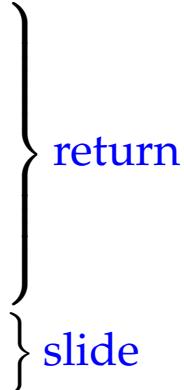


## Aktionen bei Beenden des Aufrufs:

1. Abspeichern des Rückgabewerts
  2. Rücksetzen der Register **FP**, **EP**, **SP**
  3. Rücksprung in den Code von  $f$ , d. h.  
Restauration des **PC**
  4. Aufräumen des Stack
- 
- return
- slide

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

```
codeR g(e1,...,en) ρ = codeR en ρ  
...  
codeR e1 ρ  
mark  
codeR g ρ  
call  
slide (m - 1)
```

wobei **m** der Platz für die aktuellen Parameter ist.

Beachte:

- Von jedem Ausdruck, der als aktueller Parameter auftritt, wird jeweils der **R-Wert** berechnet  $\implies$  **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

$(*g)()$  und  $g()$

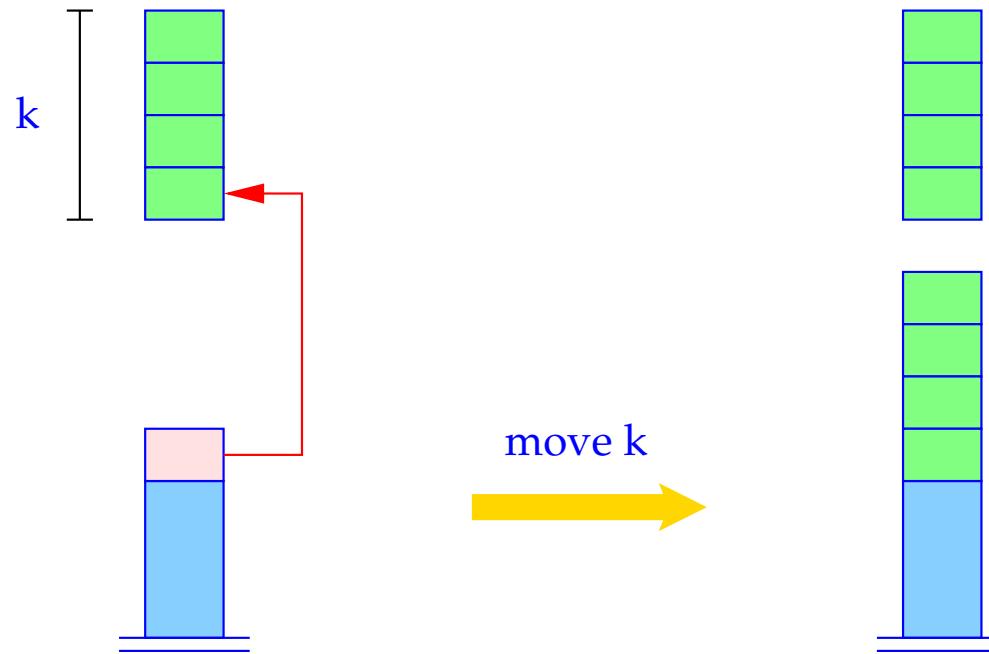
äquivalent! Per **Normalisierung**, muss man sich hier vorstellen, werden Dereferenzierungen eines Funktions-Zeigers ignoriert :-)

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

Folglich:

$$\begin{array}{lll}
 \text{code}_R f \rho & = & \text{loadc } (\rho f) & f \text{ ein Funktions-Name} \\
 \text{code}_R (*e) \rho & = & \text{code}_R e \rho & e \text{ ein Funktions-Zeiger} \\
 \text{code}_R e \rho & = & \text{code}_L e \rho \\
 & & \text{move k} & e \text{ eine Struktur der Größe } k
 \end{array}$$

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.

