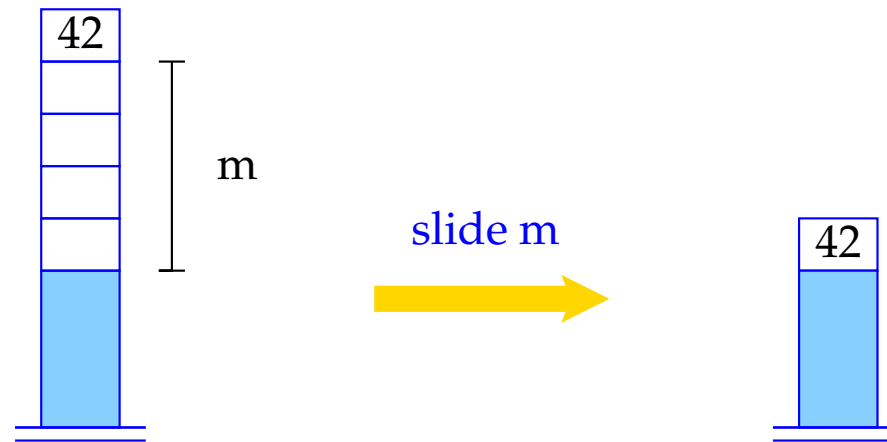


Der Befehl `slide` kopiert den Rückgabewert an die korrekte Stelle:



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

```
SP = SP-m;
```

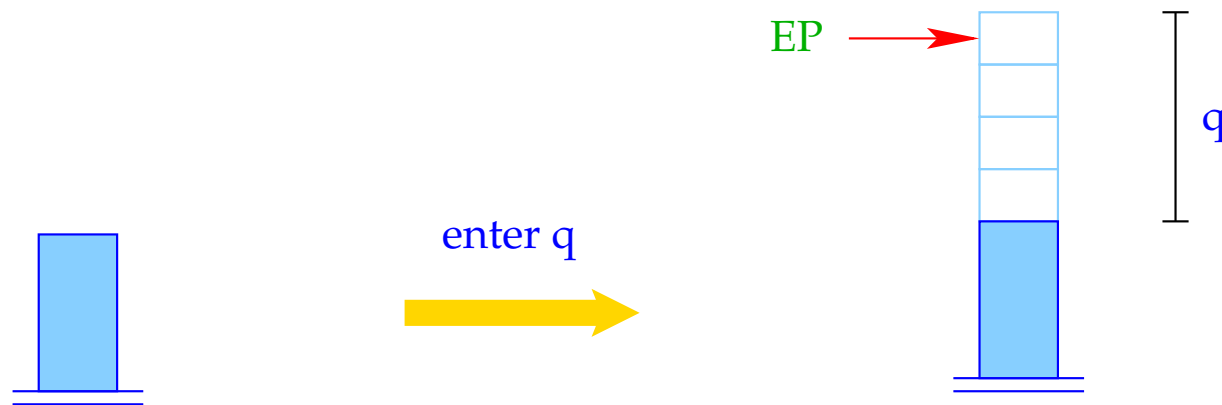
```
S[SP] = tmp;
```

Entsprechend übersetzen wir eine Funktions-Definition:

```
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
 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.

