wir eine Halde **H**:





... die wir nun als einen **abstrakten Datentyp** auffassen, in dem wir Daten-Objekte der folgenden Form ablegen können:



Die Funktion  $\text{new}(tag, args)$  des Laufzeit-Systems der **MaMa** erzeugt ein entsprechendes Objekt in **H** und liefert eine Referenz darauf zurück.

Im Folgenden unterscheiden wir drei Arten von Code für einen Ausdruck  $e$ :

- $\text{code}_V e$  — berechnet den Wert von  $e$ , legt ihn in der Halde an und liefert auf dem Keller eine Referenz darauf zurück (der Normal-Fall);
- $\text{code}_B e$  — berechnet den Wert von  $e$ , und liefert ihn direkt oben auf dem Keller zurück (geht nur für Basistypen);
- $\text{code}_C e$  — wertet den Ausdruck  $e$  **nicht** aus, sondern legt einen **Abschluss** für  $e$  in der Halde an und liefert auf dem Stack eine Referenz auf diesen Abschluss zurück  $\implies$  benötigen wir zur Implementierung von **CBN**.

Wir betrachten zuerst Übersetzungsschemata für die ersten beiden Code-Arten.

## 13 Einfache Ausdrücke

Ausdrücke, die nur Konstanten, Operator-Anwendungen und bedingte Verzweigungen enthalten, werden wie Ausdrücke in imperativen Sprachen übersetzt:

$$\begin{aligned} \text{code}_B b \rho \text{kp} &= \text{loadc } b \\ \text{code}_B (\square_1 e) \rho \text{kp} &= \text{code}_B e \rho \text{kp} \\ &\quad \text{op}_1 \\ \text{code}_B (e_1 \square_2 e_2) \rho \text{kp} &= \text{code}_B e_1 \rho \text{kp} \\ &\quad \text{code}_B e_2 \rho (\text{kp} + 1) \\ &\quad \text{op}_2 \end{aligned}$$

$\text{code}_B(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho \text{ kp} =$

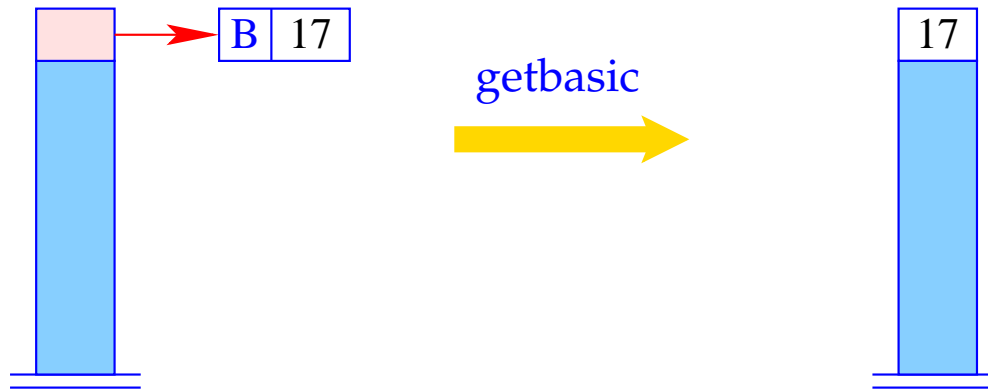
- $\text{code}_B e_0 \rho \text{ kp}$
- $\text{jumpz A}$
- $\text{code}_B e_1 \rho \text{ kp}$
- $\text{jump B}$
- A:  $\text{code}_B e_2 \rho \text{ kp}$
- B: ...

## Bemerkungen:

- $\rho$  bezeichnet die aktuelle Adress-Umgebung, in der der Ausdruck übersetzt wird.
- Das Extra-Argument  $kp$  zählt die Länge des lokalen Kellers mit  $\implies$  benötigen wir später zur Adressierung der Variablen.
- Die Instruktionen  $op_1$  und  $op_2$  implementieren die Operatoren  $\square_1$  und  $\square_2$ , so wie in der  $CMa$  die Operatoren  $neg$  und  $add$  die Negation bzw. die Addition implementieren.
- Für alle übrigen Ausdrücke berechnen wir erst den Wert im Heap und dereferenzieren dann:

$$code_B e \rho kp = code_V e \rho kp \text{ getbasic}$$

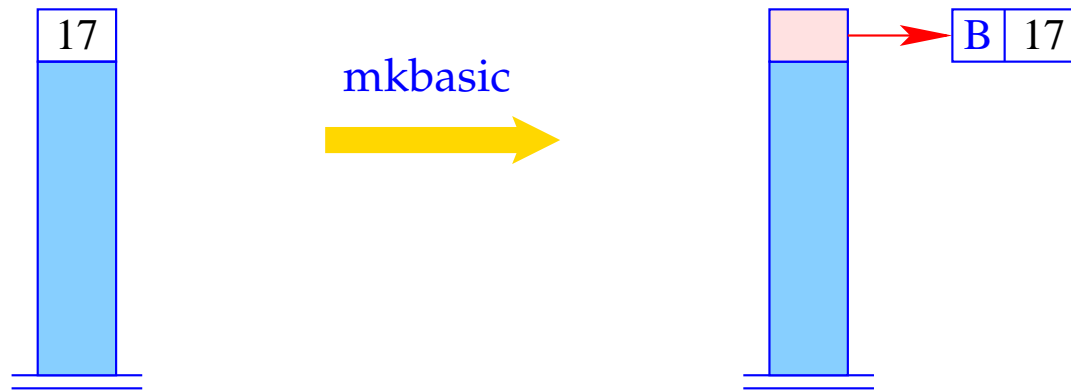
Dabei ist:



```
if (H[S[SP]] != (B,_))  
    Error "not basic!";  
else  
    S[SP] = H[S[SP]].v;
```

Für  $\text{code}_V$  und einfache Ausdrücke finden wir analog:

$$\begin{aligned} \text{code}_V b \rho \text{kp} &= \text{loadc } b; \text{mkbasic} \\ \text{code}_V (\square_1 e) \rho \text{kp} &= \text{code}_B e \rho \text{kp} \\ &\quad \text{op}_1; \text{mkbasic} \\ \text{code}_V (e_1 \square_2 e_2) \rho \text{kp} &= \text{code}_B e_1 \rho \text{kp} \\ &\quad \text{code}_B e_2 \rho (\text{kp} + 1) \\ &\quad \text{op}_2; \text{mkbasic} \\ \text{code}_V (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rho \text{kp} &= \text{code}_B e_0 \rho \text{kp} \\ &\quad \text{jumpz } A \\ &\quad \text{code}_V e_1 \rho \text{kp} \\ &\quad \text{jump } B \\ &\quad A: \text{code}_V e_2 \rho \text{kp} \\ &\quad B: \dots \end{aligned}$$



$S[SP] = \text{new } (B, S[SP]);$



## 14 Der Zugriff auf Variablen

**Beispiel:** Betrachte die Funktion  $f$  :

$$\mathbf{fn} \ a \Rightarrow \mathbf{let} \ b = a * a \\ \mathbf{in} \ b + c$$

Die Funktion  $f$  benutzt die **globale** Variable  $c$  sowie die **lokalen** Variablen  $a$  (als formalem Parameter) und  $b$  (eingeführt durch **let**).

Der Wert einer globalen Variable wird beim **Anlegen** der Funktion bestimmt (**Statische Bindung!**) und später nur nachgeschlagen.

## Idee:

- Die Bindungen der globalen Variablen verwalten wir in einem Vektor im Heap (**Global Vector**).
- Beim Anlegen eines F-Objekts wird der Global Vector für die Funktion ermittelt und in der **gp**-Komponente abgelegt.
- Bei der Auswertung eines Ausdrucks zeigt das (**neue**) Register **GP** (**Global Pointer**) auf den aktuellen Global Vector.
- Die lokalen Variablen verwalten wir dagegen auf dem Keller.