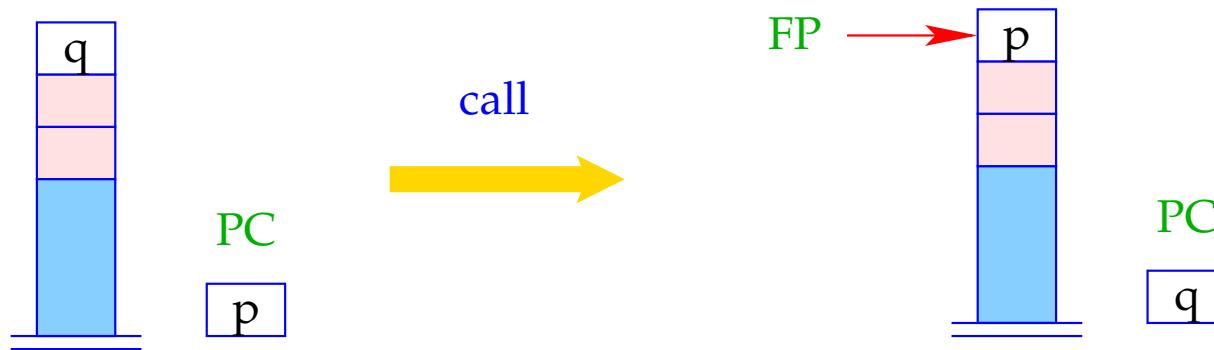


$S[SP+1] = EP;$

$S[SP+2] = FP;$

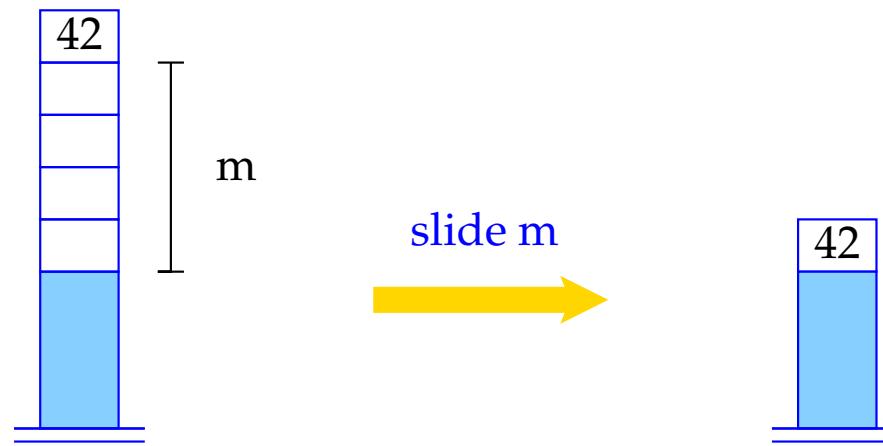
$SP = SP + 2;$

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:



$\text{tmp} = S[\text{SP}];$

$\text{SP} = \text{SP}-m;$

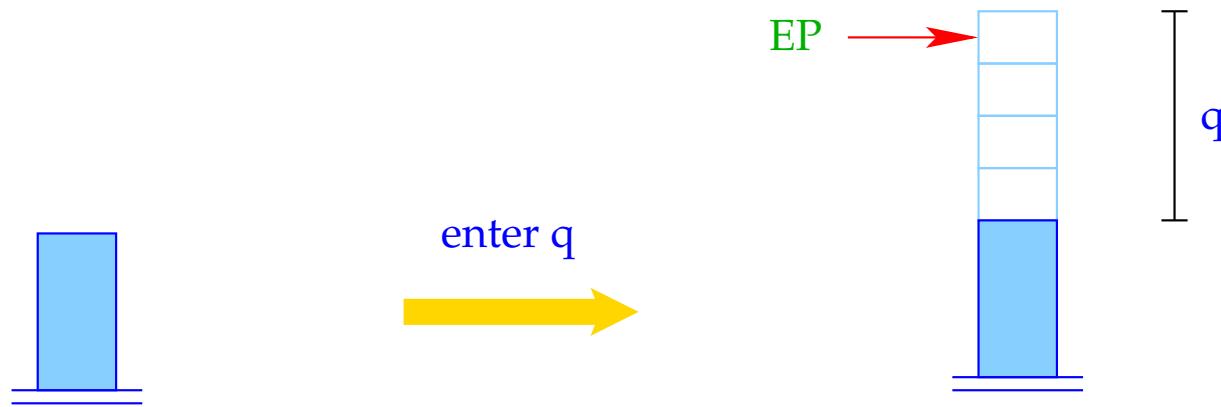
$S[\text{SP}] = \text{tmp};$

Entsprechend übersetzen wir eine Funktions-Definition:

```
code t f (specs){V_defs ss} ρ =  
  _f: enter q      // setzen des EP  
        alloc k      // Anlegen der lokalen Variablen  
        code ss ρf  
        return         // Verlassen der Funktion
```

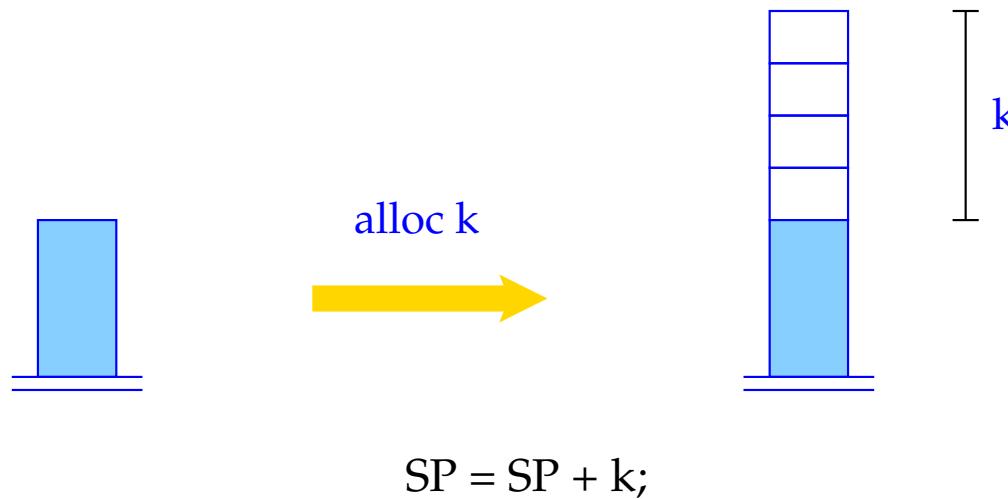
wobei  $q = max + k$  wobei  
 $max =$  maximale Länge des lokalen Kellers  
 $k =$  Platz für die lokalen Variablen  
 $\rho_f =$  Adress-Umgebung für  $f$   
// berücksichtigt  $specs$ ,  $V\_defs$  und  $\rho$

