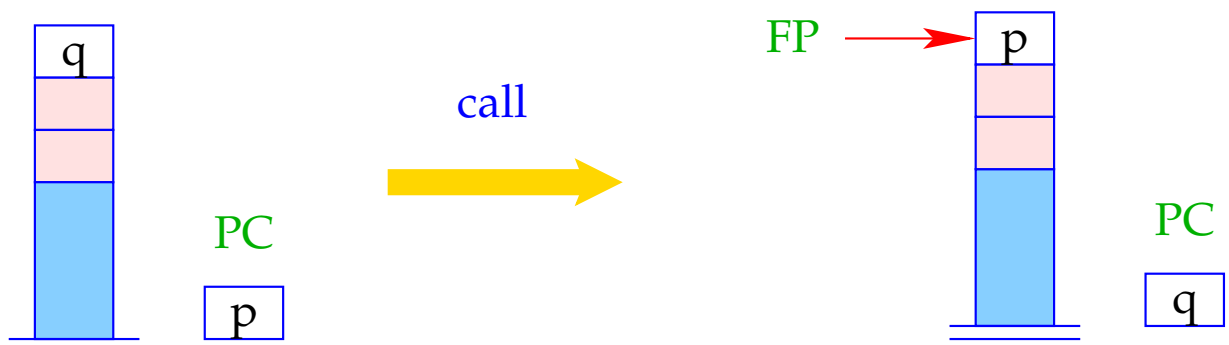
Der Befehl mark legt Platz für Rückgabewert und organisatorische Zellen an und rettet FP und EP.

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.



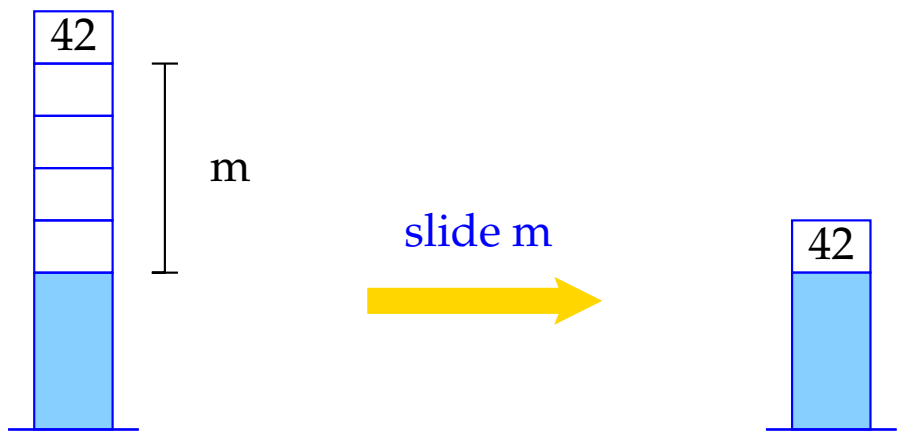
tmp = S[SP];

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 $\mathrm{q} = max + \mathrm{k}$ wobei

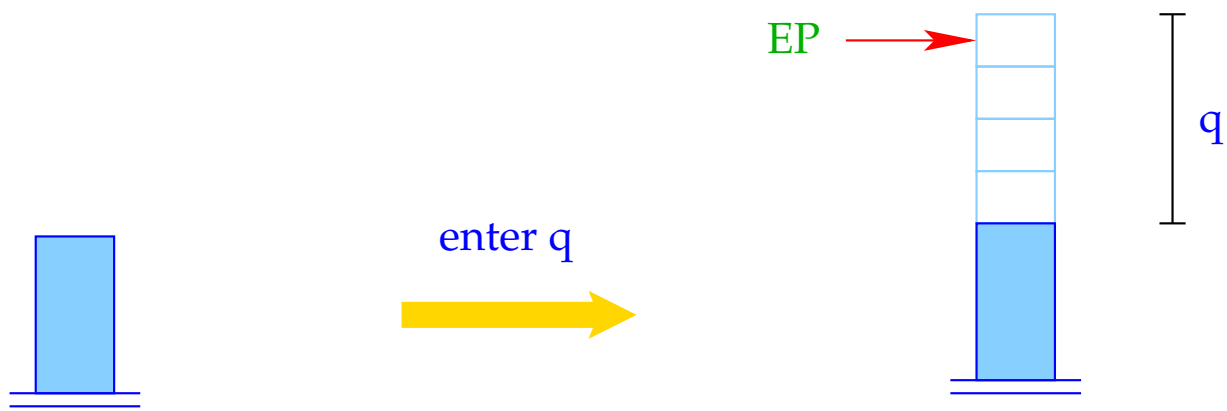
max = maximale Länge des lokalen Kellers

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.

