Compilerbau

Martin Plümicke
Andreas Stadelmeier

SS 2025

Beschreibung

In der Vorlesung werden anwendungsnahe Konzepte und Techniken zu Programmiersprachen und Compilerbau vermittelt. Konkret werden zunächst die Phasen des Compilerbaus an Hand eines Java-Compilers vorgestellt. Als Implementierungstechnik wird die funktionale Programmiersprache Haskell verwendet. Dazu werden die notwendigen Grundlagen der funktionalen Programmierung aufbauend auf den Kenntissen der Grundvorlesung vermittelt. Im 2. Teil der Lehrveranstaltungen werden die Studierenden in Gruppenarbeit einen Mini-Java-Compiler mit den gelernten Techniken implementieren.

Literatur


Bauer and Höllerer.
Übersetzung objektorientierter Programmiersprachen.
Springer-Verlag, 1998, (in german).


Alfred V. Aho, Ravi Lam, Monica S.and Sethi, and Jeffrey D. Ullman.
Compiler: Prinzipien, Techniken und Werkzeuge.
Pearson Studium Informatik. Pearson Education Deutschland, 2. edition, 2008.
(in german).


Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers Principles, Techniques and Tools.
Addison Wesley, 1986.


Reinhard Wilhelm and Dieter Maurer.
Übersetzerbau.
Springer-Verlag, 2. edition, 1992.
(in german).

Literatur II

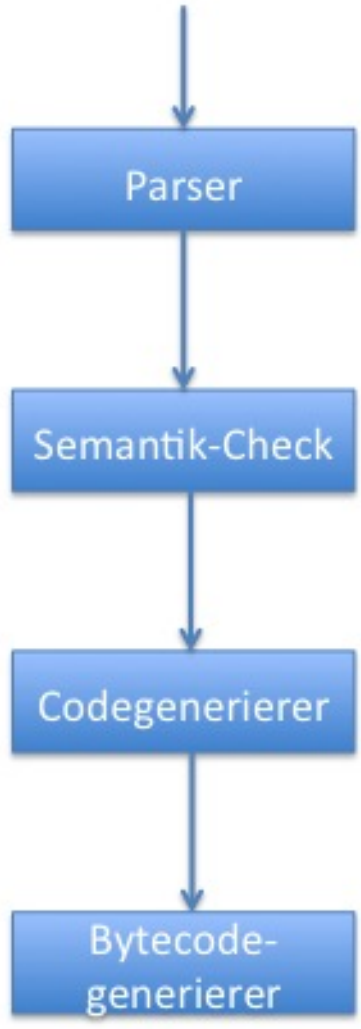

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.


Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java® Virtual Machine Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.

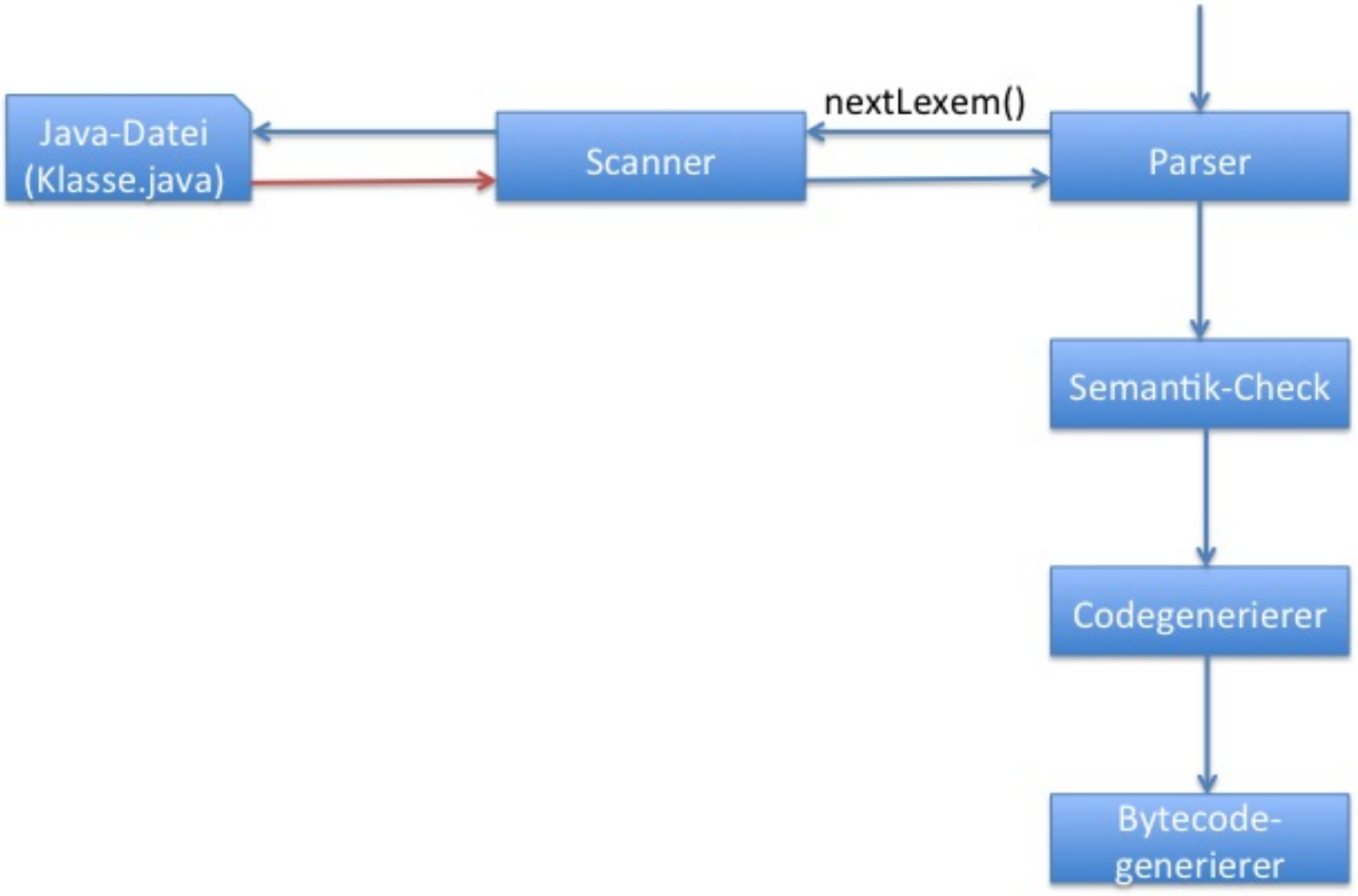

Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.