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

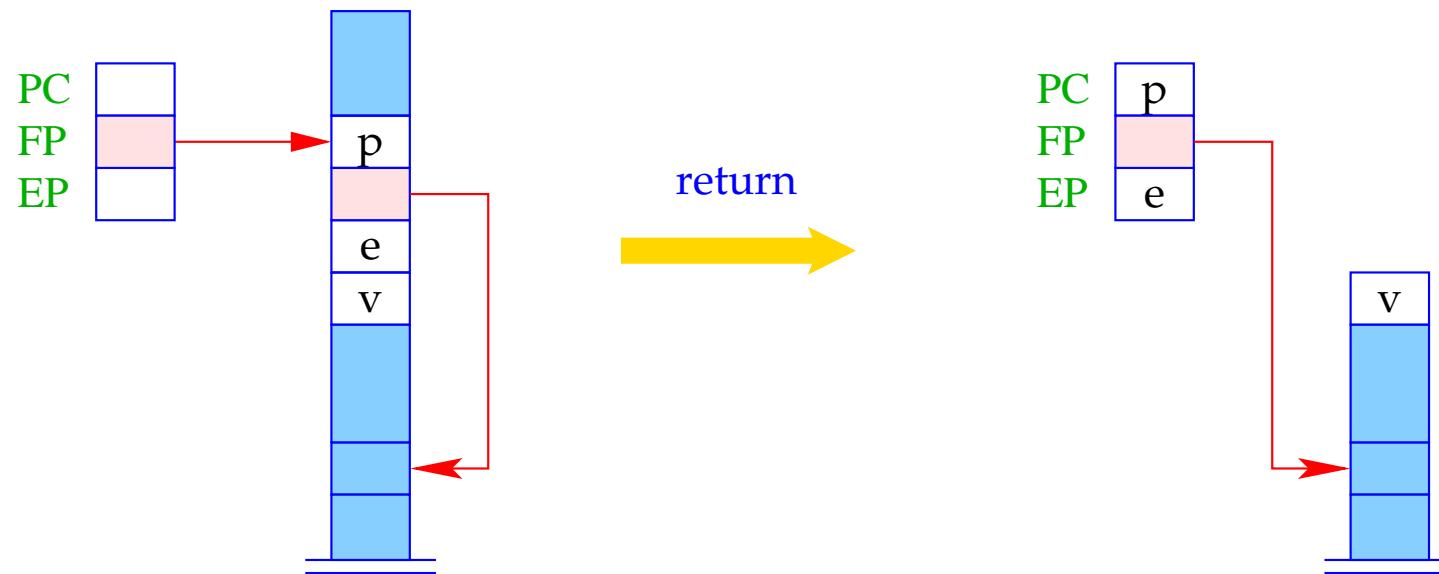


$EP = SP + q;$   
if ( $EP \geq NP$ )  
    Error ("Stack Overflow");

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



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

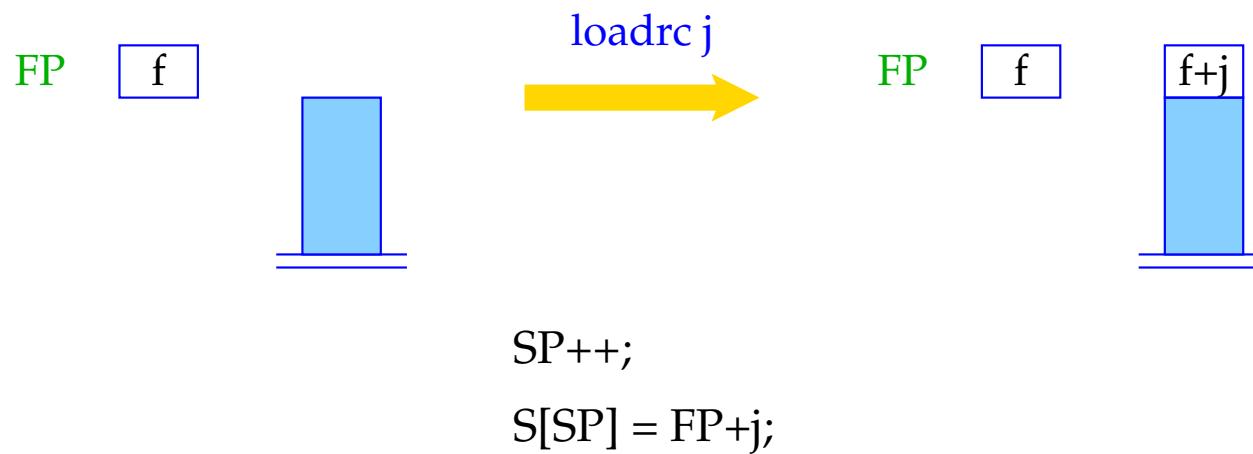
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

Darum modifizieren wir  $\text{code}_L$  für Variablen-Namen.

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

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

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



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  $-3$ .

```
code return  $e$ ;  $\rho$  = codeR  $e$   $\rho$   
                      storer -3  
                      return
```

**Beispiel:** Für die Funktion

```
int fac (int  $x$ ) {  
    if ( $x \leq 0$ ) return 1;  
    else return  $x * \text{fac} (x - 1)$ ;  
}
```

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  $q = 1 + 5 = 6$ .

## 10 Übersetzung ganzer Programme

Vor der Programmausführung gilt:

$$\text{SP} = -1 \quad \text{FP} = \text{EP} = 0 \quad \text{PC} = 0 \quad \text{NP} = \text{MAX}$$

Sei  $p \equiv V\_defs \ F\_def_1 \dots F\_def_n$ , 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**.