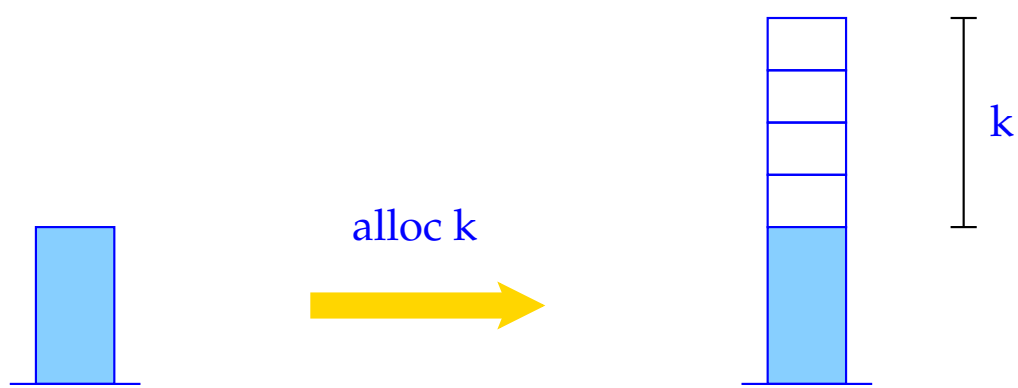


EP = SP + q;

if (EP $\geq$ NP)

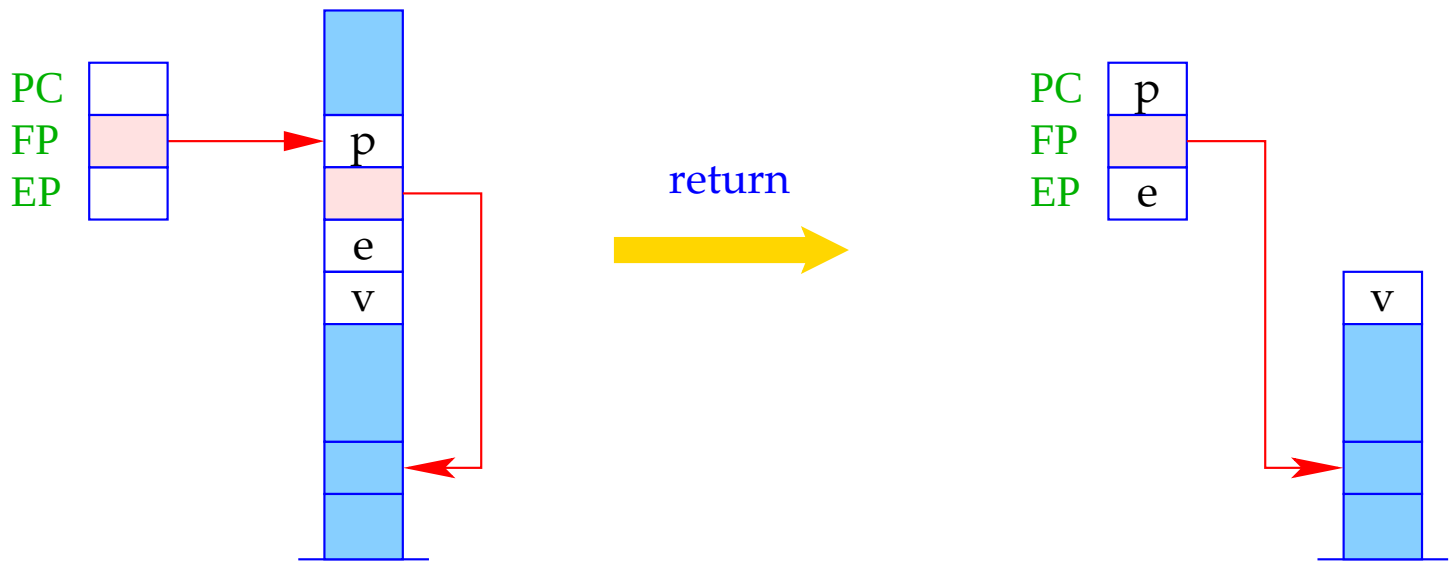
Error ("Stack Overflow");

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:



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

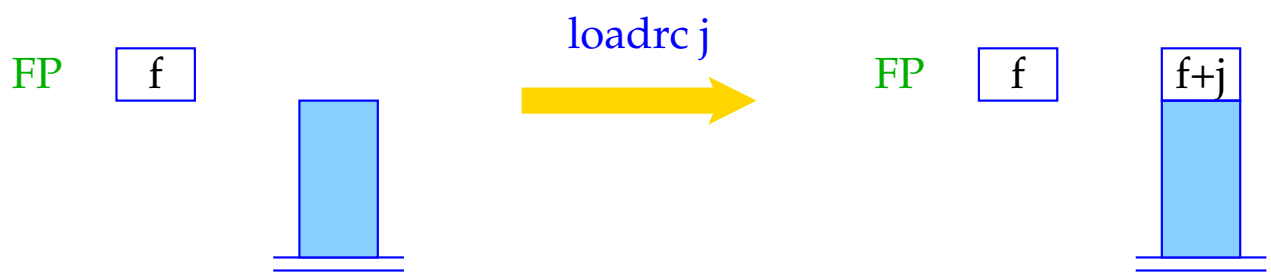
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