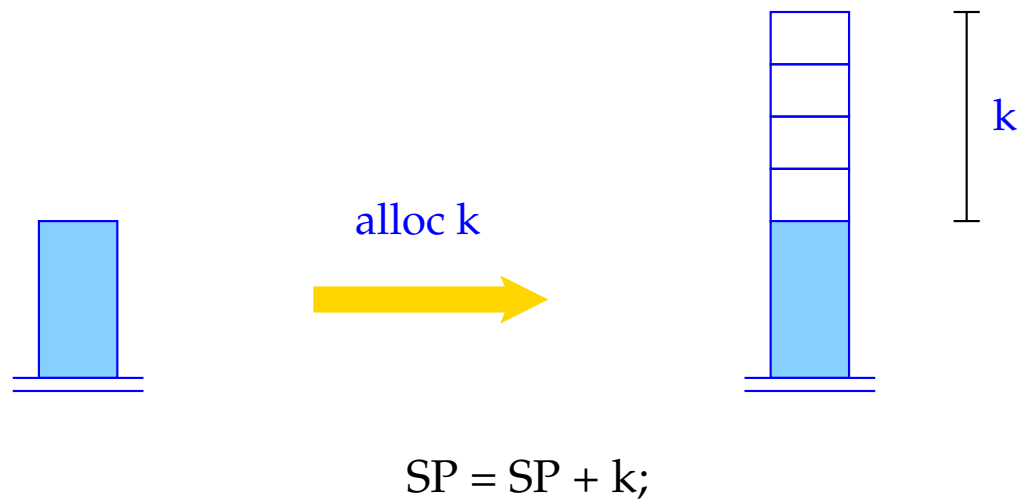


```
EP = SP + q;
```

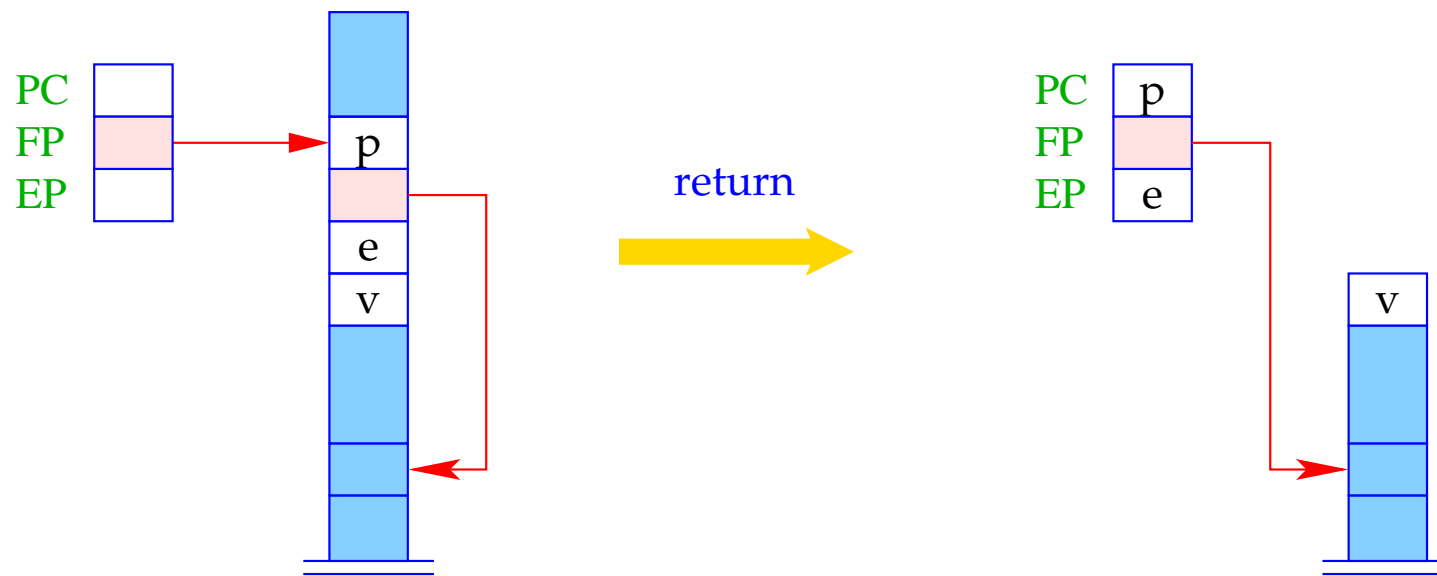
```
if (EP ≥ NP)
```

```
    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`, `FP` und `EP` 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

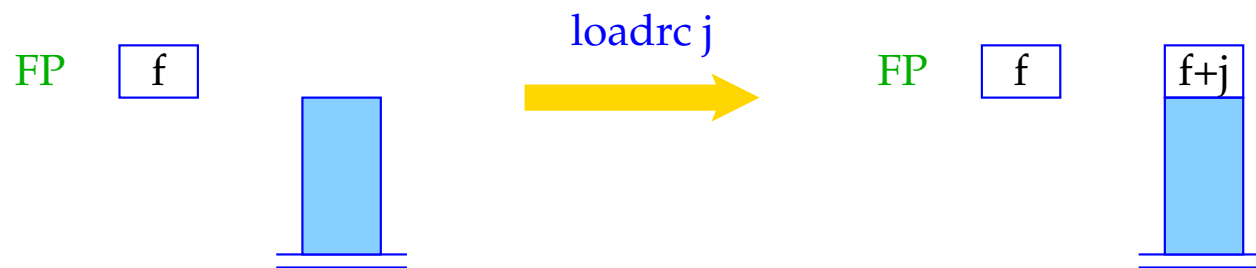
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

Darum modifizieren wir `codeL` 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`.