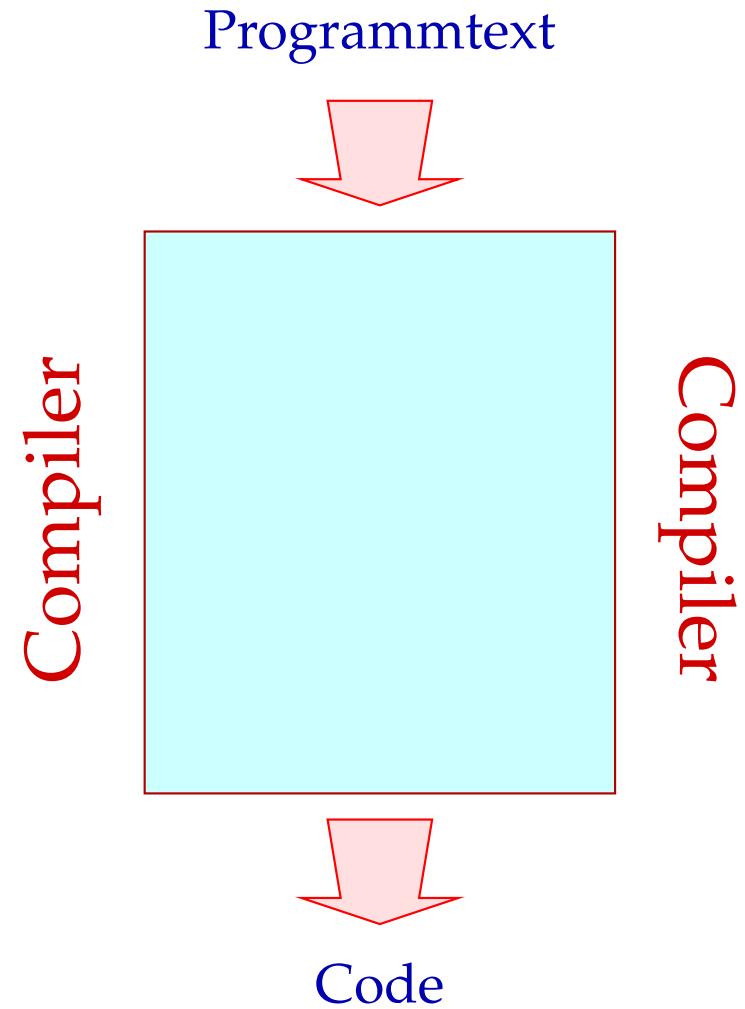
Dann definieren wir:

$$\begin{aligned}\text{code } p \emptyset &= & \text{enter } (k + 5) \\ && \text{alloc } (k + 1) \\ && \text{mark} \\ && \text{loadc } \text{main} \\ && \text{call} \\ && \text{slide } (k + 1) \\ && \text{halt} \\ \text{f}_1: & \text{ code } F_{\text{def}_1} \rho \\ && \vdots \\ \text{f}_n: & \text{ code } F_{\text{def}_n} \rho\end{aligned}$$

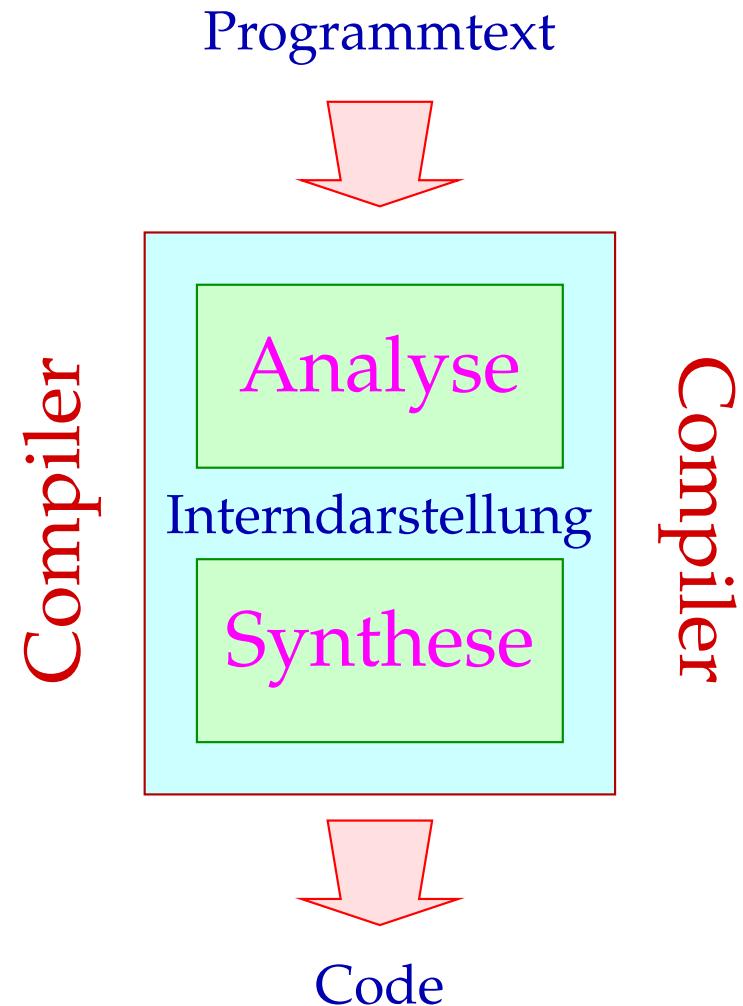
wobei  $\emptyset \hat{=} \text{leere Adress-Umgebung};$   
 $\rho \hat{=} \text{globale Adress-Umgebung};$   
 $k \hat{=} \text{Platz für globale Variablen}$