Peter Thiemann.
Grundlagen der funktionalen Programmierung.
Teubner, 1994.

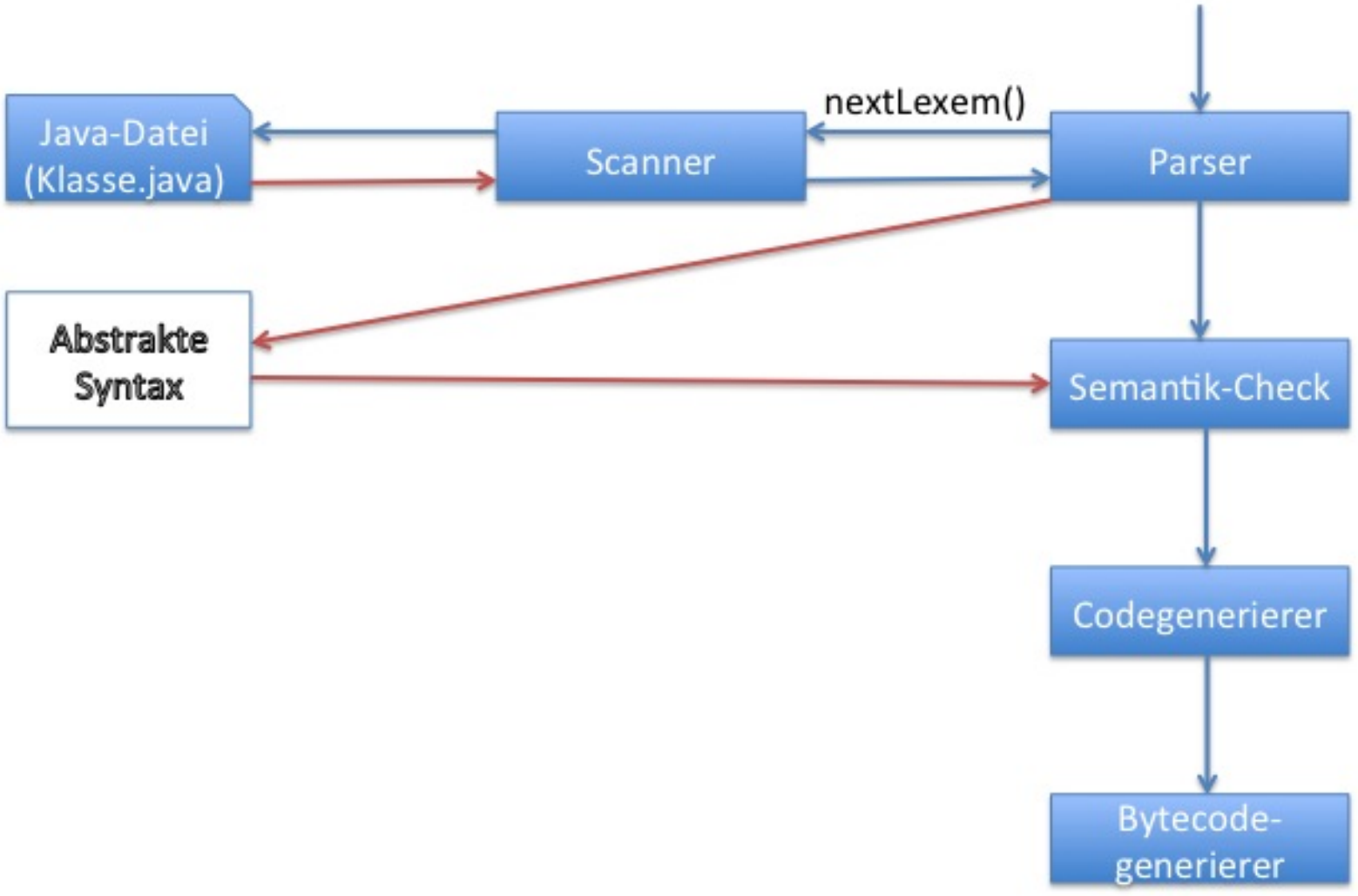
Compiler Überblick



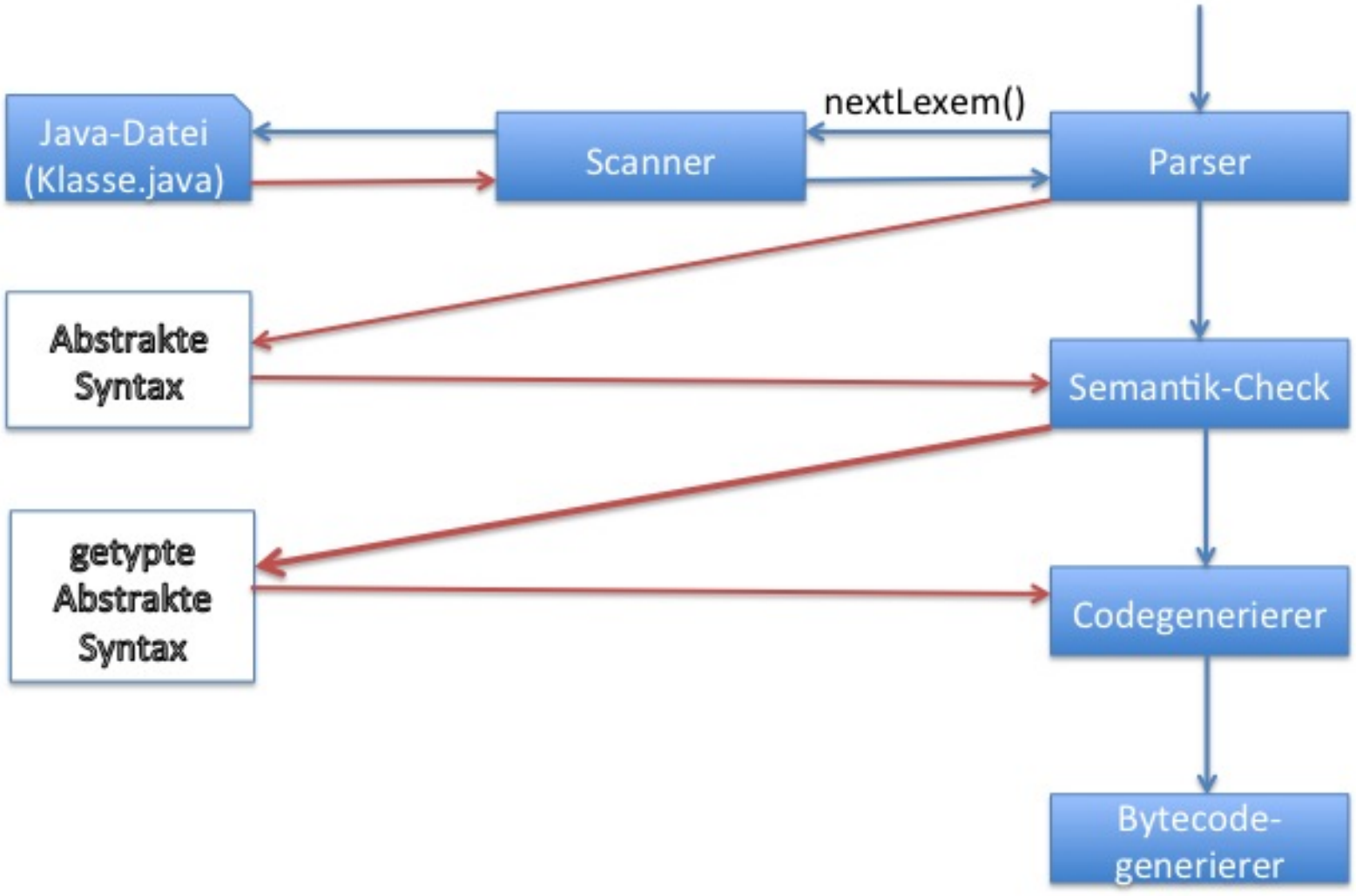
Compiler Überblick



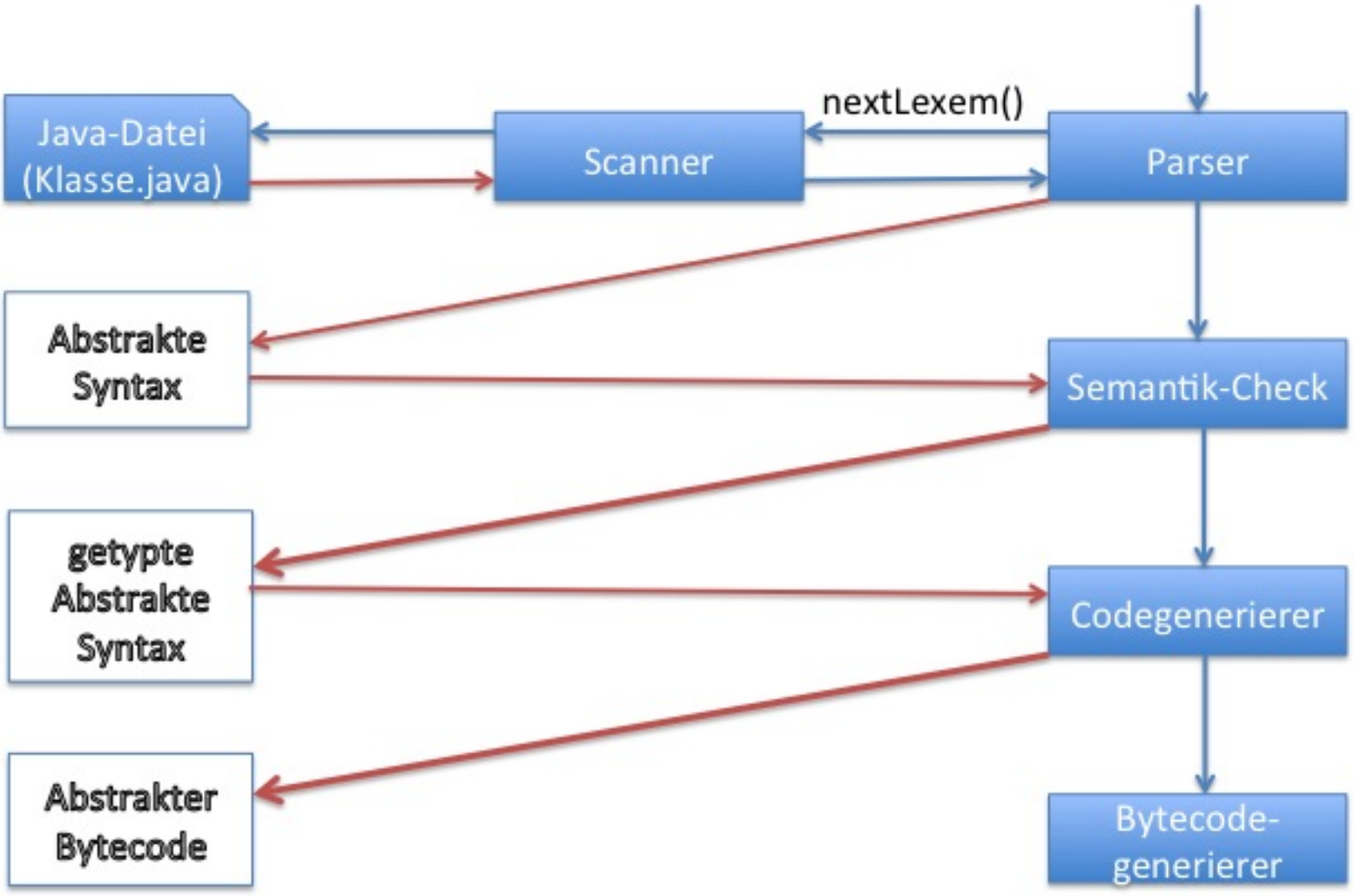
