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:

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

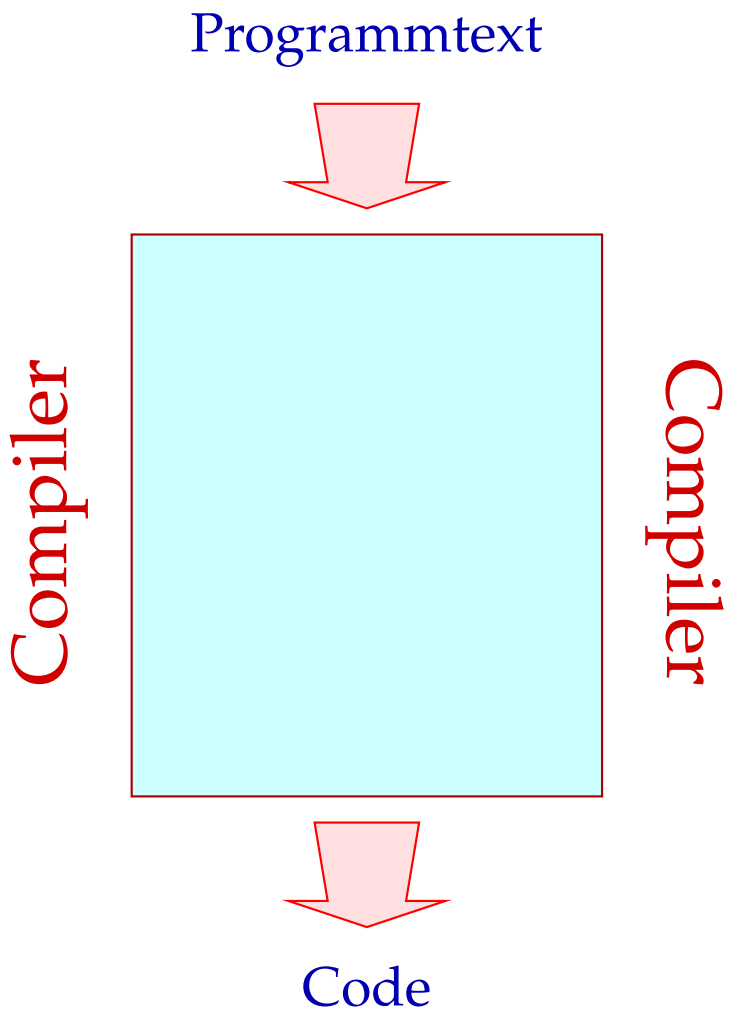
wobei $\emptyset \quad \widehat{=}$ leere Adress-Umgebung;

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

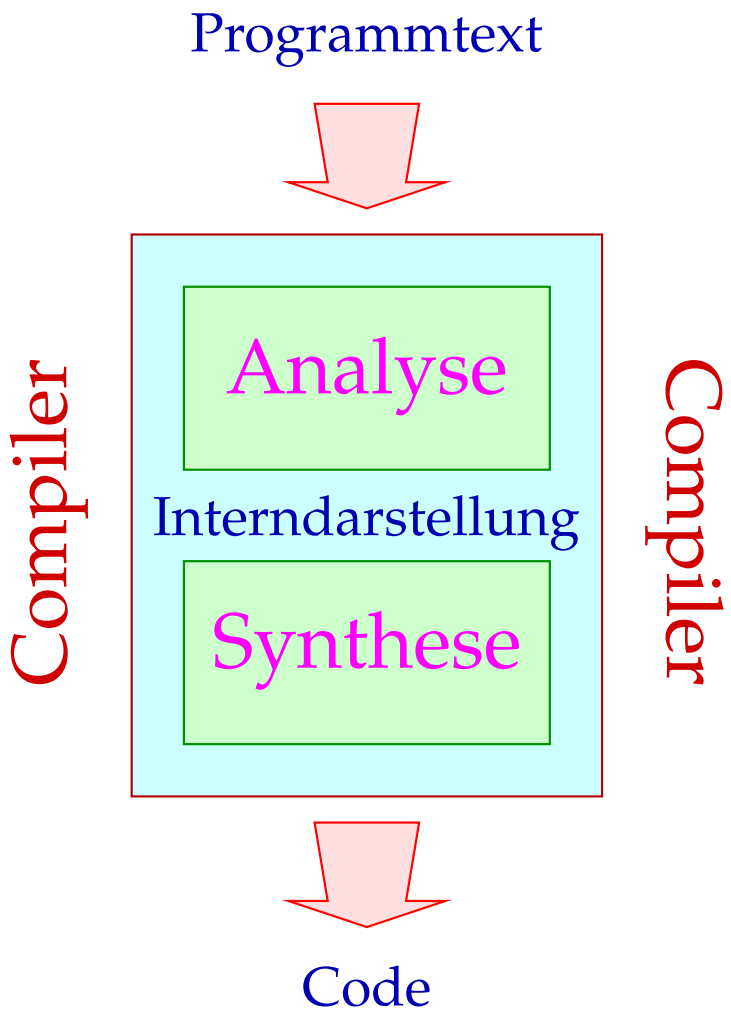
$\mathrm{k} \quad \widehat{=}$ Platz für globale Variablen