# Die Analyse-Phase

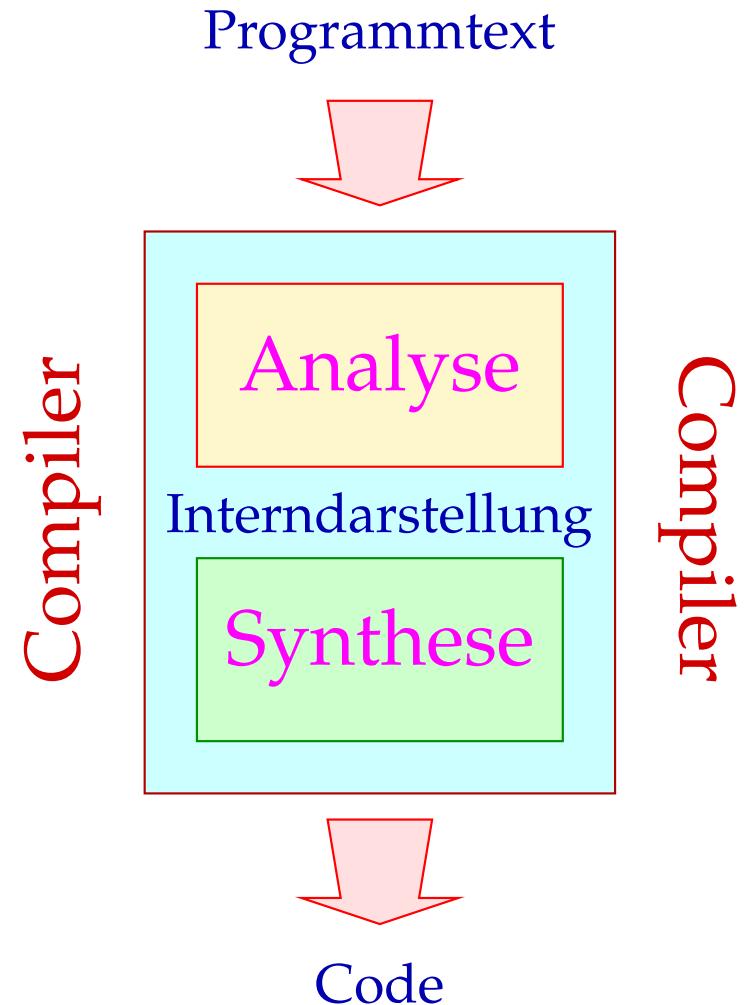
# Orientierung:



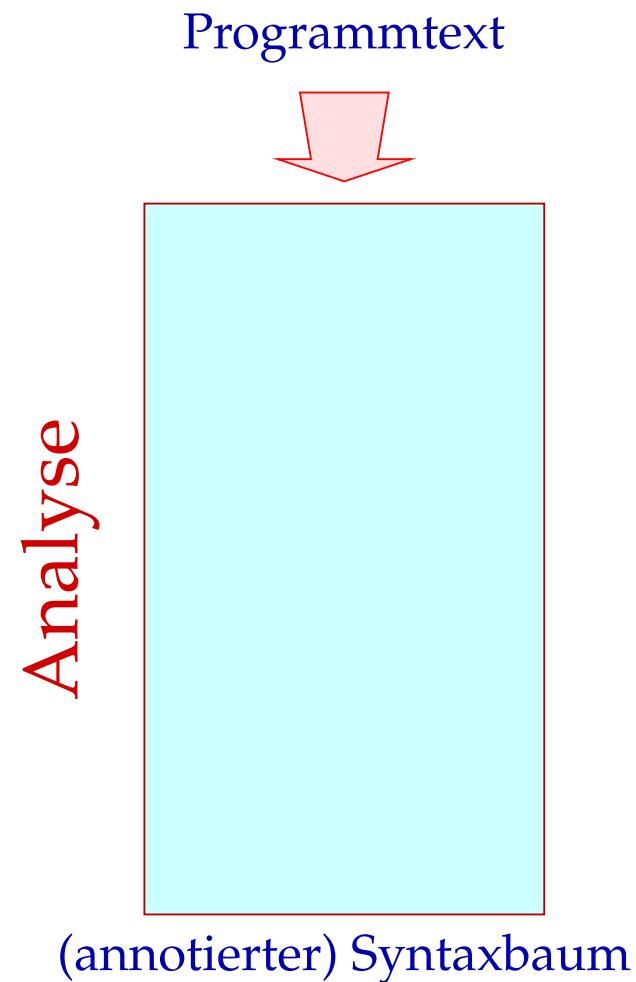
# Orientierung:



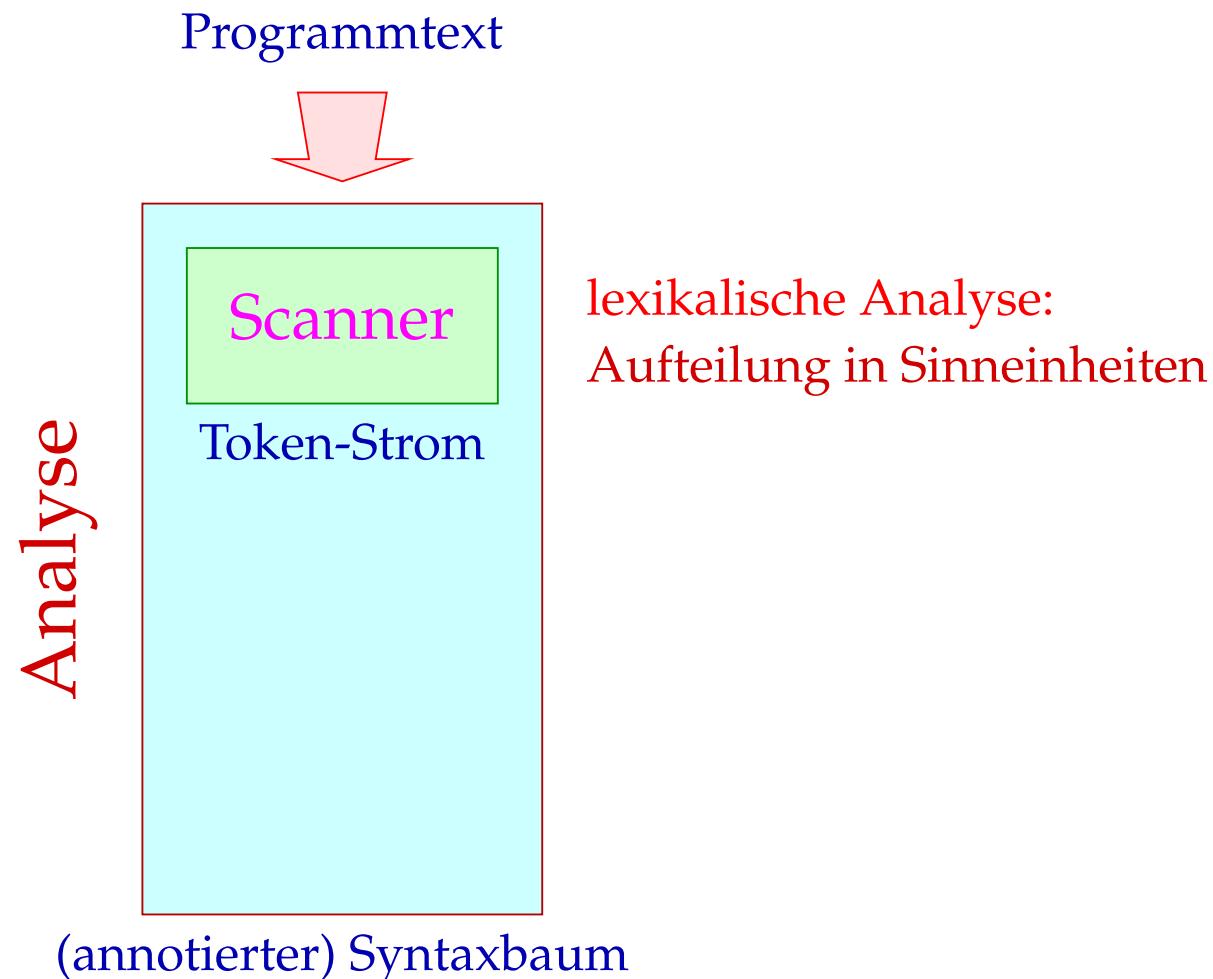
# Orientierung:



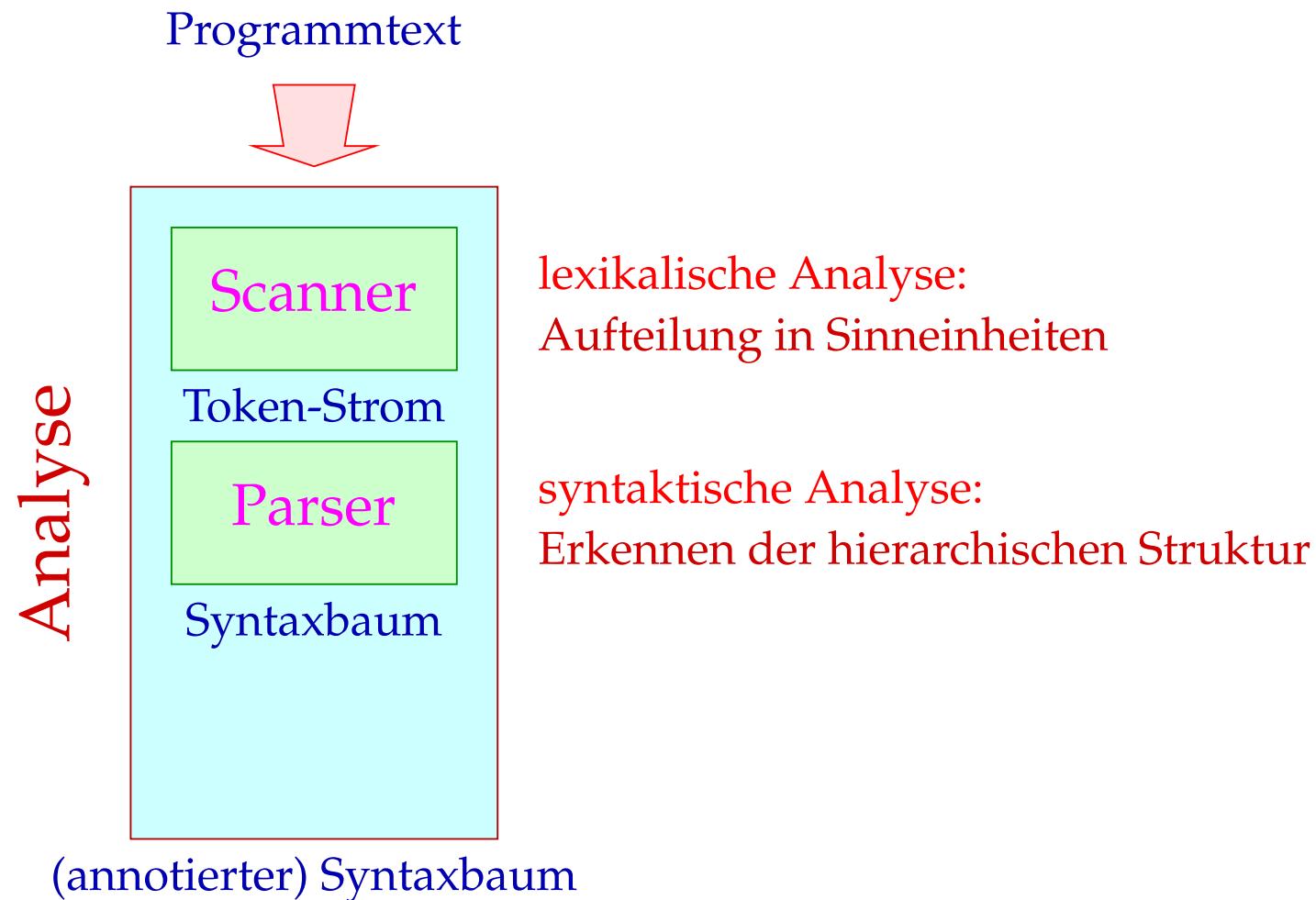
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse**-Phase :-)