Compiler Überblick



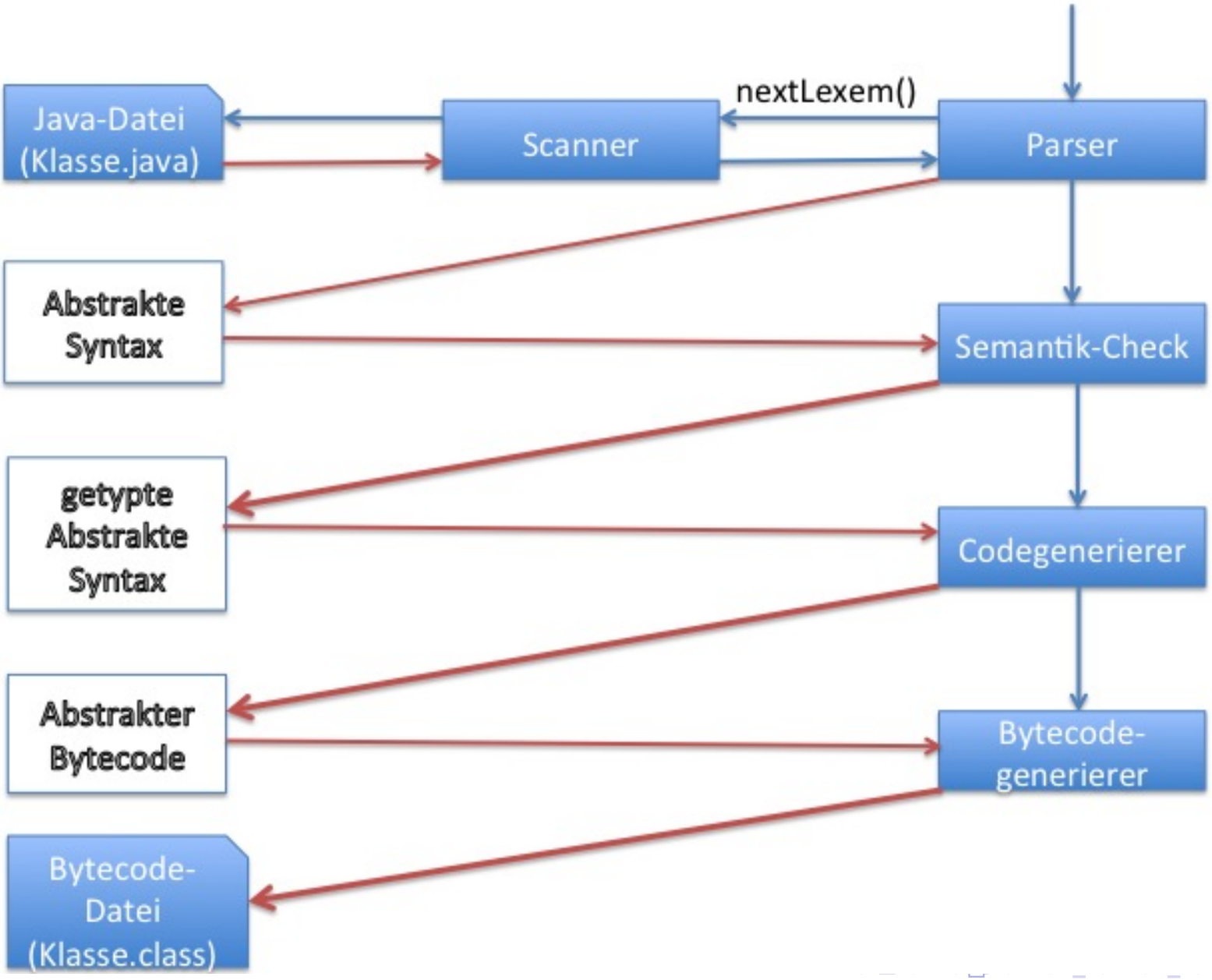
Compiler Überblick



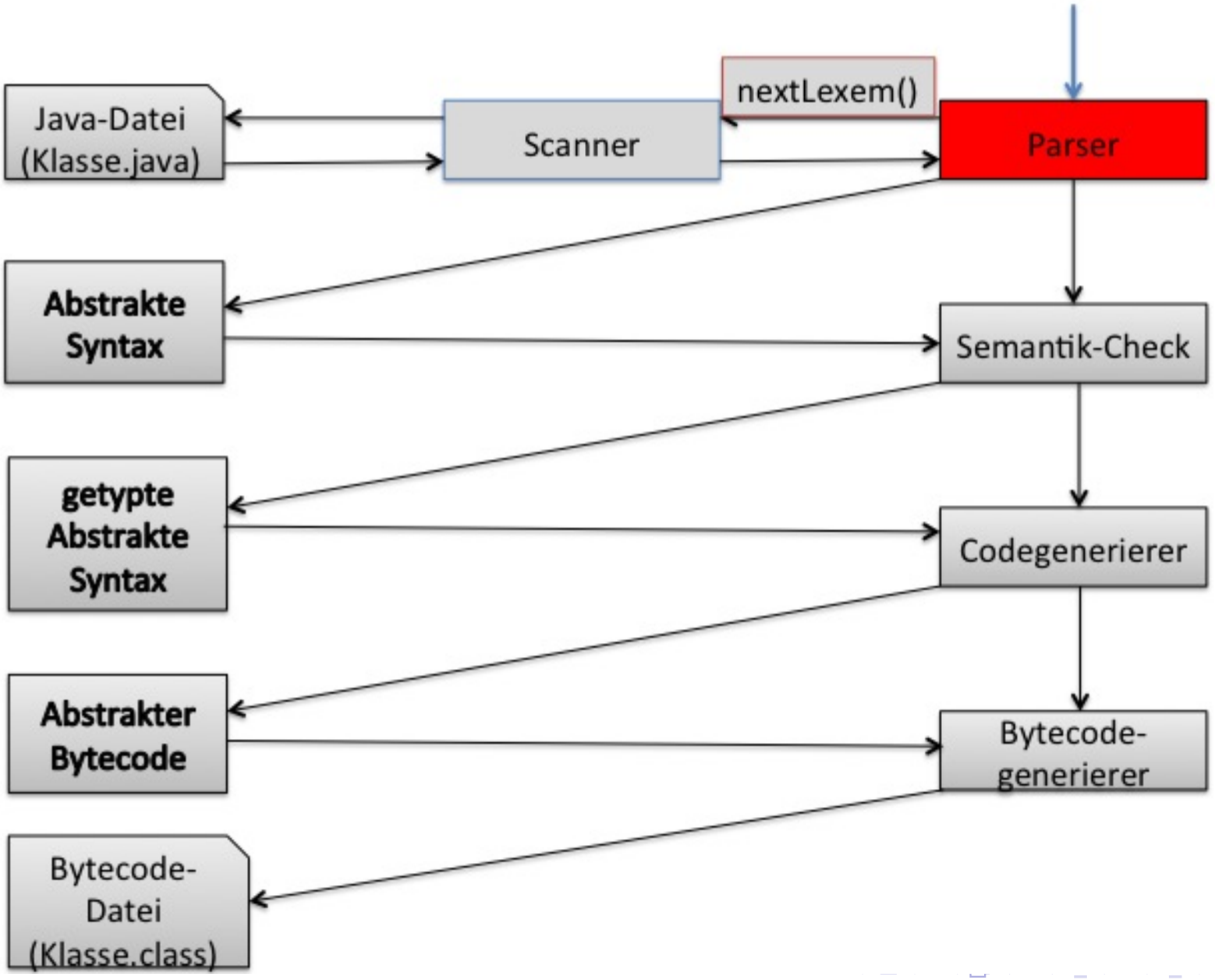
Compiler Überblick



Compiler Überblick



Parser


Java-Datei
(Klasse.java)
Scanner
nextLexem()
Parser
Abstrakte
Syntax
Semantik-Check
getypte
Abstrakte
Syntax
Codegenerierer
Abstrakter
Bytecode
Bytecode-
generierer
Bytecode-
Datei
(Klasse.class)

Programmiersprachen

Programmiersprachen werden als formale Sprachen über einem Alphabet von Tokens definiert.