```
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`.

$$\text{code } \text{return } e; \rho = \text{code}_R e \rho$$

`storer -3`
`return`

Beispiel: Für die Funktion

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

erzeugen wir:

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

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

10 Übersetzung ganzer Programme

Vor der Programmausführung gilt:

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

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`.

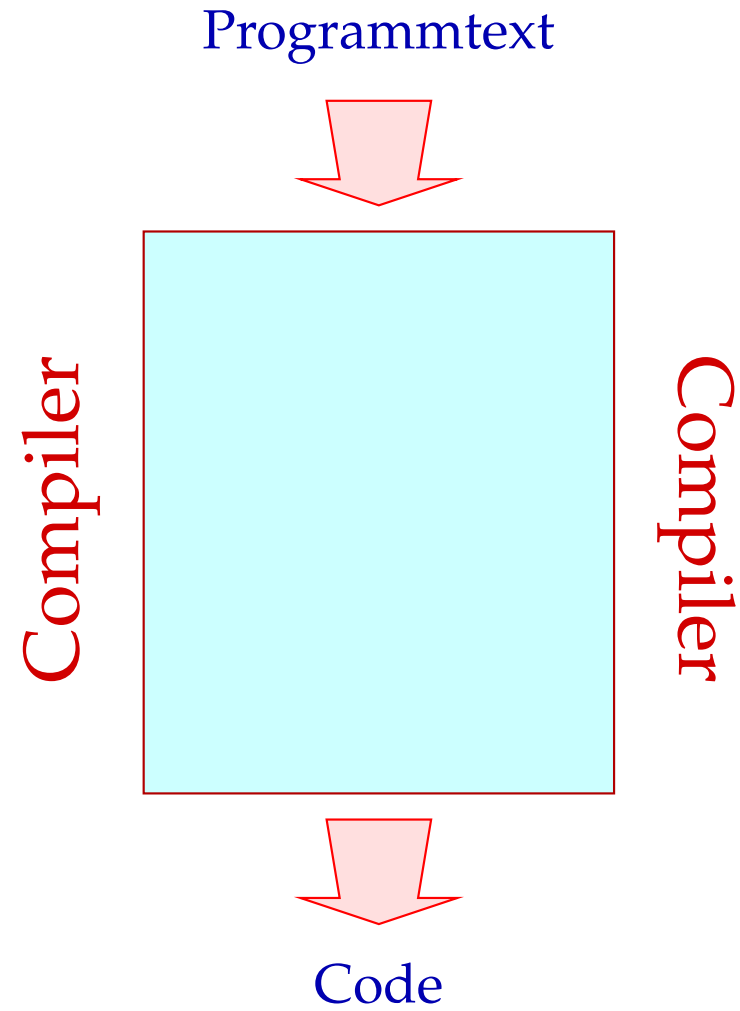
Dann definieren wir:

```
code  $p \ \emptyset$    =   enter (k + 4)
                   alloc (k + 1)