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
bla; storer j	=	loadrc j; bla store

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

$$\text{code}\ \textbf{return}\, e;\ \rho \quad = \quad \begin{array}{l} \text{code}_{\text{R}}\ e\ \rho \\ \text{storer -3} \\ \text{return} \end{array}$$

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
```

Dabei ist $\rho_{\text{fac}} : x \mapsto (L, -3)$ und $\mathrm{q} = 1 + 5 = 6$.

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:

$$
\begin{array}{lcll}
\text{code } p\ \emptyset & = & & \text{enter } (\mathrm{k}+6) \\
 & & & \text{alloc } (\mathrm{k}+1) \\
 & & & \text{loadc } 0 \\
 & & & \text{mark} \\
 & & & \text{loadc _main} \\
 & & & \text{call} \\
 & & & \text{slide } (\mathrm{k}+1) \\
 & & & \text{halt} \\
 & & _\mathrm{f}_1: & \text{code } F_def_1\ \rho \\
 & & & \vdots \\
 & & _\mathrm{f}_n: & \text{code } F_def_n\ \rho
\end{array}
$$

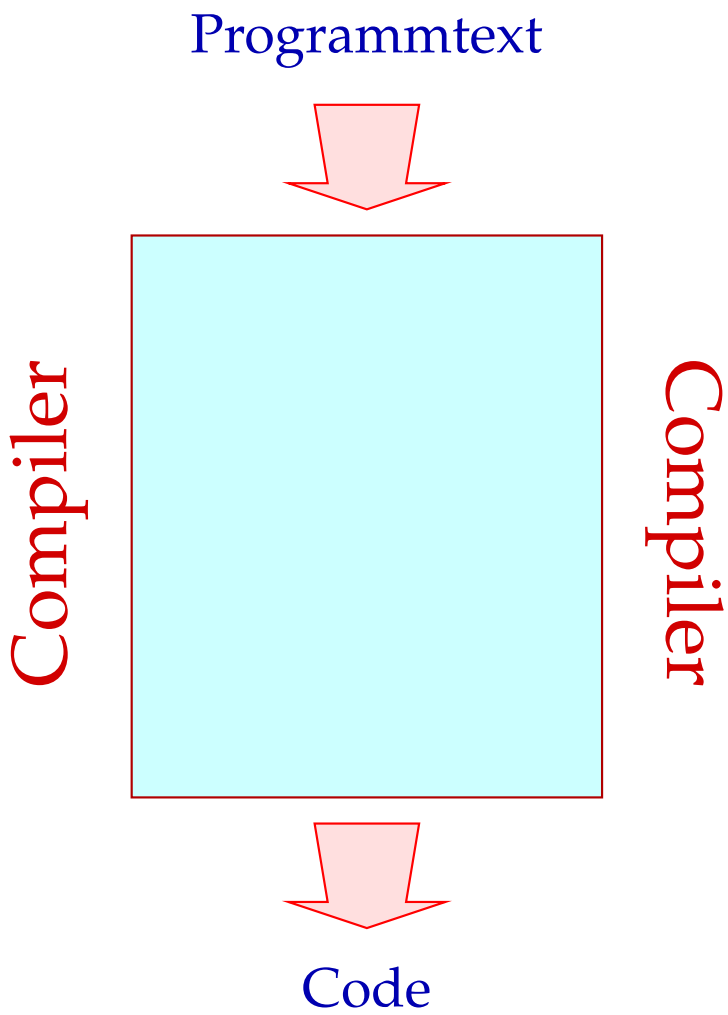
wobei $\emptyset \quad \widehat{=} \quad$ leere Adress-Umgebung;

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

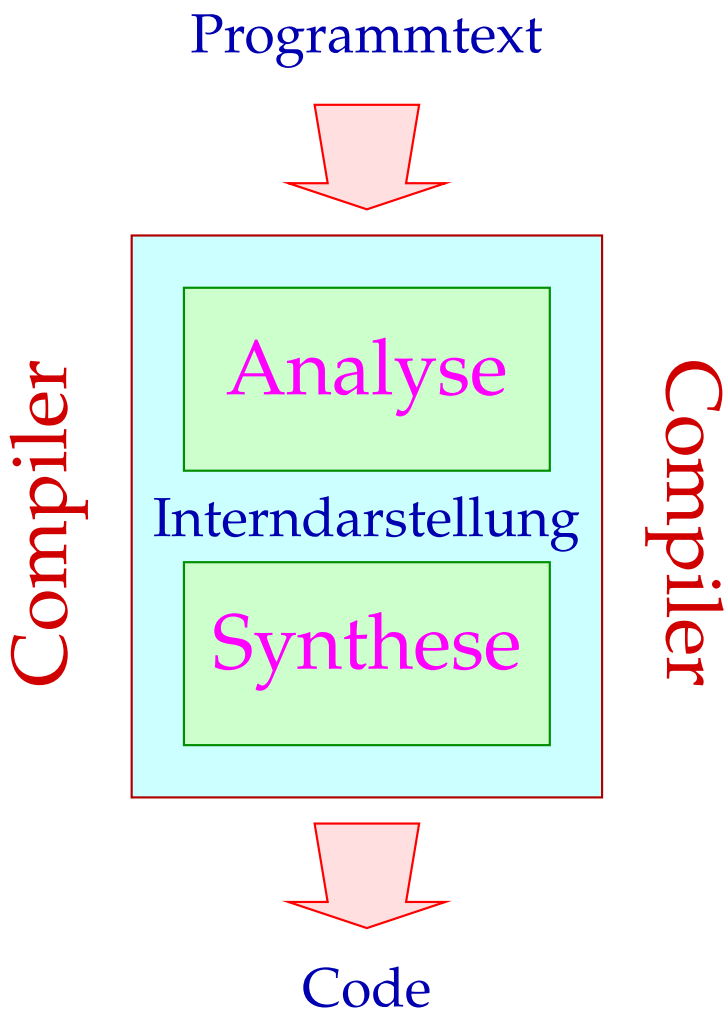
$\mathrm{k} \quad \widehat{=} \quad$ Platz für globale Variablen

Die Analyse-Phase

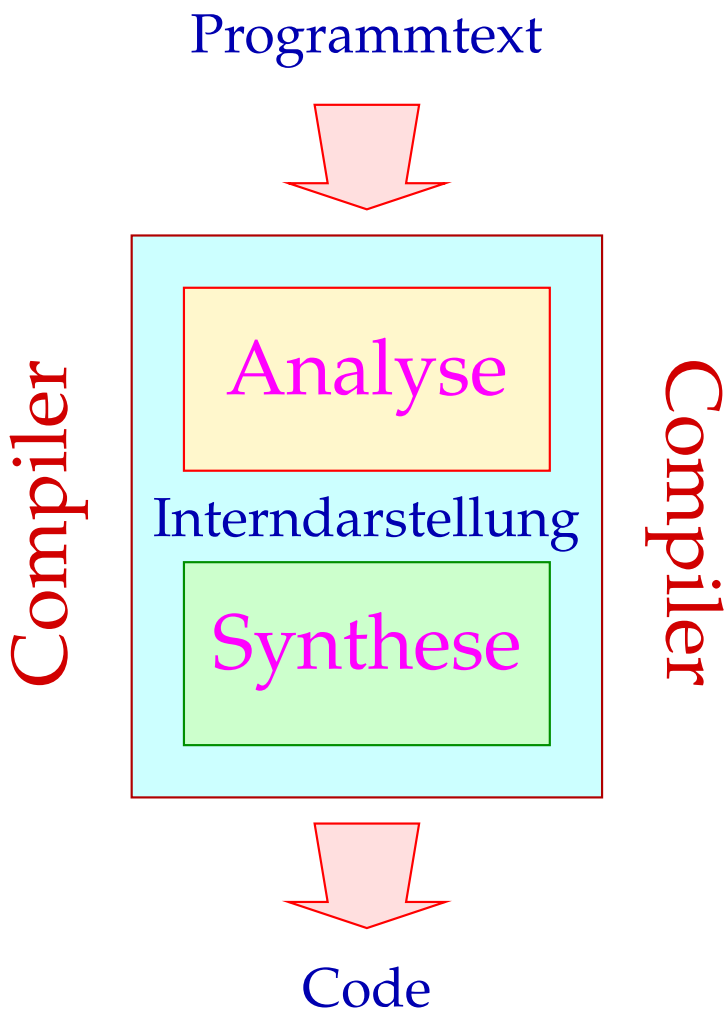
Orientierung:



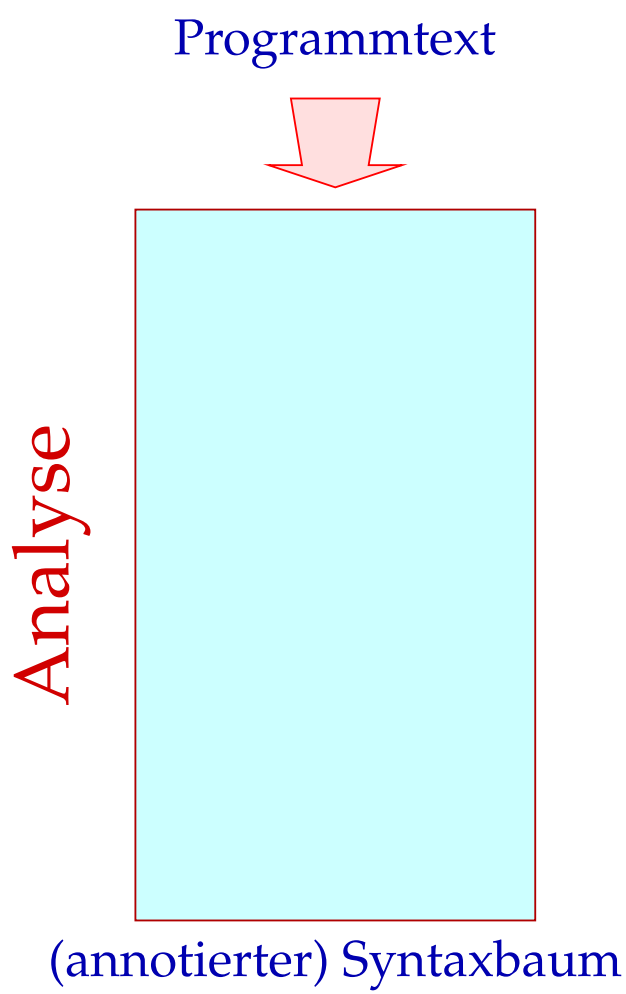
Orientierung:



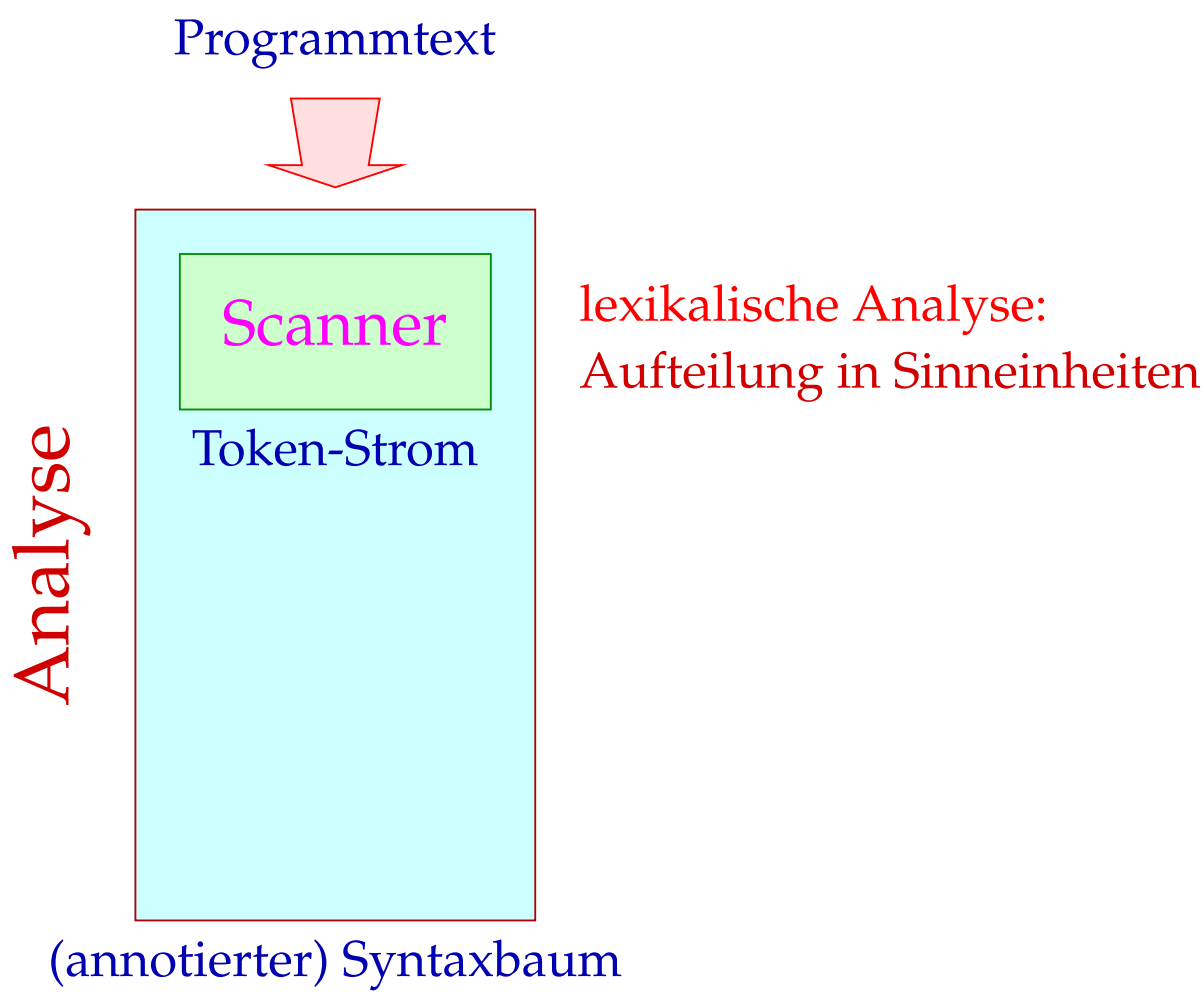
Orientierung:



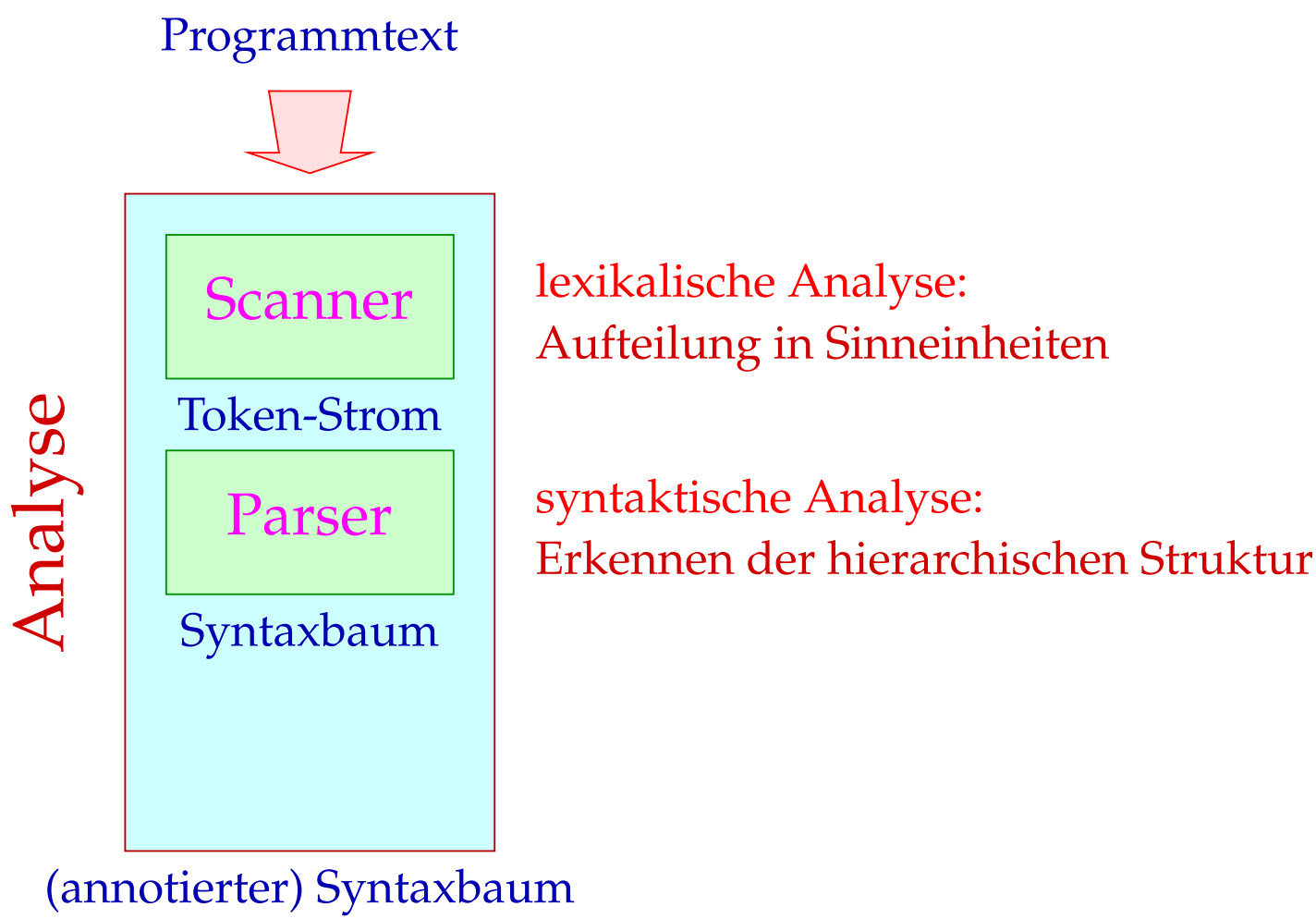