Spezifikation eines Parser

Eingabe: Grammatik $G = (N, \Sigma, \Pi, S)$, $w \in \Sigma^*$

Ausgabe: $erg \in \{True, False\}$

Nachbedingung: $erg = (w \in \mathcal{L}(G))$

Mit anderen Worten: Es muss eine Ableitung $S \stackrel{*}{\rightarrow} w$ gefunden werden.

Beispiel

$G = (N, T, \Pi, S)$ mit

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

Beispiel

$G = (N, T, \Pi, S)$ mit

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

$w = cab$

Beispiel

$G = (N, T, \Pi, S)$ mit

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

$w = cab$

$\Rightarrow Ableitung : S \to cAb \to cab$

Beispiel

$G = (N, T, \Pi, S)$ mit

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

$w = cab$

$\Rightarrow Ableitung : S \to cAb \to cab$

Ergebnis: $erg = True$

Ansatz für einen Parser

Programmiersprachen werden als kontextfreie Grammatiken mit kontextsensitiven Nebenbedingungen beschrieben

Ansatz für einen Parser

Programmiersprachen werden als kontextfreie Grammatiken mit kontextsensitiven Nebenbedingungen beschrieben

- Cocke-Younger-Kasami–Algorithmus
  Nachteil:
  - Voraussetzung: Grammatik Chomsky-Normalform
  - Aufwand $O(n^3)$

Ansatz für einen Parser

Programmiersprachen werden als kontextfreie Grammatiken mit kontextsensitiven Nebenbedingungen beschrieben

- Cocke-Younger-Kasami–Algorithmus
  Nachteil:
  - Voraussetzung: Grammatik Chomsky-Normalform
  - Aufwand $O(n^3)$
- (Nicht deterministische) Push–Down–Automaten
  Nachteil:
  - Polynomialer Aufwand

$\Rightarrow$ Betrachtung einer Teilmenge der Chomsky-2–Sprachen (LR–Sprachen) (sind äquivalent zu den Sprachen, die durch deterministische Push–Down–Automaten erkannt werden.

$\Rightarrow$ Effiziente Implementierung möglich.

Ableitungsbaum

Man kann die Ableitung eines Wortes als Baum betrachten.

Ableitungsbaum

Man kann die Ableitung eines Wortes als Baum betrachten.

Aufbau:

Top-down: Linksableitungen (man erhält eine Ableitung, bei der immer das am weitesten links stehende Nichtterminal abgeleitet wird)

Ableitungsbaum

Man kann die Ableitung eines Wortes als Baum betrachten.

Aufbau:

Top-down: Linksableitungen (man erhält eine Ableitung, bei der immer das am weitesten links stehende Nichtterminal abgeleitet wird)

Bottom–Up: Rechtsableitungen (man erhält (rückwärts) eine Ableitung bei der immer das am weitesten rechts stehende Nichtterminal abgeleitet wird)

Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

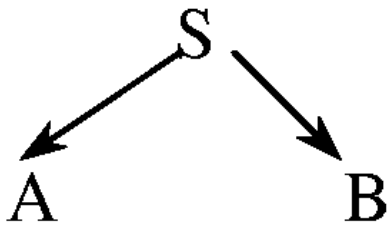
Eingabewort: $w = abc$

Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

Eingabewort: $w = abc$

$nexttoken() = a$

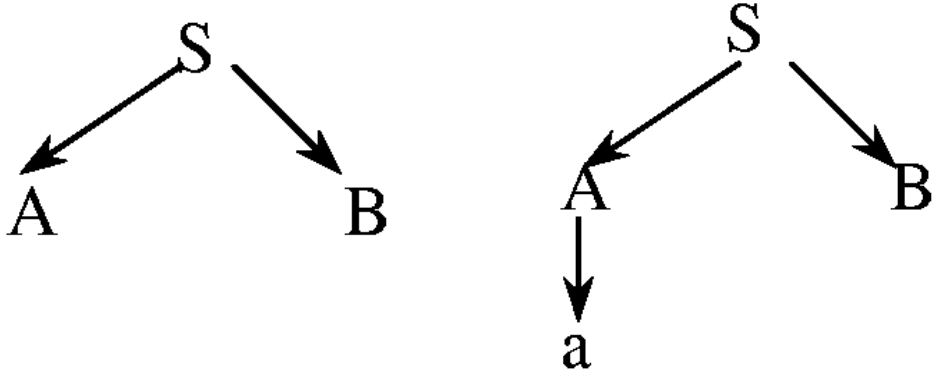


Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

Eingabewort: $w = abc$

$nexttoken() = a$

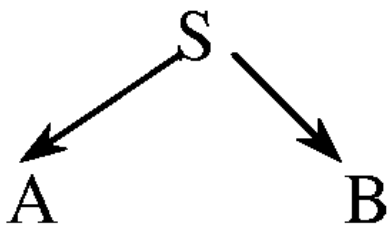


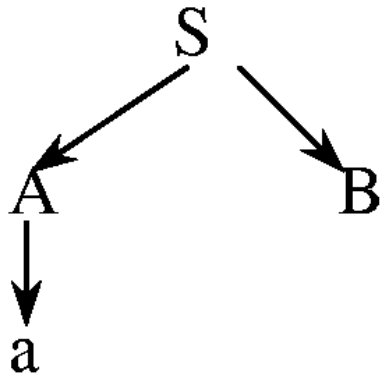
Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