                   mark
                   loadc _main
                   call
                   slide k
                   halt
                   _f1: code  $F_{def_1} \ \rho$ 
                   ⋮
                   _fn: code  $F_{def_n} \ \rho$ 
```

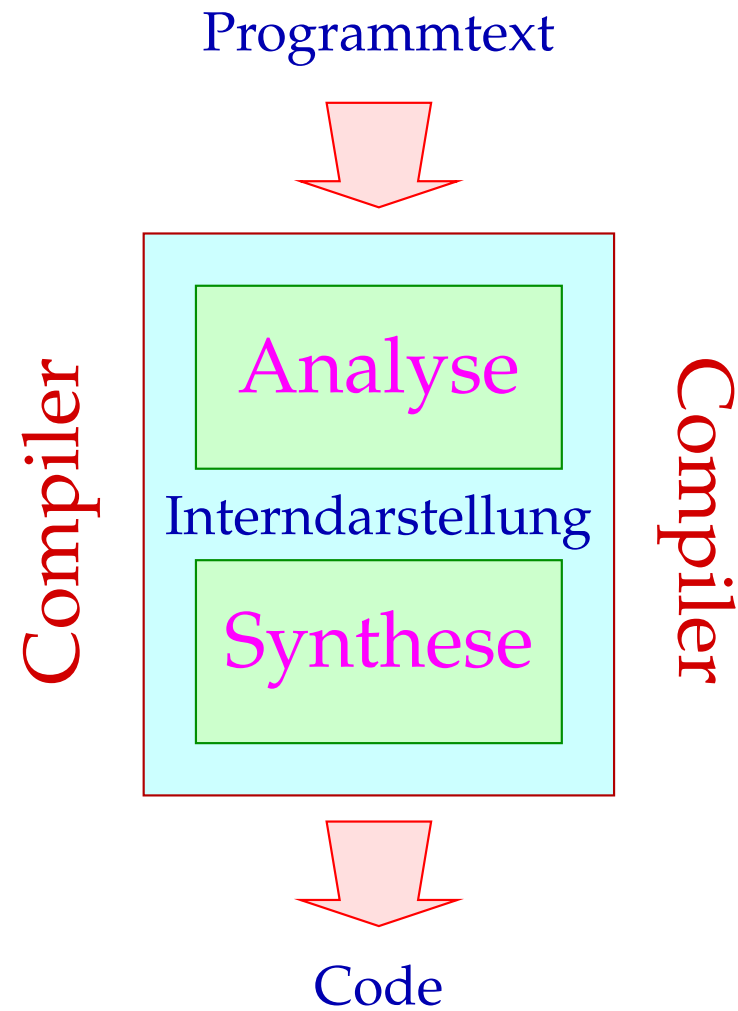