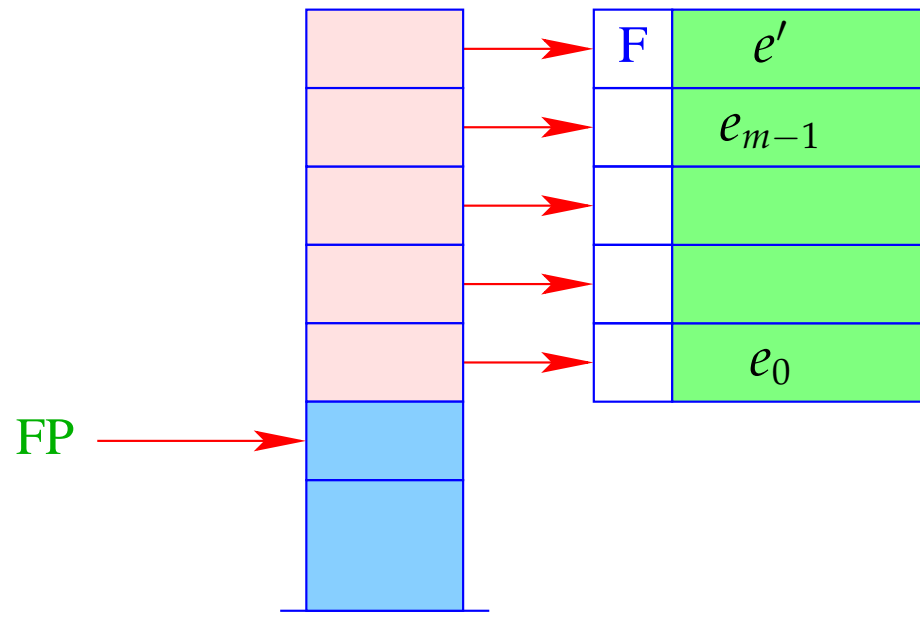
Adress-Umgebungen haben darum die Form:

$$\rho : Vars \rightarrow \{L, G\} \times \mathbb{Z}$$

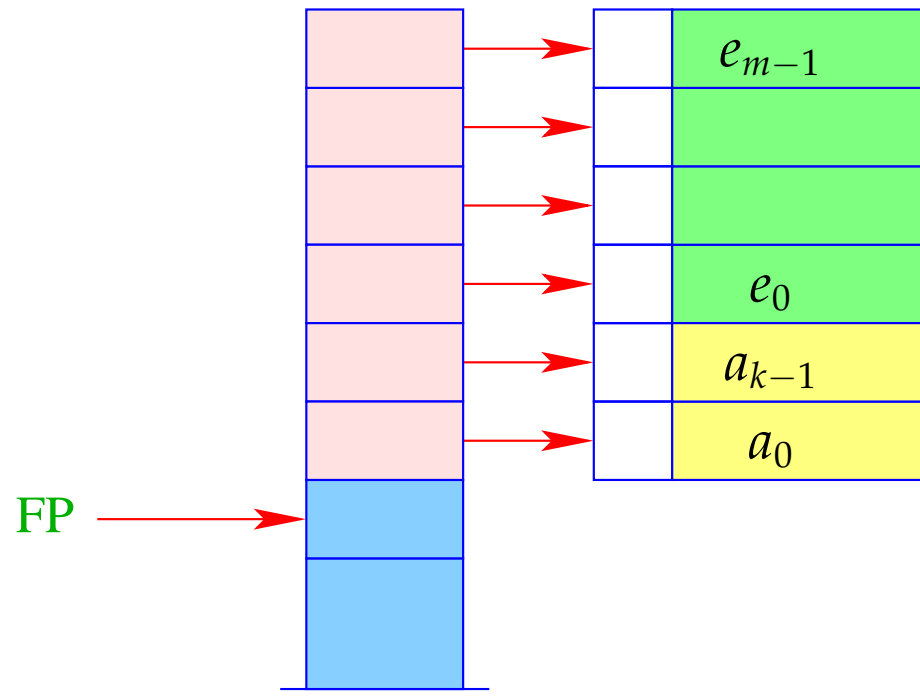
- Die globalen Variablen numerieren wir einfach geeignet durch.
- Für die Adressierung der lokalen Variablen gibt es zwei Möglichkeiten.

Sei  $e \equiv e' e_0 \dots e_{m-1}$  die Anwendung einer Funktion  $e'$  auf Argumente  $e_0, \dots, e_{m-1}$ .