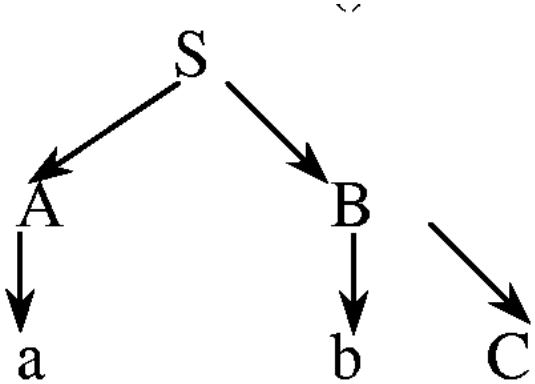
Eingabewort: $w = abc$

$nexttoken() = a$





$nexttoken() = b$

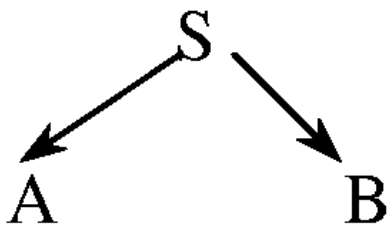


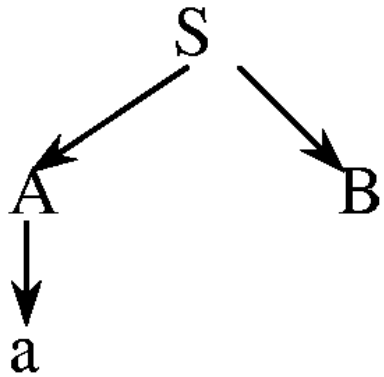
Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

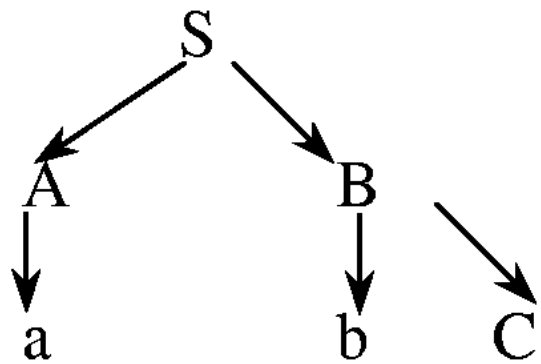
Eingabewort: $w = abc$

nexttoken() = *a*

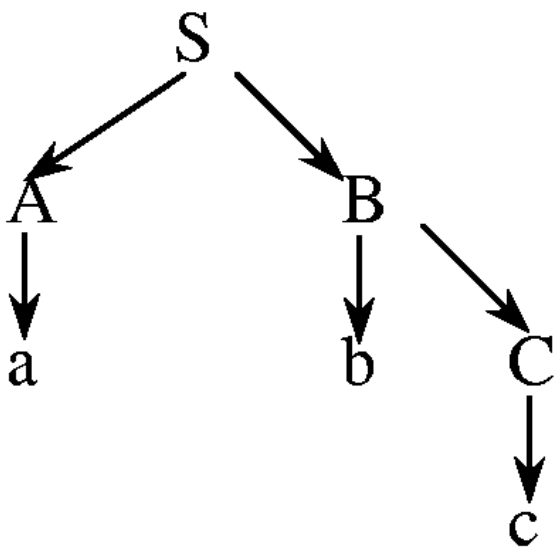




nexttoken() = *b*



nexttoken() = *c*



Recursive Decent–Syntaxanalyse

- Eingabe wird durch eine Menge rekursiver Funktionen abgearbeitet.
- Jedem Nichtterminal der Grammatik entspricht eine Funktion.
- Die Folge der Funktionsaufrufe bestimmt implizit den Ableitungsbaum.

Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$

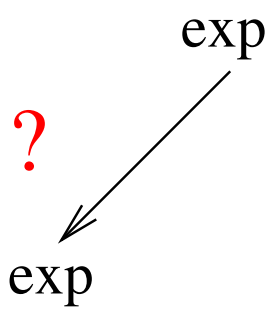
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



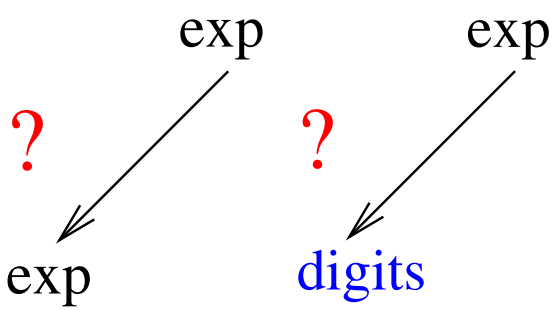
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



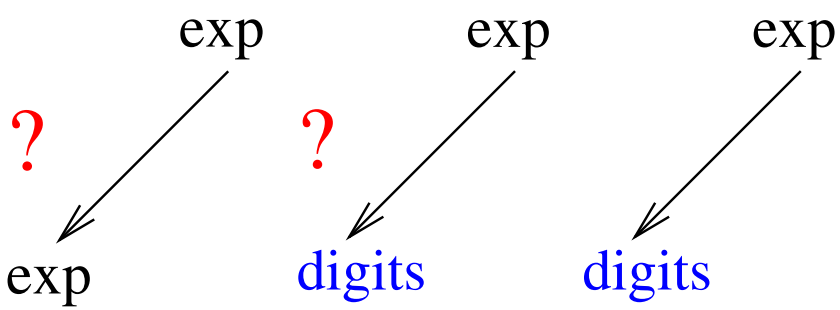
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



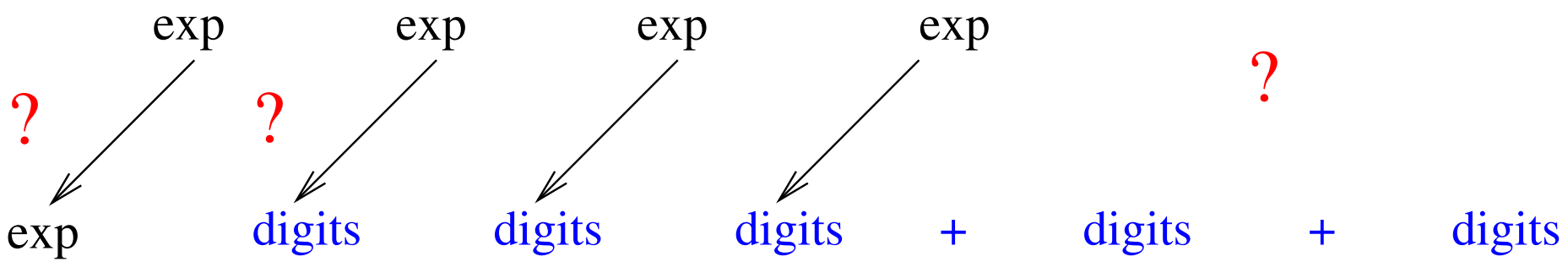
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$