wobei $\emptyset \hat{=}$ leere Adress-Umgebung;
 $\rho \hat{=}$ globale Adress-Umgebung;
 $k \hat{=}$ Platz für globale Variablen

Die Analyse-Phase

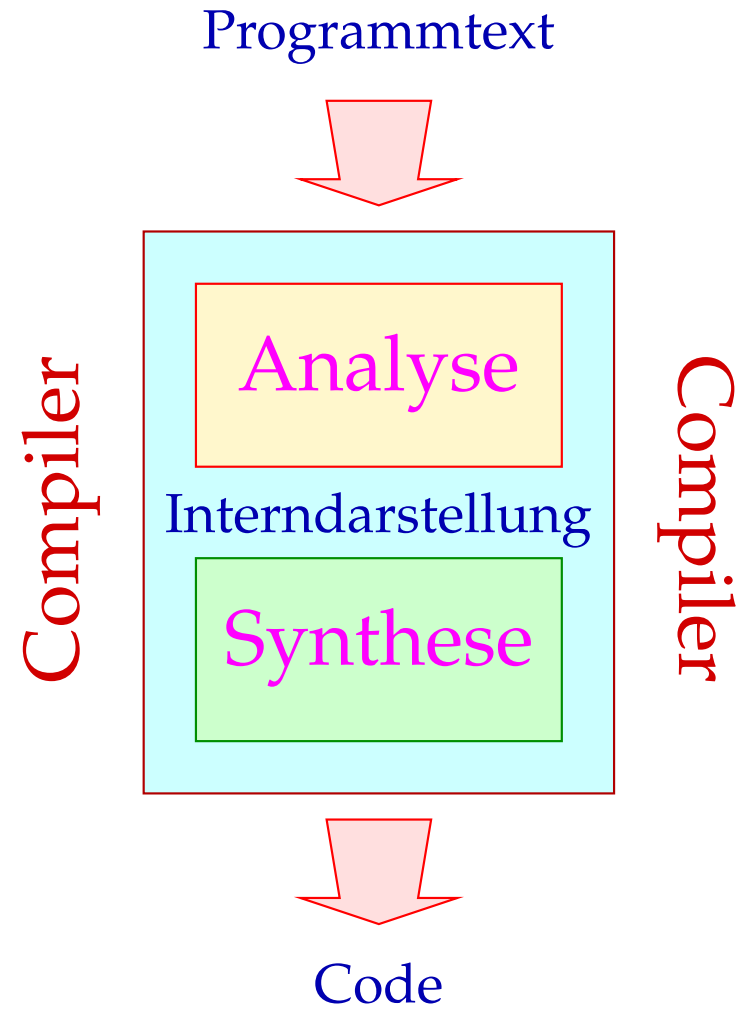
Orientierung:



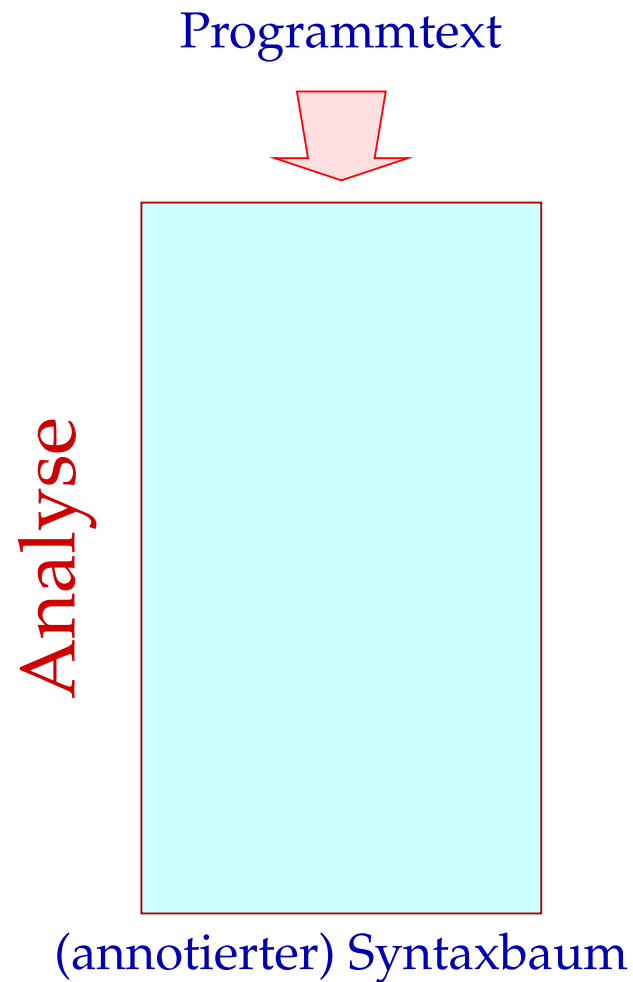
Orientierung:



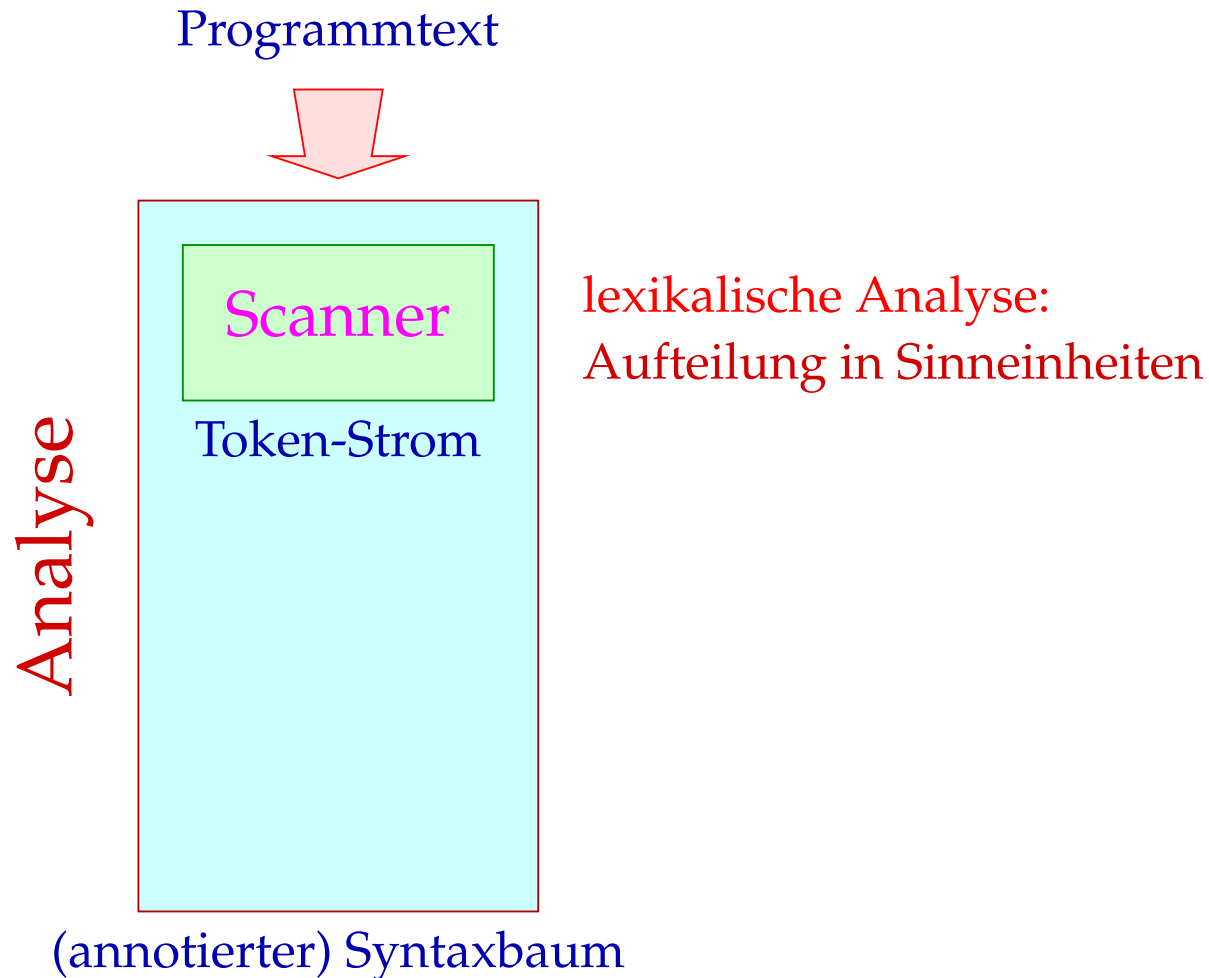
Orientierung:



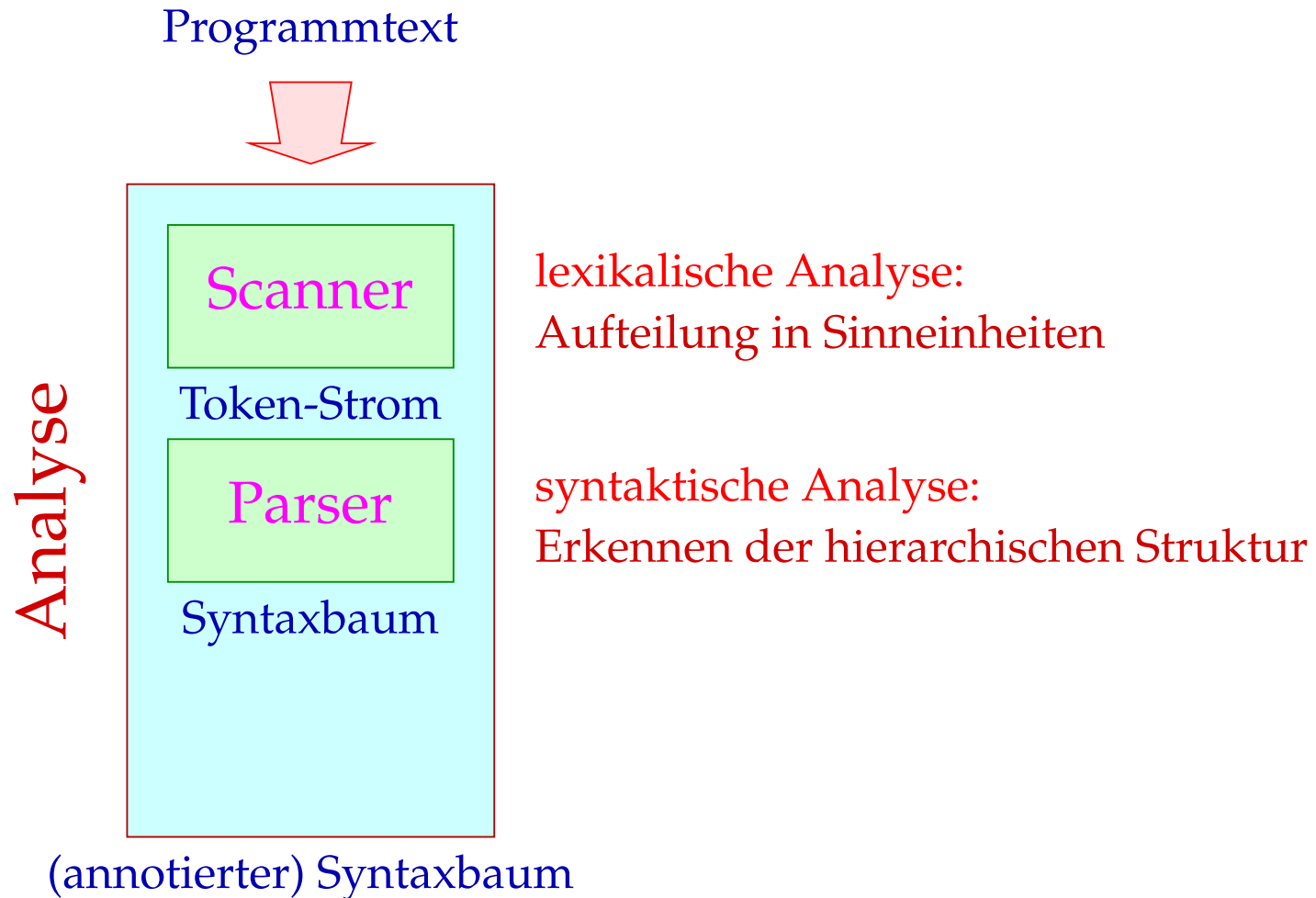