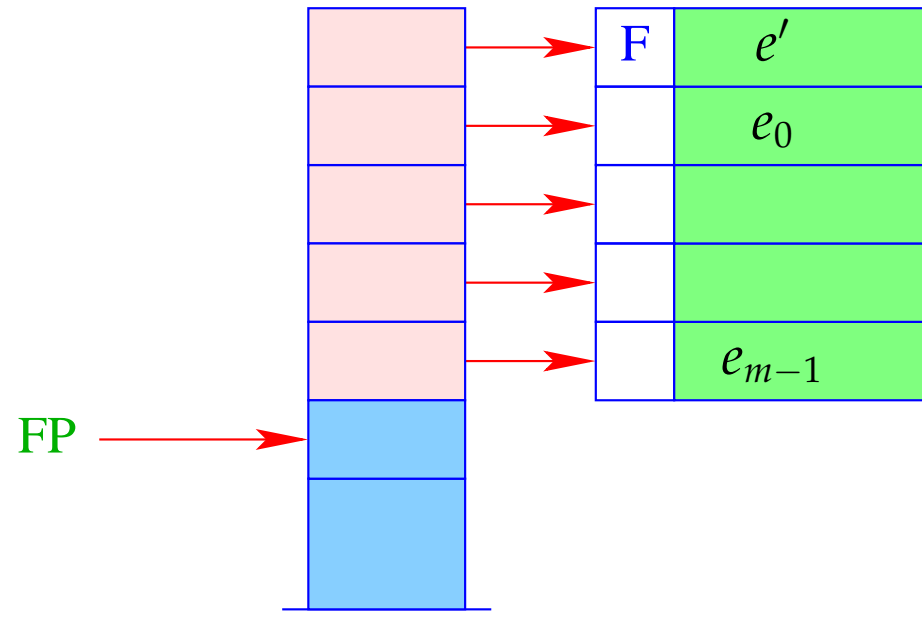
### Mögliche Kellerorganisation:



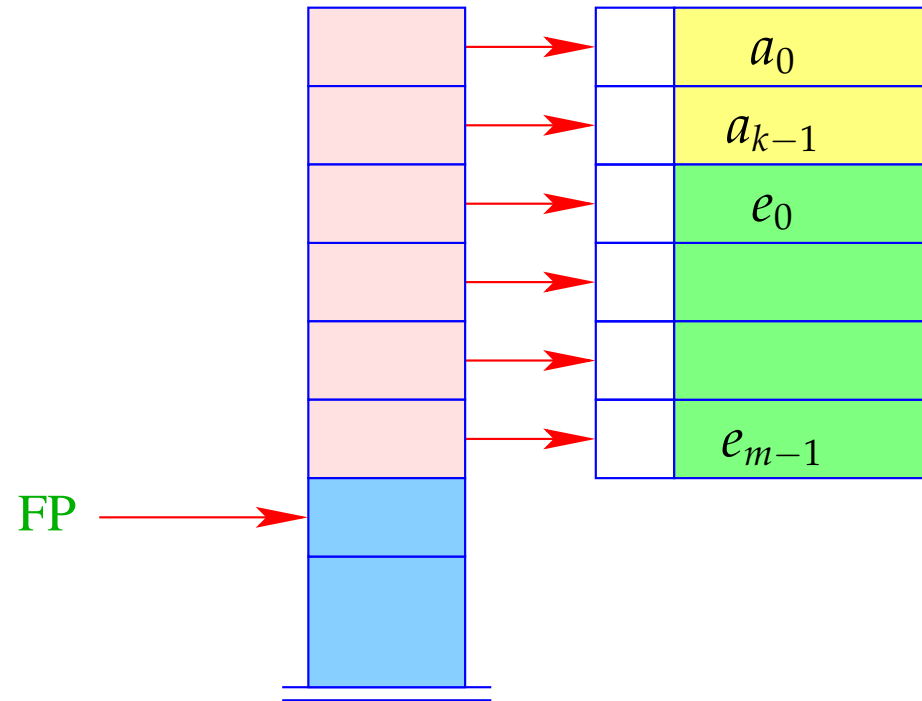
- + Adressierung der Parameter kann relativ zu FP erfolgen :-)
- Stellt sich heraus, dass sich  $e'$  zu einer Funktion evaluiert, die bereits partiell auf aktuelle Parameter  $a_0, \dots, a_{k-1}$  angewendet ist, müssen diese unterhalb von  $e_0$  in den Keller hinein gefrickelt werden :-)



Alternative:



- + Die weiteren Argumente  $a_0, \dots, a_{k-1}$  wie auch die lokalen Variablen können einfach oben auf den Keller gelegt werden :-)



- Adressierung relativ zu FP ist aber leider nicht mehr möglich ... ;-?

## Ausweg:

- Wir adressieren relativ zum Stackpointer **SP !!!**
- Leider ändert sich der Stackpointer während der Programm-Ausführung ...