Die Analyse-Phase

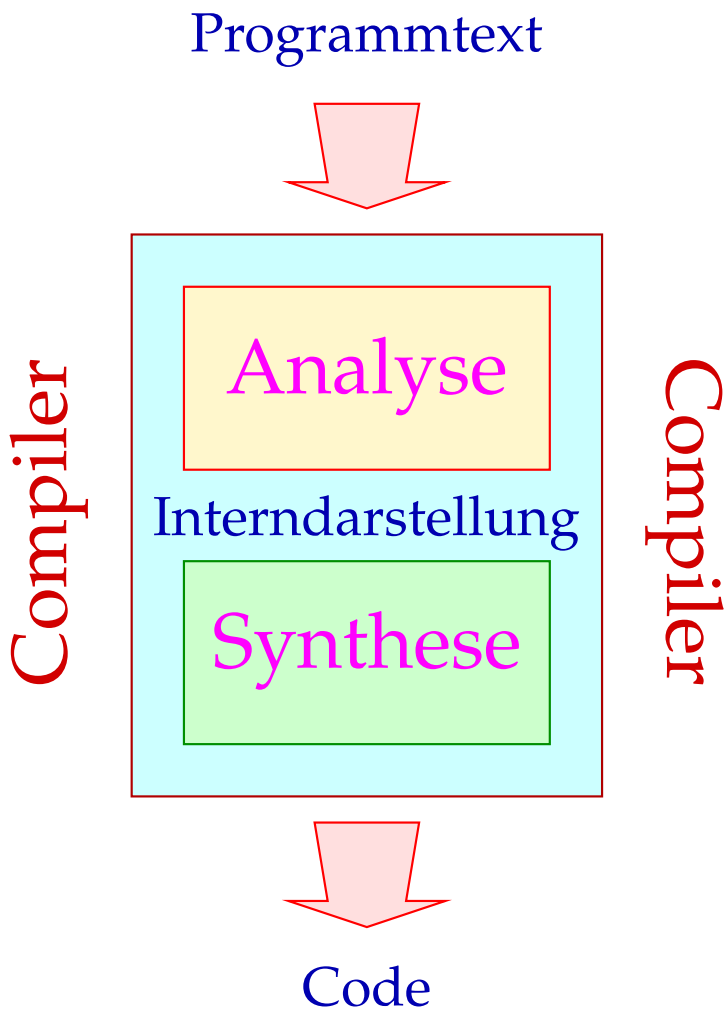
Orientierung:



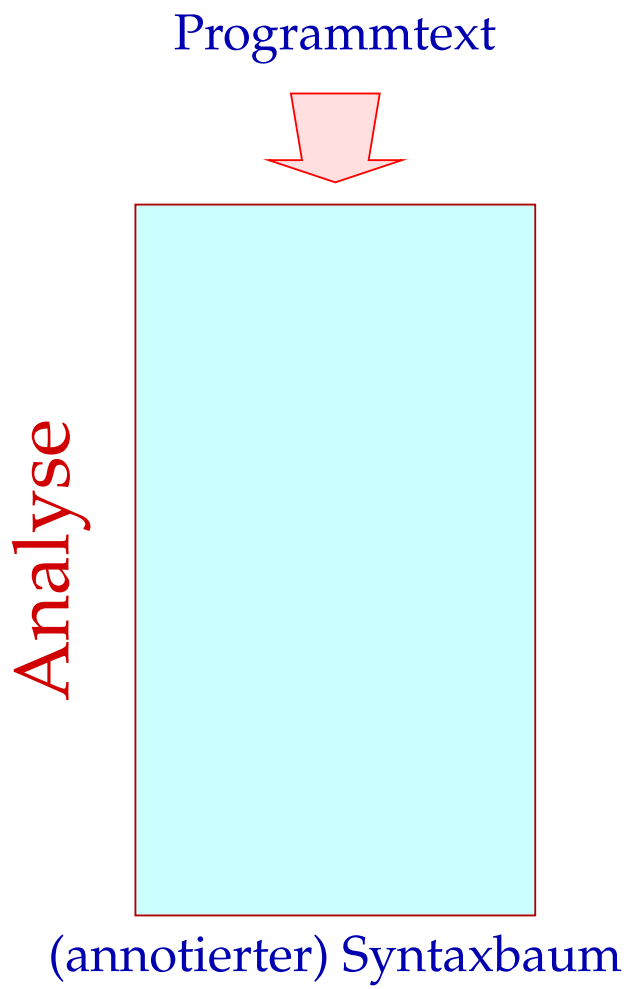
Orientierung:



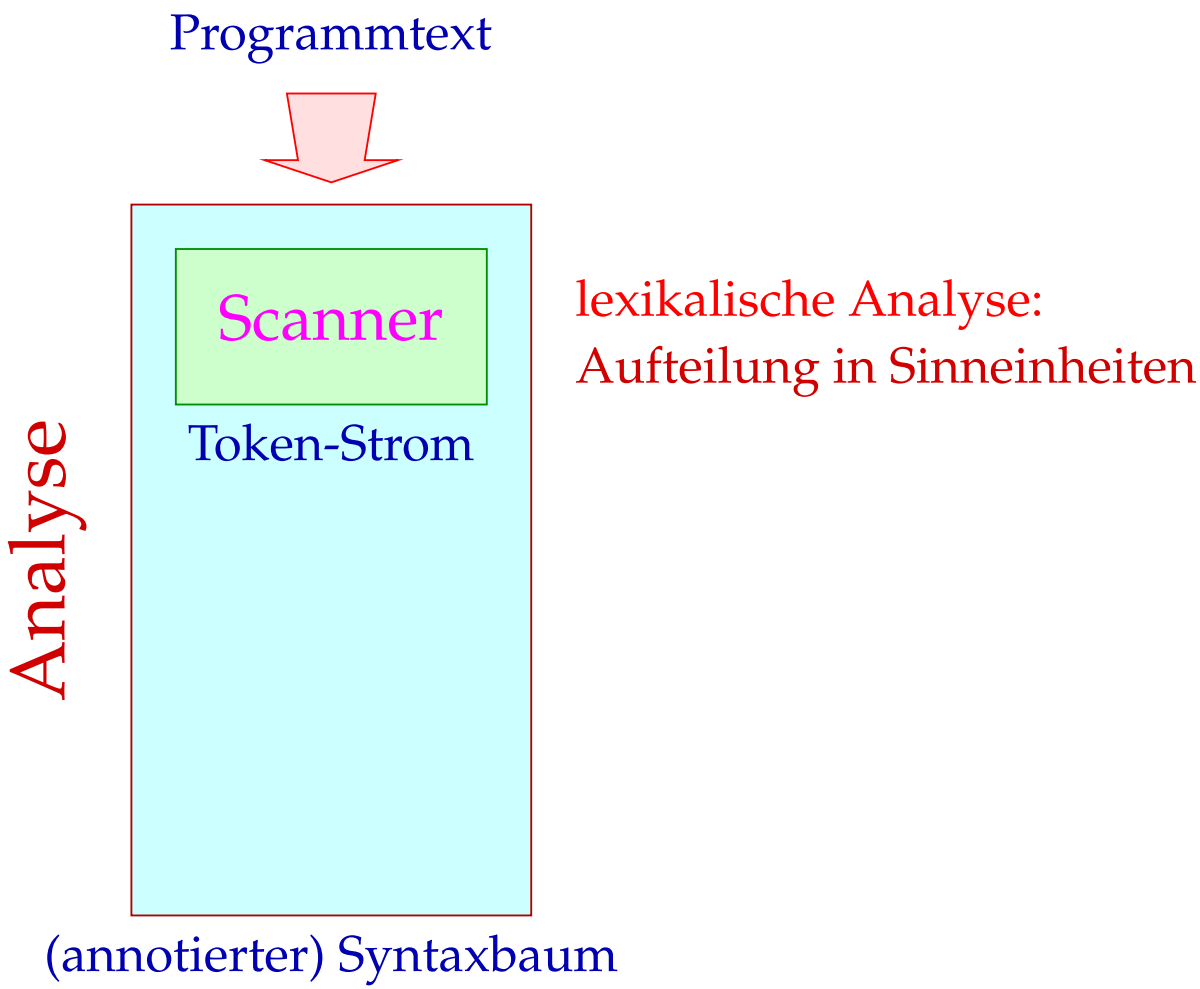
Orientierung:



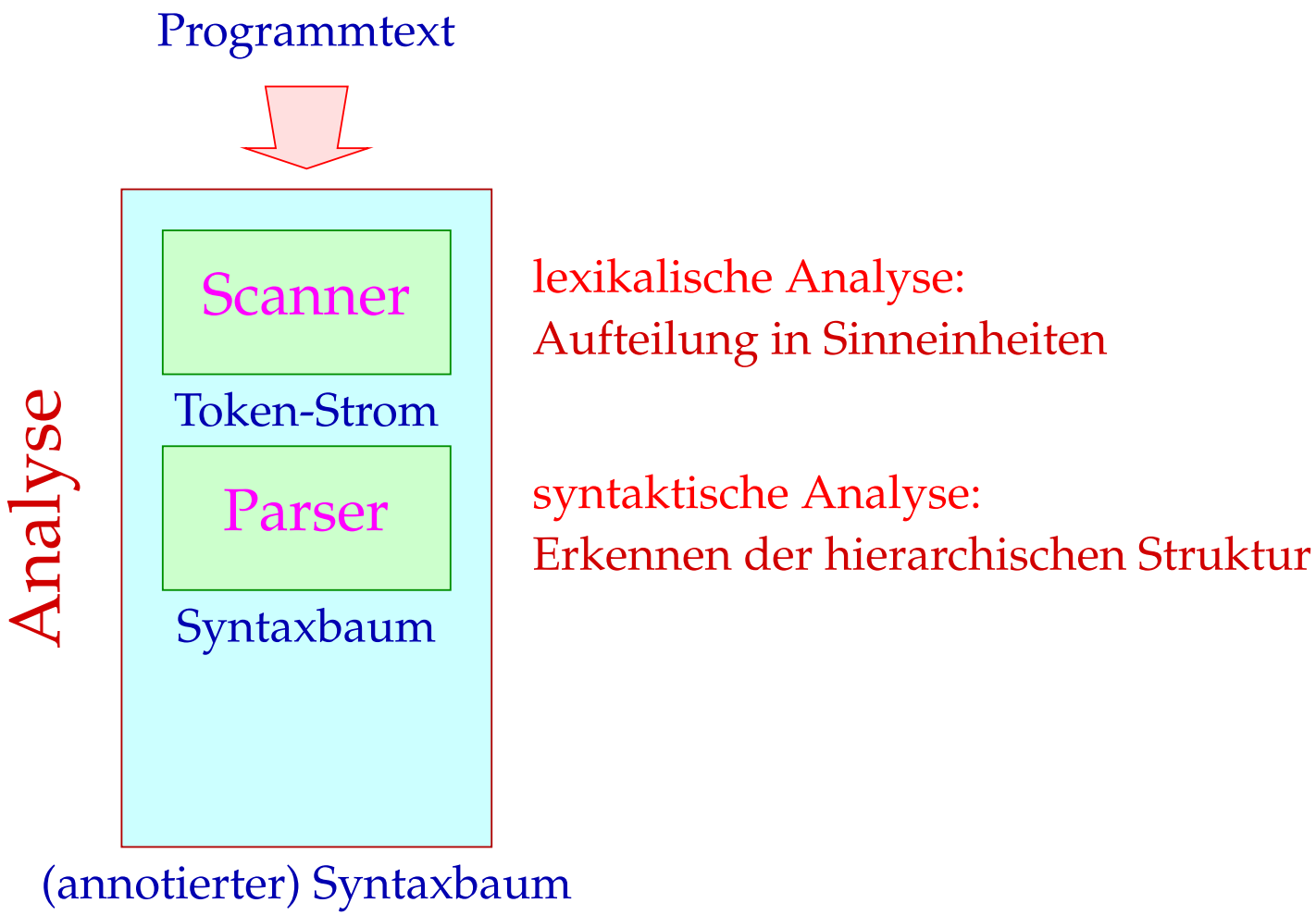
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



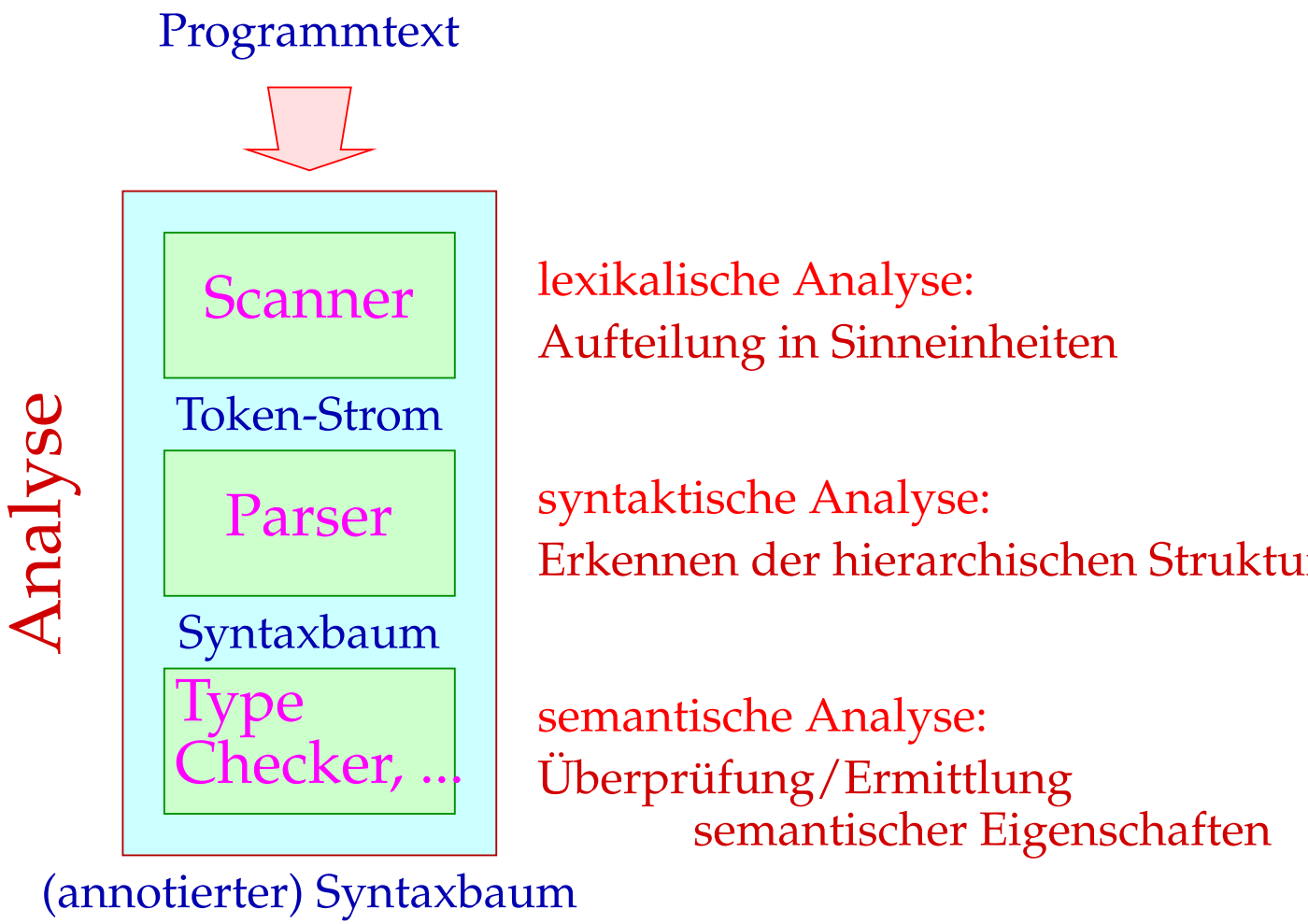
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