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



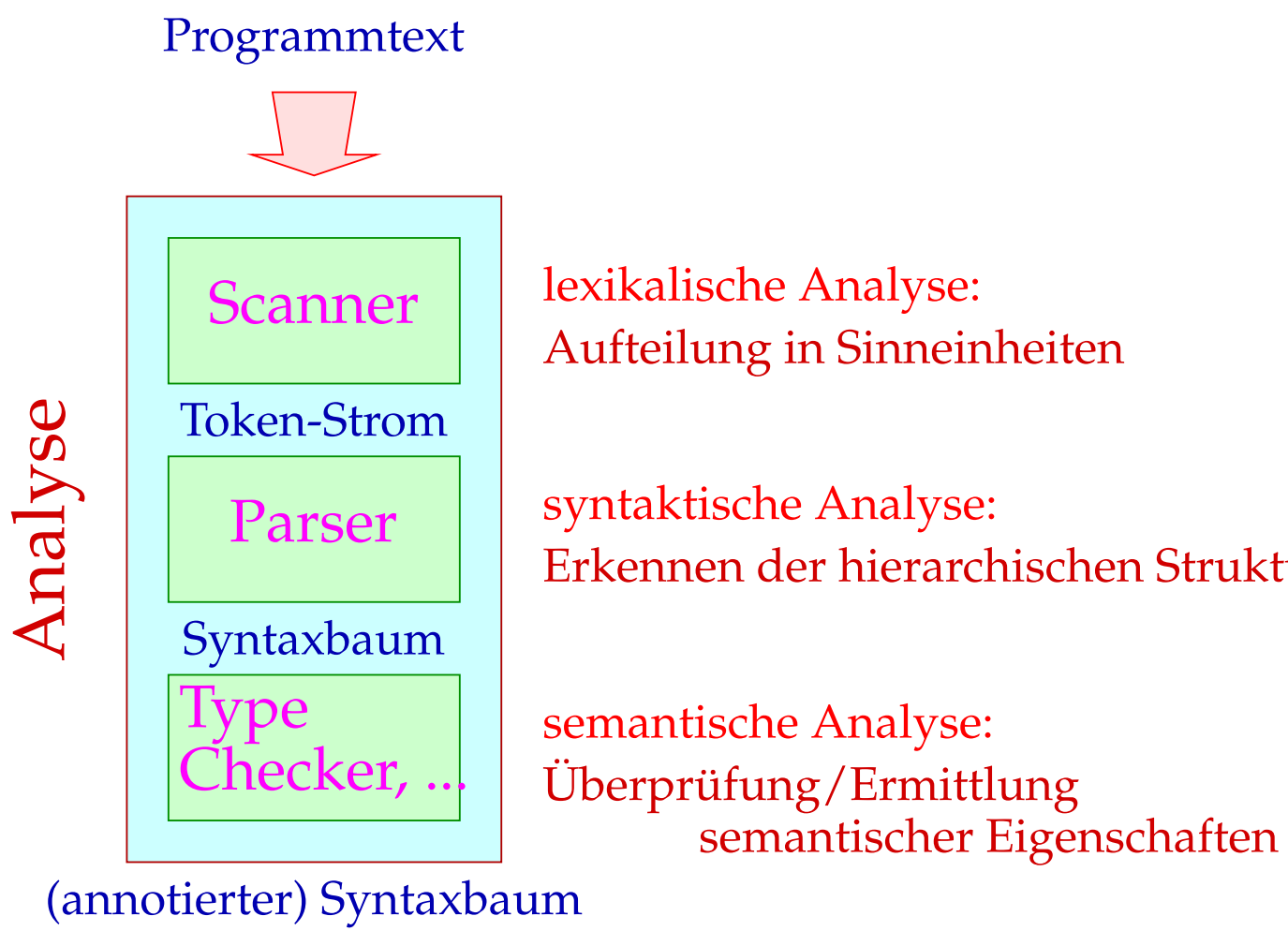
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



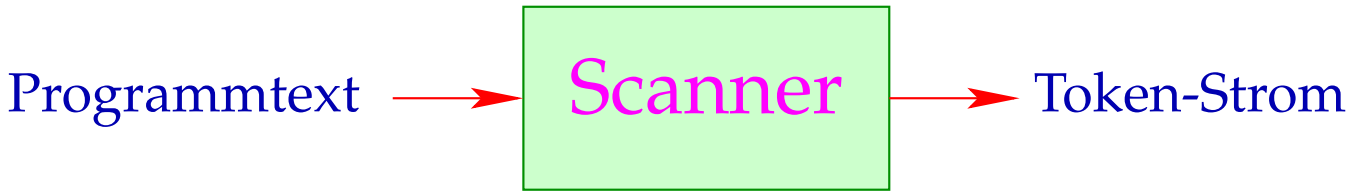
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



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

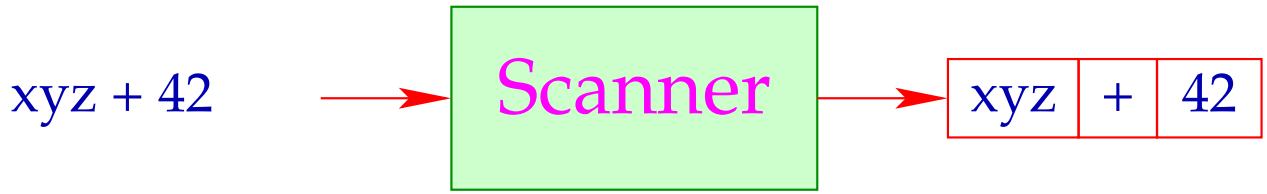


1 Die lexikalische Analyse



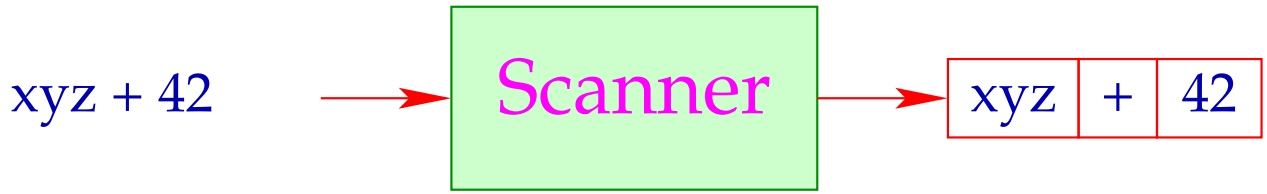
Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.