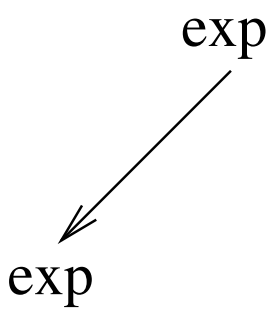
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



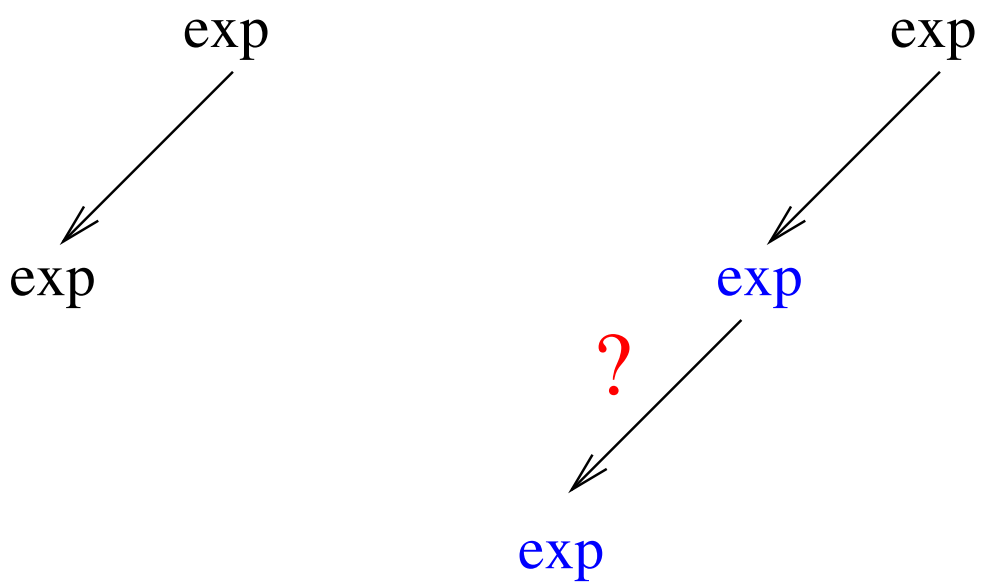
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



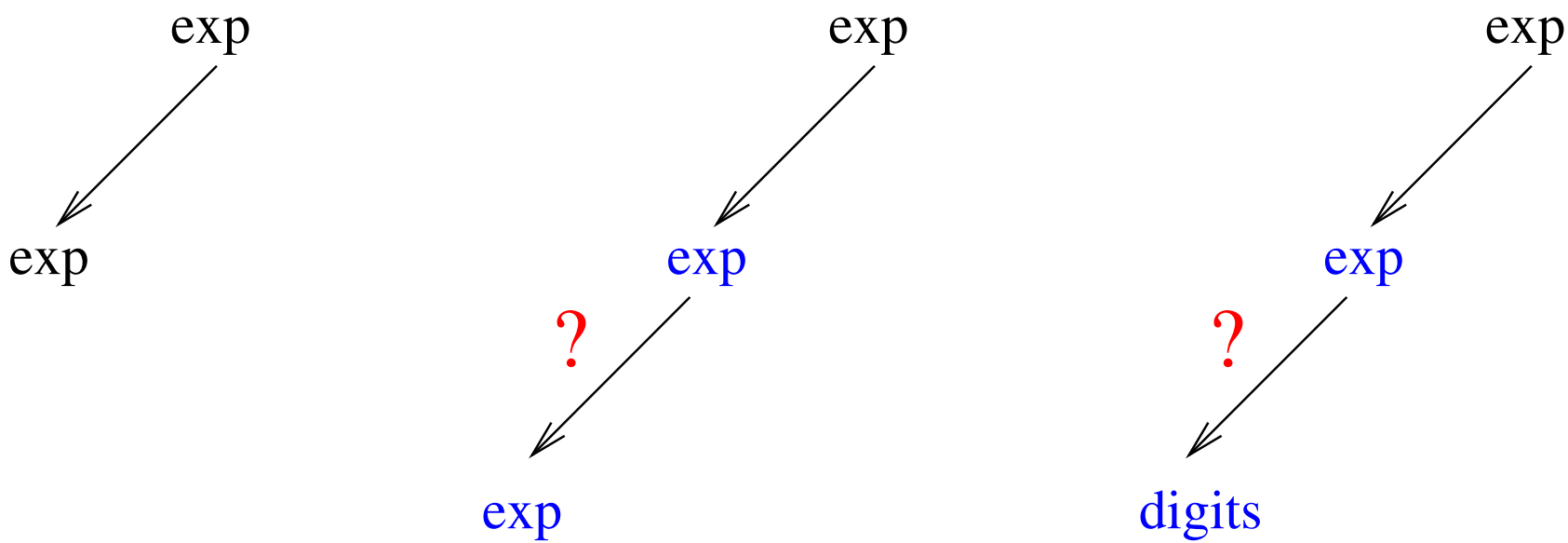
Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \{\ \mathit{Exp} \rightarrow\ \mathit{let\ var\ =\ Exp\ in\ Exp}$$
$$\mid\ \ \mathit{Exp\ +\ Exp}$$
$$\mid\ \ \mathit{var}$$
$$\mid\ \ \mathit{digits}\}$$

Problem: Linksrekursion

Eingabewort $1 + 1 + 1$



Elimination der Linksrekursion

$G = (N, T, \Pi, S)$ mit

$$\begin{aligned}\Pi = \{\ \mathit{Exp} \rightarrow\ & \mathit{let\ var\ =\ Exp\ in\ Exp} \\ \mid\ & \mathit{Exp\ +\ Exp} \\ \mid\ & \mathit{var} \\ \mid\ & \mathit{digits}\}\end{aligned}$$

Elimination der Linksrekursion

$G = (N, T, \Pi, S)$ mit

$$\begin{aligned}\Pi = \{\ Exp \rightarrow\ & let\ var\ =\ Exp\ in\ Exp \\ \mid\ & Exp\ +\ Exp \\ \mid\ & var \\ \mid\ & digits\}\end{aligned}$$

$$\begin{aligned}\Pi = \{\ Exp\ \ \rightarrow\ & TExp\ Exp' \\ Exp'\ \rightarrow\ & +\ TExp\ Exp' \\ \mid\ & \epsilon \\ TExp \rightarrow\ & let\ var\ =\ Exp\ in\ Exp \\ \mid\ & var \\ \mid\ & digits\ \}\end{aligned}$$