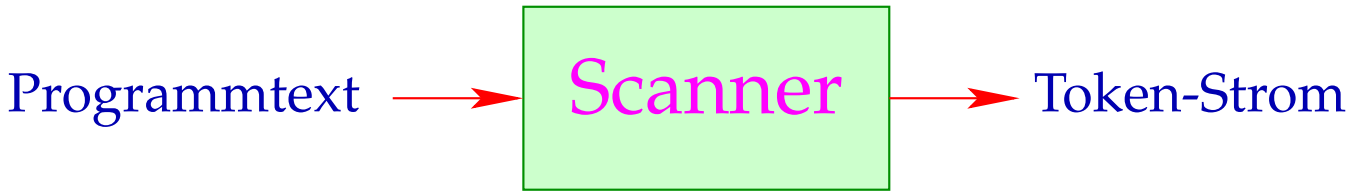
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



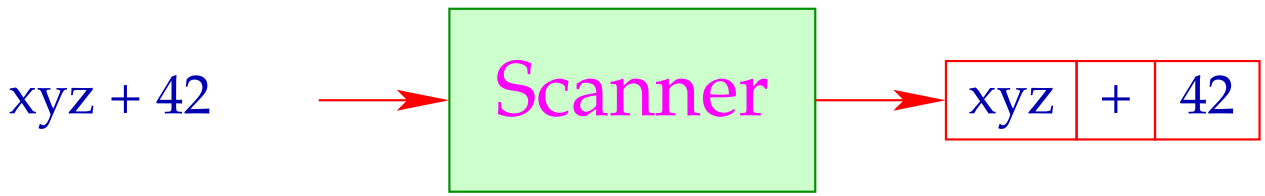
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



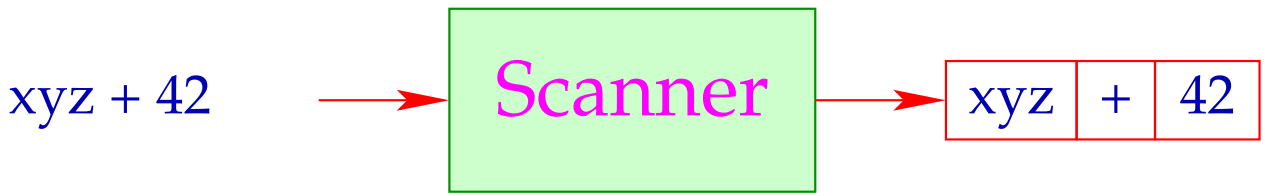
1 Die lexikalische Analyse



1 Die lexikalische Analyse

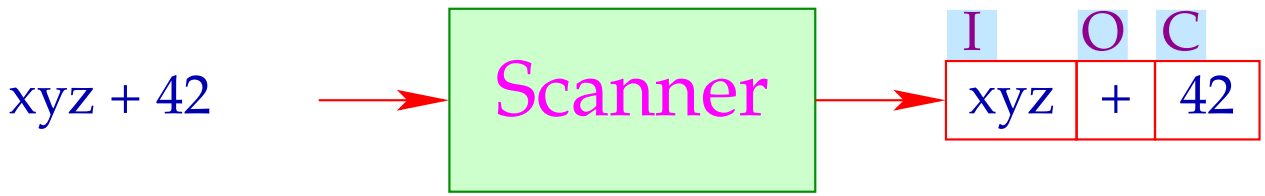


1 Die lexikalische Analyse



- 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:

  $\rightarrow$ Konstanten;

  $\rightarrow$ 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 :-)