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse-Phase** :-)



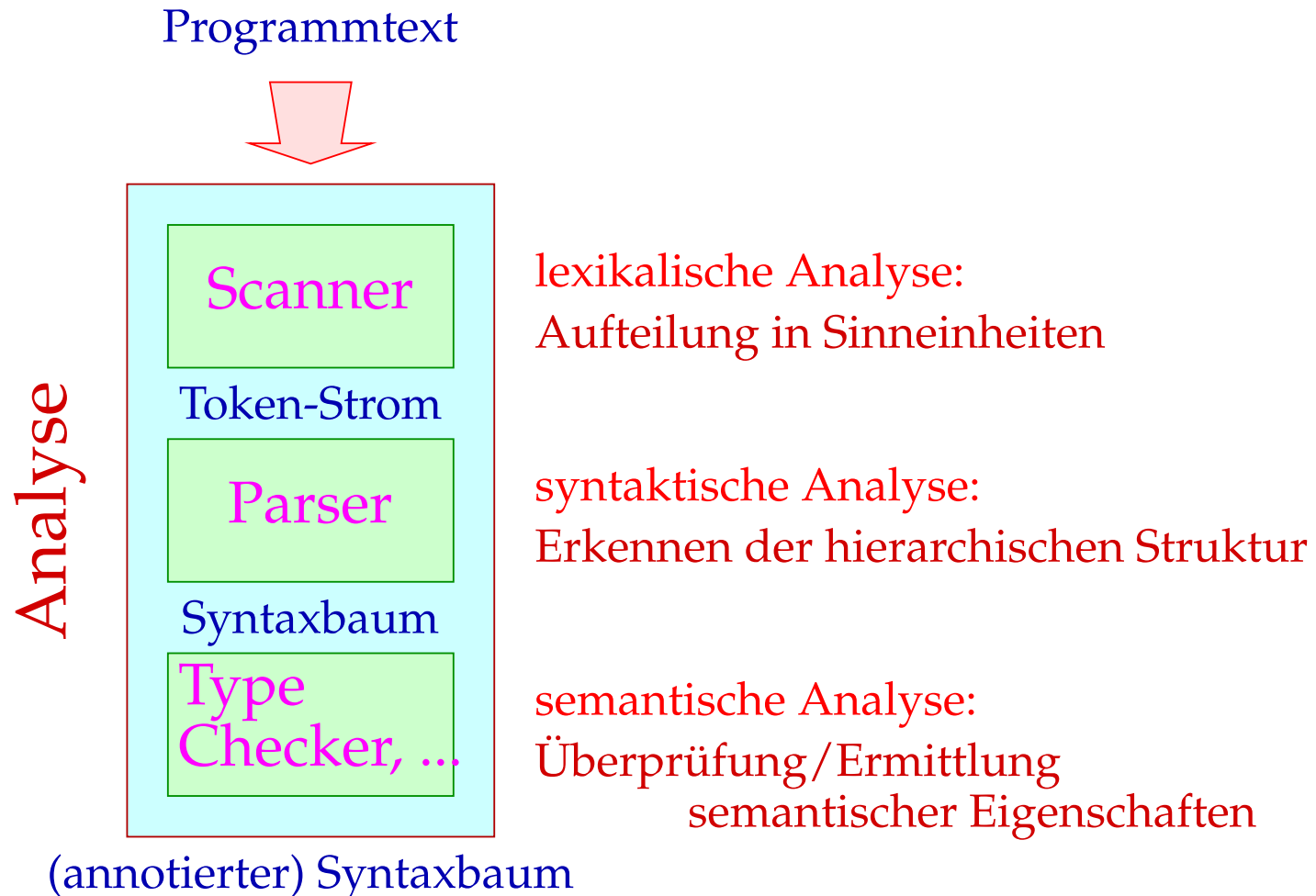
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse-Phase** :-)



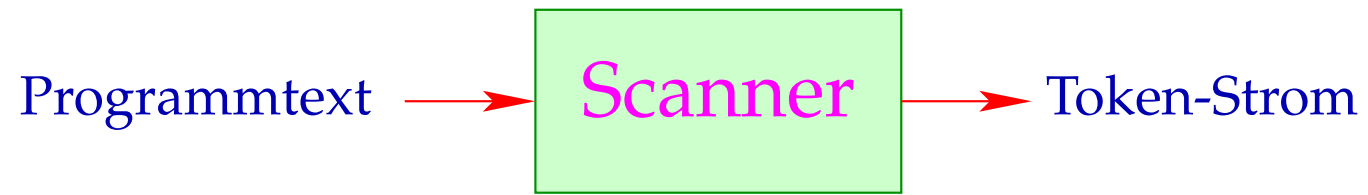
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse-Phase** :-)



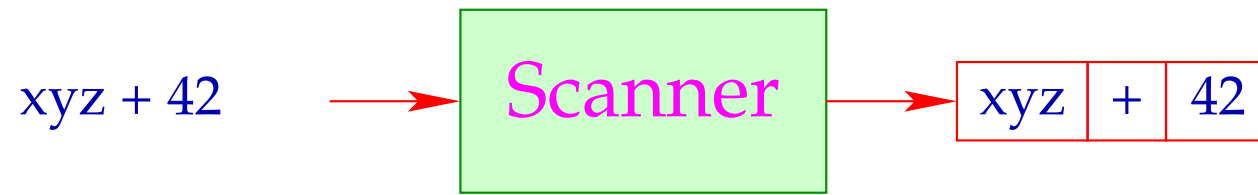