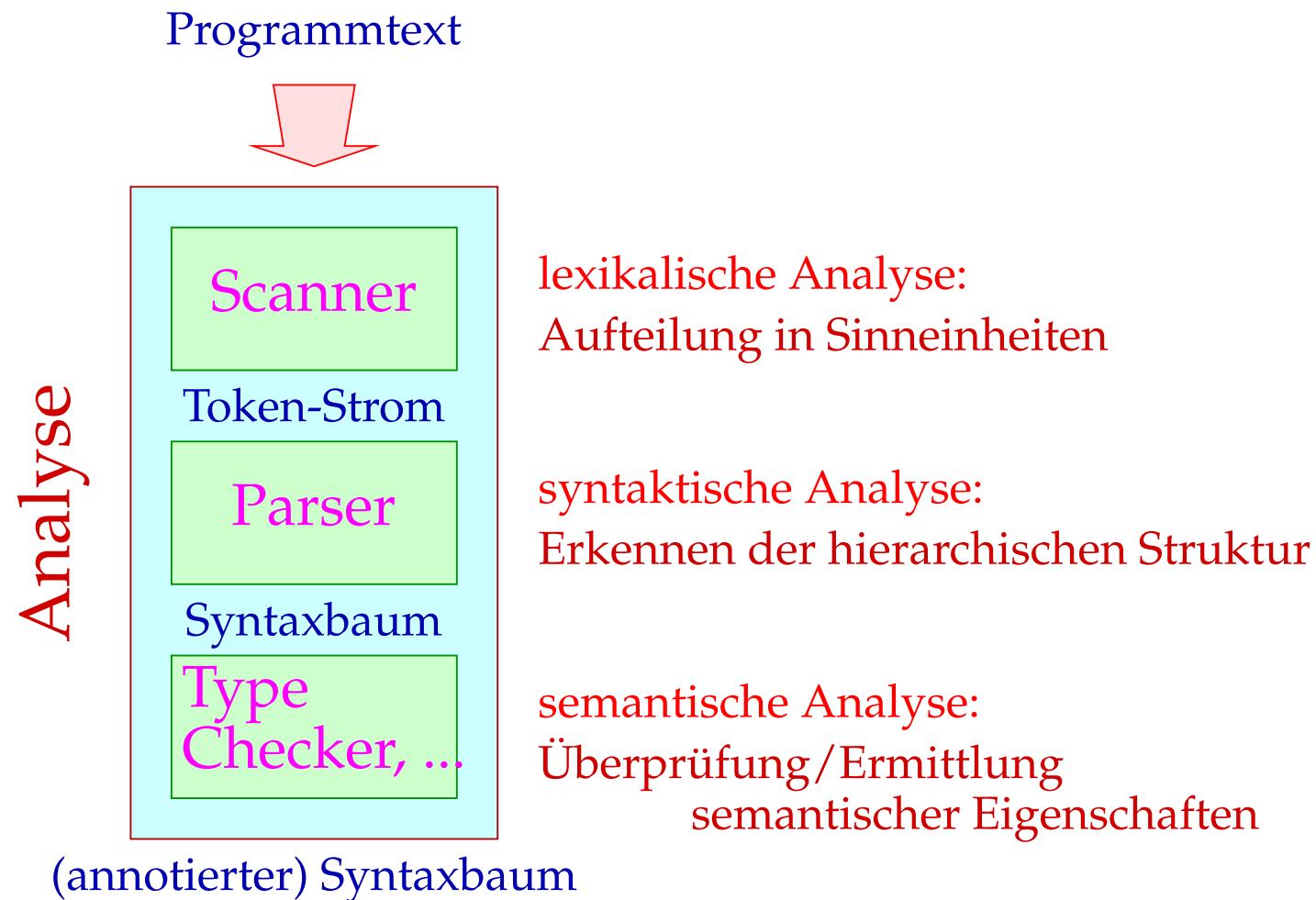
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse**-Phase :-)



Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse**-Phase :-)



Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse**-Phase :-)



# 1 Die lexikalische Analyse



# 1 Die lexikalische Analyse



# 1 Die lexikalische Analyse



- Ein **Token** ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in **Klassen** zusammen gefasst. Zum Beispiel:
  - Namen (**Identifier**) wie xyz, pi, ...
  - Konstanten wie 42, 3.14, "abc", ...
  - Operatoren wie +, ...
  - reservierte Worte wie if, int, ...

# 1 Die lexikalische Analyse



- Ein **Token** ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in **Klassen** zusammen gefasst. Zum Beispiel:
  - Namen (**Identifier**) wie xyz, pi, ...
  - Konstanten wie 42, 3.14, "abc", ...
  - Operatoren wie +, ...
  - reservierte Worte wie if, int, ...

Sind Tokens erst einmal klassifiziert, kann man die Teilwörter **vorverarbeiten**:

- **Wegwerfen** irrelevanter Teile wie **Leerzeichen, Kommentaren,...**
- **Aussondern** von **Pragmas**, d.h. Direktiven an den Compiler, die nicht Teil des Programms sind, wie **include**-Anweisungen;
- **Ersetzen** der Token bestimmter Klassen durch ihre Bedeutung / Interndarstellung, etwa bei:
  - **Konstanten**;
  - **Namen**: die typischerweise zentral in einer **Symbol**-Tabelle verwaltet, evt. mit reservierten Worten verglichen (soweit nicht vom Scanner bereits vorgenommen **:-)** und gegebenenfalls durch einen Index ersetzt werden.

==> **Sieber**

## Diskussion:

- Scanner und Sieber werden i.a. in einer Komponente zusammen gefasst, indem man dem Scanner nach Erkennen eines Tokens gestattet, eine Aktion auszuführen :-)
- Scanner werden i.a. nicht von Hand programmiert, sondern aus einer Spezifikation **generiert**:



# Vorteile:

## Produktivität:

Die Komponente lässt sich **schneller** herstellen :-)

## Korrektheit:

Die Komponente realisiert (beweisbar :-)) die Spezifikation.

## Effizienz:

Der Generator kann die erzeugte Programmkomponente mit den effizientesten Algorithmen ausstatten.

# Vorteile:

## Produktivität:

Die Komponente lässt sich **schneller** herstellen :-)

## Korrektheit:

Die Komponente realisiert (beweisbar :-)) die Spezifikation.

## Effizienz:

Der Generator kann die erzeugte Programmkomponente mit den effizientesten Algorithmen ausstatten.

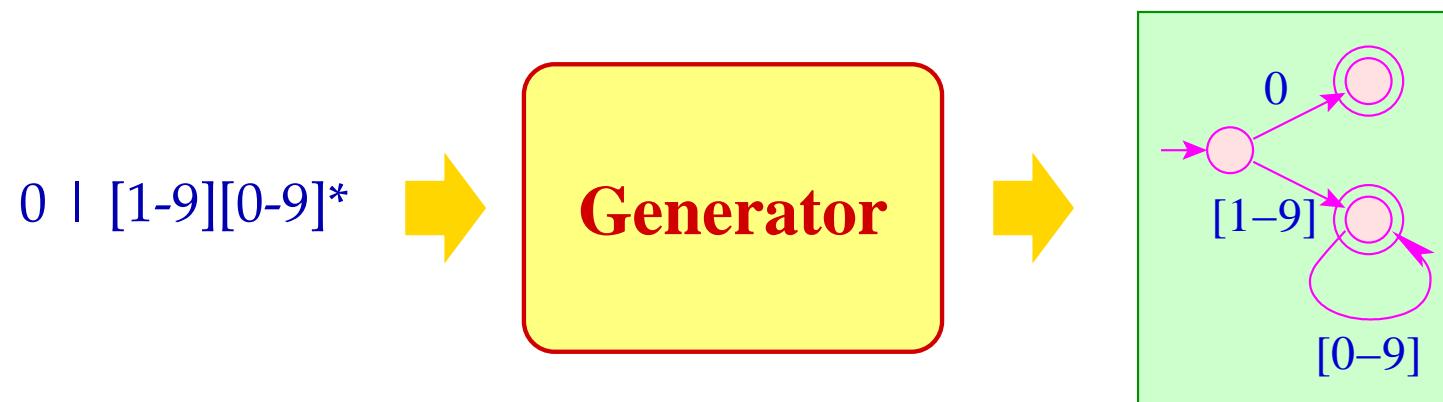
# Einschränkungen:

- Spezifizieren ist auch **Programmieren** — nur eventuell einfacher :-)
- Generierung statt Implementierung lohnt sich nur für **Routine-Aufgaben**  
... und ist nur für Probleme möglich, die **sehr gut verstanden** sind :-()

... in unserem Fall:



... in unserem Fall:



**Spezifikation von Token-Klassen:** Reguläre Ausdrücke;

**Generierte Implementierung:** Endliche Automaten +  $\text{X}$  :-)