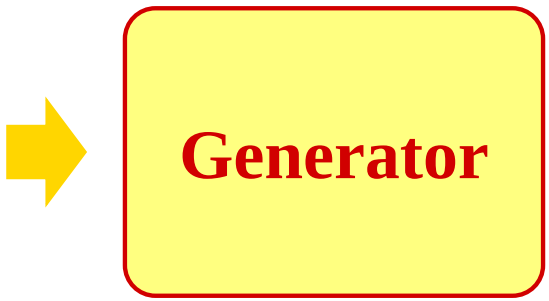
→ Generierung statt Implementierung lohnt sich nur für Routine-Aufgaben

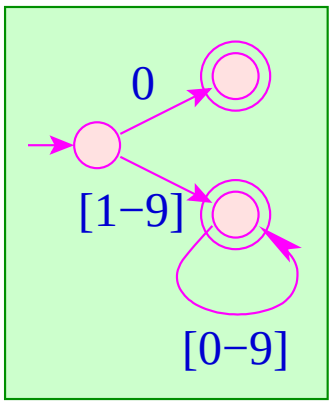
... und ist nur für Probleme möglich, die sehr gut verstanden sind :-(

... in unserem Fall:

... in unserem Fall:

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





Spezifikation von Token-Klassen: Reguläre Ausdrücke;

Generierte Implementierung: Endliche Automaten + X :-)

1.1 Grundlagen: Reguläre Ausdrücke

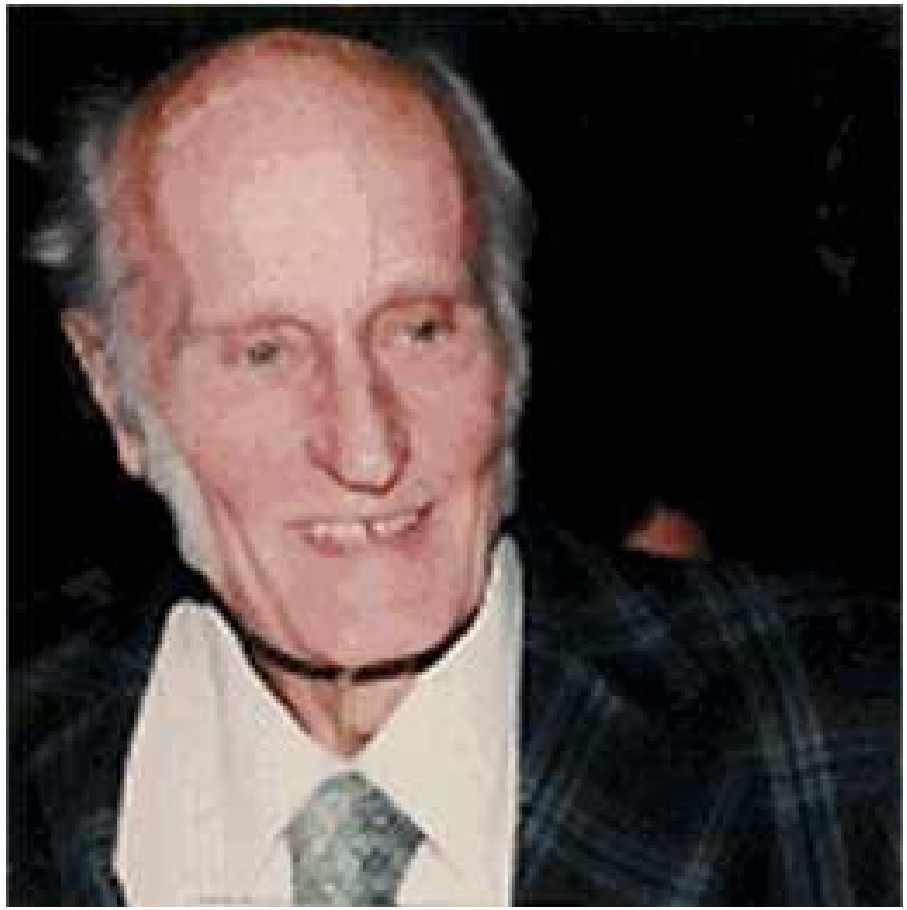
- Programmtext benutzt ein endliches Alphabet Σ von Eingabe-Zeichen, z.B. ASCII :-)
- Die Menge der Textabschnitte einer Token-Klasse ist i.a. regulär.
- Reguläre Sprachen kann man mithilfe regulärer Ausdrücke spezifizieren.

1.1 Grundlagen: Reguläre Ausdrücke

- Programmtext benutzt ein endliches Alphabet Σ von Eingabe-Zeichen, z.B. ASCII :-)
- Die Menge der Textabschnitte einer Token-Klasse ist i.a. regulär.
- Reguläre Sprachen kann man mithilfe regulärer Ausdrücke spezifizieren.

Die Menge $\mathcal{E}_\Sigma$ der (nicht-leeren) regulären Ausdrücke ist die kleinste Menge $\mathcal{E}$ mit:

- $\epsilon \in \mathcal{E}$ (ϵ neues Symbol nicht aus Σ);
- $a \in \mathcal{E}$ für alle $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), {e_1}^* \in \mathcal{E}$ sofern $e_1, e_2 \in \mathcal{E}$.