1 Die lexikalische Analyse



Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.

1 Die lexikalische Analyse



- Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in Klassen zusammen gefasst. Zum Beispiel:

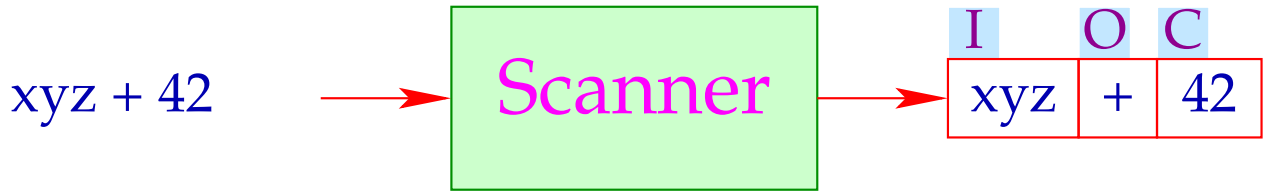
  → Namen (Identifier) wie `xyz`, `pi`, ...

  → 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.

$\Longrightarrow$ 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.

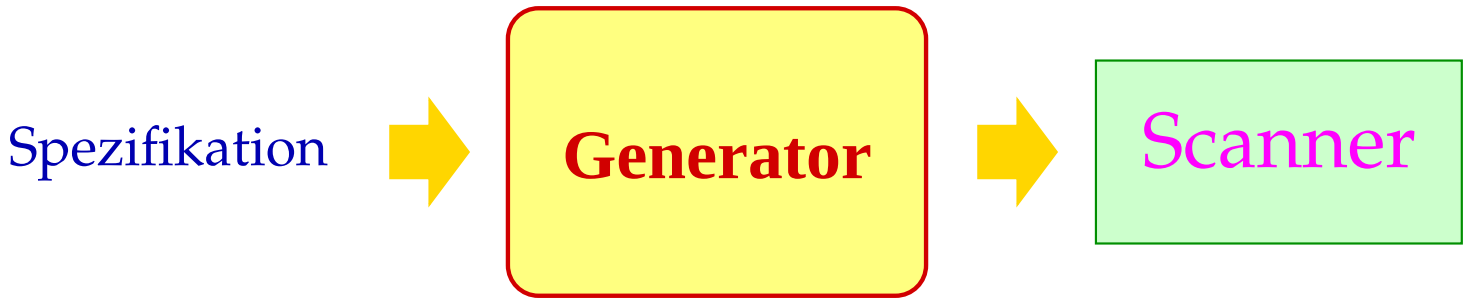
Einschränkungen:

→ Spezifizieren ist auch Programmieren — nur eventuell einfacher :-)

→ Generierung statt Implementierung lohnt sich nur für Routine-Aufgaben

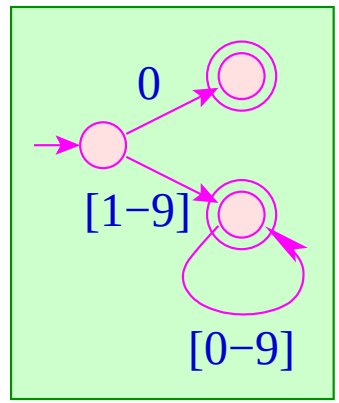
... und ist nur für Probleme möglich, die sehr gut verstanden sind :-(

... in unserem Fall:



... in unserem Fall:

0 | [1-9][0-9]*



Spezifikation von Token-Klassen: Reguläre Ausdrücke;

Generierte Implementierung: Endliche Automaten + X :-)