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die **Analyse-Phase** :-)



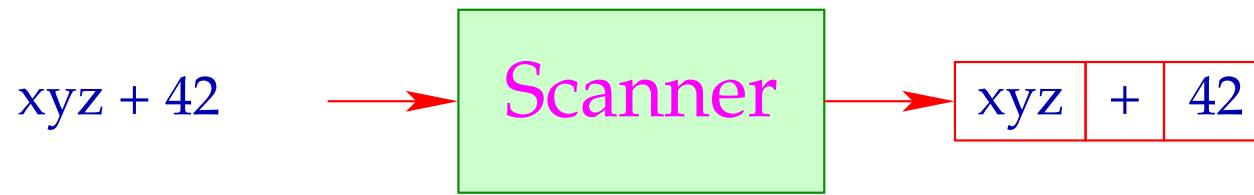
1 Die lexikalische Analyse



1 Die lexikalische Analyse

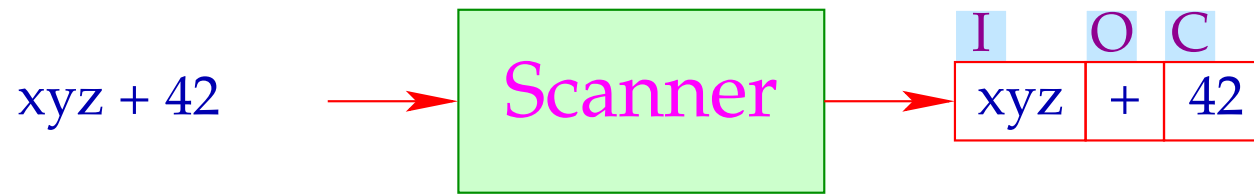


1 Die lexikalische Analyse



- 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 :-)