Stephen Kleene, Madison Wisconsin, 1909-1994

Beispiele:

$((a \cdot b^*) \cdot a)$

$(a \mid b)$

$((a \cdot b) \cdot (a \cdot b))$

Beispiele:

$$((a \cdot b^*) \cdot a)$$

$$(a \mid b)$$

$$((a \cdot b) \cdot (a \cdot b))$$

Achtung:

- Wir unterscheiden zwischen Zeichen $a, 0, |, \ldots$ und Meta-Zeichen $(, |,), \ldots$
- Um (hässliche) Klammern zu sparen, benutzen wir Operator-Präzedenzen:

  $$^* > \cdot > |$$

  und lassen "$\cdot$" weg :-)

Beispiele:

$$((a \cdot b^*) \cdot a)$$

$$(a \mid b)$$

$$((a \cdot b) \cdot (a \cdot b))$$

Achtung:

- Wir unterscheiden zwischen Zeichen $a, 0, |$,... und Meta-Zeichen $(, |,)$,...
- Um (hässliche) Klammern zu sparen, benutzen wir Operator-Präzedenzen:

  $$^* > \cdot > |$$

  und lassen "$\cdot$" weg :-)
- Reale Spezifikations-Sprachen bieten zusätzliche Konstrukte wie:

  $$e? \equiv (\epsilon \mid e)$$
  $$e^+ \equiv (e \cdot e^*)$$

  und verzichten auf "ϵ" :-)

Spezifikationen benötigen eine Semantik :-)

Im Beispiel:

Spezifikation	Semantik
ab^*a	$\{ab^n a \mid n \geq 0\}$
$a \mid b$	$\{a, b\}$
$abab$	$\{abab\}$

Für $e \in \mathcal{E}_\Sigma$ definieren wir die spezifizierte Sprache $[\![e]\!] \subseteq \Sigma^*$ induktiv durch:

$$
\begin{aligned}
[\![\epsilon]\!] &= \{\epsilon\} \\
[\![a]\!] &= \{a\} \\
[\![e^*]\!] &= ([\![e]\!])^* \\
[\![e_1|e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!] \\
[\![e_1 \cdot e_2]\!] &= [\![e_1]\!] \cdot [\![e_2]\!]
\end{aligned}
$$

Beachte:

- Die Operatoren $(\_)^*, \cup, \cdot$ sind die entsprechenden Operationen auf Wort-Mengen:

$$\begin{aligned} (L)^* &= \{w_1 \dots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{aligned}$$

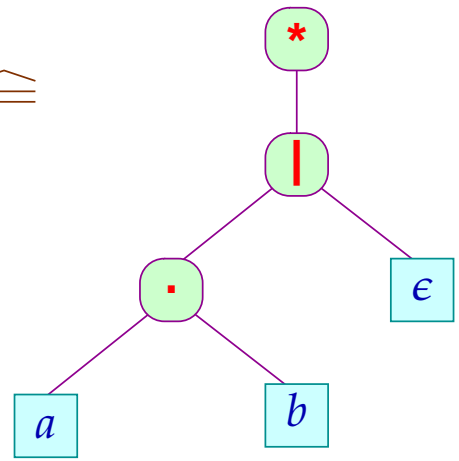
Beachte:

- Die Operatoren $(\_)^*, \cup, \cdot$ sind die entsprechenden Operationen auf Wort-Mengen:

$$\begin{aligned} (L)^* &= \{w_1 \dots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{aligned}$$

- Reguläre Ausdrücke stellen wir intern als markierte geordnete Bäume dar:

$(ab|\epsilon)^*$ $\widehat{=}$



Innere Knoten: Operator-Anwendungen;

Blätter: einzelne Zeichen oder ϵ.

Finger-Übung:

Zu jedem regulären Ausdruck e können wir einen Ausdruck e' (evt. mit "?") konstruieren so dass:

- $[\![e]\!] = [\![e']\!]$;
- Falls $[\![e]\!] = \{\epsilon\}$, dann ist $e' \equiv \epsilon$;
- Falls $[\![e]\!] \neq \{\epsilon\}$, dann enthält e' kein "ϵ".

Finger-Übung:

Zu jedem regulären Ausdruck e können wir einen Ausdruck e' (evt. mit "?") konstruieren so dass:

- $[\![e]\!] = [\![e']\!]$;
- Falls $[\![e]\!] = \{\epsilon\}$, dann ist $e' \equiv \epsilon$;
- Falls $[\![e]\!] \neq \{\epsilon\}$, dann enthält e' kein "ϵ".

Konstruktion:

Wir definieren eine Transformation $\mathcal{T}$ von regulären Ausdrücken durch:

$$
\begin{array}{lcl}
\mathcal{T}[\epsilon] & = & \epsilon \\
\mathcal{T}[a] & = & a \\
\mathcal{T}[e_1|e_2] & = & \mathsf{case}\ (\mathcal{T}[e_1], \mathcal{T}[e_2])\ \mathsf{of} \\
 & & \quad (\epsilon, \epsilon) : \epsilon \\
 & & \quad | \quad (e'_1, \epsilon) : e'_1? \\
 & & \quad | \quad (\epsilon, e'_2) : e'_2? \\
 & & \quad | \quad (e'_1, e'_2) : (e'_1 \mid e'_2) \\
\mathcal{T}[e_1 \cdot e_2] & = & \mathsf{case}\ (\mathcal{T}[e_1], \mathcal{T}[e_2])\ \mathsf{of} \\
 & & \quad (\epsilon, \epsilon) : \epsilon \\
 & & \quad | \quad (e'_1, \epsilon) : e'_1 \\
 & & \quad | \quad (\epsilon, e'_2) : e'_2 \\
 & & \quad | \quad (e'_1, e'_2) : (e'_1 \cdot e'_2) \\
\mathcal{T}[e^*] & = & \mathsf{case}\ \mathcal{T}[e]\ \mathsf{of} \\
 & & \quad \epsilon : \epsilon \\
 & & \quad | \quad e_1 : e_1{}^* \\
\mathcal{T}[e?] & = & \mathsf{case}\ \mathcal{T}[e]\ \mathsf{of} \\
 & & \quad \epsilon : \epsilon \\
 & & \quad | \quad e_1 : e_1?
\end{array}
$$

Unsere Anwendung:

Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*
```

.

Unsere Anwendung:

Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*
```

Bemerkungen:

- "le" und "di" sind Zeichenklassen.
- Definierte Namen werden in "{", "}" eingeschlossen.
- Zeichen werden von Meta-Zeichen durch "\" unterschieden.

Unsere Anwendung:

Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le}|{di})*
```

Gleitkommazahlen:

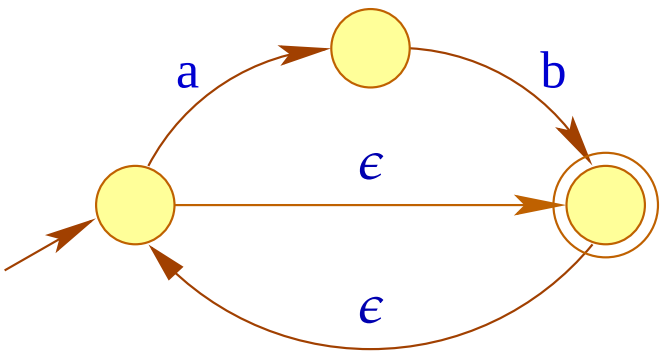
```
Float = {di}* (\.{di}|{di}\.) {di}*((e|E)(\+|\-)?{di}+)?
```

Bemerkungen:

- "le" und "di" sind Zeichenklassen.
- Definierte Namen werden in "{", "}" eingeschlossen.
- Zeichen werden von Meta-Zeichen durch "\" unterschieden.

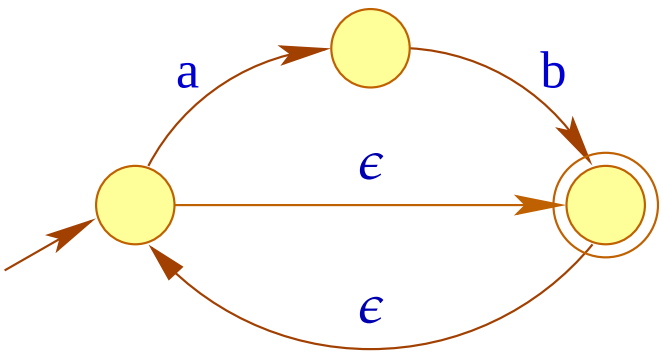
1.2 Grundlagen: Endliche Automaten

Beispiel:



1.2 Grundlagen: Endliche Automaten

Beispiel:



Knoten: Zustände;

Kanten: Übergänge;

Beschriftungen: konsumierter Input :-)

Michael O. Rabin, Stanford University

Dana S. Scott, Carnegy Mellon University, Pittsburgh

Formal ist ein nicht-deterministischer endlicher Automat mit ϵ-Übergängen (ϵ-NFA) ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

Formal ist ein nicht-deterministischer endlicher Automat mit ϵ-Übergängen (ϵ-NFA) ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

Für ϵ-NFAs ist:

$$\delta \quad \subseteq \quad Q \times (\Sigma \cup \{\epsilon\}) \times Q$$

Formal ist ein nicht-deterministischer endlicher Automat mit ϵ-Übergängen (ϵ-NFA) ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

Für ϵ-NFAs ist:

$$\delta \quad \subseteq \quad Q \times (\Sigma \cup \{\epsilon\}) \times Q$$

- Gibt es keine ϵ-Übergänge (p, ϵ, q), ist A ein NFA.
- Ist $\delta : Q \times \Sigma \to Q$ eine Funktion und $\#I = 1$, heißt A deterministisch (DFA).