First

$G = (N, T, \Pi, S)$

$$First : (N \cup T)^* \to \mathcal{P}(T)$$

gegeben durch:

$First(a\beta) = \{a\}, \ falls\ a \in T, \beta \in (N \cup T)^*$

First

$G = (N, T, \Pi, S)$

$$First : (N \cup T)^* \to \mathcal{P}(T)$$

gegeben durch:

$First(a\beta) = \{a\}$, *falls* $a \in T, \beta \in (N \cup T)^*$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha),$$

falls $A \in N, (A \to \varepsilon) \notin \Pi, \beta, \alpha \in (N \cup T)^*$

First

$G = (N, T, \Pi, S)$

$$First : (N \cup T)^* \to \mathcal{P}(T)$$

gegeben durch:

$First(a\beta) = \{a\}$, *falls* $a \in T, \beta \in (N \cup T)^*$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha),$$

falls $A \in N, (A \to \varepsilon) \notin \Pi, \beta, \alpha \in (N \cup T)^*$

$$First(A\beta) = \left(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \right) \cup First(\beta),$$

falls $A \in N, (A \to \varepsilon) \in \Pi, \beta, \alpha \in (N \cup T)^*$

First

$G = (N, T, \Pi, S)$

$$First : (N \cup T)^* \to \mathcal{P}(T)$$

gegeben durch:

$First(a\beta) = \{a\}$, *falls* $a \in T, \beta \in (N \cup T)^*$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha),$$

falls $A \in N, (A \to \varepsilon) \notin \Pi, \beta, \alpha \in (N \cup T)^*$

$$First(A\beta) = \left(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \right) \cup First(\beta),$$

falls $A \in N, (A \to \varepsilon) \in \Pi, \beta, \alpha \in (N \cup T)^*$

$First(\gamma)$ bestimmt die Menge aller Terminale die an der 1. Stelle von γ ableitbar sind

First Beispiel

$$First(a\beta) = \{a\}, \qquad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \qquad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \qquad A \to \varepsilon \in \Pi$$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$First(\, aZ \,) =$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$$First(\, aZ \,) = \{\, a \,\}$$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$First(\, aZ \,) = \{\, a \,\}$
$First(\, Zb \,) =$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$First(\, aZ \,) = \{\, a \,\}$
$First(\, Zb \,) = \{\, a$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$$First(\, aZ \,) = \{\, a \,\}$$
$$First(\, Zb \,) = \{\, a, b \,\}$$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$$First(\, aZ \,) = \{\, a \,\}$$
$$First(\, Zb \,) = \{\, a, b \,\}$$
$$First(\, Rb \,) =$$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$$First(\, aZ \,) = \{\, a \,\}$$
$$First(\, Zb \,) = \{\, a, b \,\}$$
$$First(\, Rb \,) = \{\, a$$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$$First(\, aZ \,) = \{\, a \,\}$$
$$First(\, Zb \,) = \{\, a, b \,\}$$
$$First(\, Rb \,) = \{\, a, b$$

First Beispiel

$$First(a\beta) = \{a\}, \quad a \in T$$

$$First(A\beta) = \bigcup_{(A \to \alpha) \in \Pi} First(\alpha), \quad A \to \varepsilon \notin \Pi,$$

$$First(A\beta) = \Big(\bigcup_{\substack{(A \to \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \Big) \cup First(\beta), \quad A \to \varepsilon \in \Pi$$

$$\begin{aligned} \Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \mid \varepsilon \\ & R \to aR \mid bR \mid \varepsilon \} \end{aligned}$$

$$First(\, aZ \,) = \{\, a \,\}$$
$$First(\, Zb \,) = \{\, a, b \,\}$$
$$First(\, Rb \,) = \{\, a, b \,\}$$

Follow

$G = (N, T, \Pi, S)$

$$\mathit{Follow} : N \to \mathcal{P}(T)$$

$\mathit{Follow}(A)$ bestimmt die Menge aller Terminale, die in einem beliebigen Ableitungsschritt auf das Nichtterminal A folgen können.

Idee für einen *Recursive-Decent–Parser*

- Für jedes Nichtterminal wird eine Funktion definiert

Idee für einen *Recursive-Decent–Parser*

- Für jedes Nichtterminal wird eine Funktion definiert
- Die jeweils abzuarbeitende Produktion $A \rightarrow \gamma$ wird ausgewählt, wenn das nächste Symbol der Eingabe a in $First(\gamma)$ liegt

Idee für einen *Recursive-Decent–Parser*

- Für jedes Nichtterminal wird eine Funktion definiert
- Die jeweils abzuarbeitende Produktion $A \rightarrow \gamma$ wird ausgewählt, wenn das nächste Symbol der Eingabe a in $First(\gamma)$ liegt
- Rechte Seiten von Produktionen werden abgearbeitet:
  - Terminale werden mit dem nächsten Symbol der Eingabe verglichen
    Bei Übereinstimmung wird weitergelesen, sonst Abbruch mit `error`

Idee für einen *Recursive-Decent–Parser*

- Für jedes Nichtterminal wird eine Funktion definiert
- Die jeweils abzuarbeitende Produktion $A \rightarrow \gamma$ wird ausgewählt, wenn das nächste Symbol der Eingabe a in $First(\gamma)$ liegt
- Rechte Seiten von Produktionen werden abgearbeitet:
  - Terminale werden mit dem nächsten Symbol der Eingabe verglichen
    Bei Übereinstimmung wird weitergelesen, sonst Abbruch mit `error`
  - Für Nichtterminale wird die jeweilige Funktion aufgerufen

Anwendung auf die bekannte Grammatik

$$\begin{aligned}\Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \\ & R \to aR \mid bR \mid \varepsilon\}\end{aligned}$$

```
S('a':rest) = Z(rest)
S('b':rest) = Z(rest)

Z('a':rest) = R(rest)
```

Anwendung auf die bekannte Grammatik

$$\begin{aligned}\Pi = \{ & S \to aZ \mid bZ, \\ & Z \to aR \mid a \\ & R \to aR \mid bR \mid \varepsilon\}\end{aligned}$$

```
S('a':rest) = Z(rest)
S('b':rest) = Z(rest)

Z('a':rest) = R(rest)   oder if (rest==[]) then []   MEHRDEUTIG
```

Anwendung auf die bekannte Grammatik

$$\Pi = \{S \to aZ \mid bZ,$$
$$Z \to aR \mid a$$
$$R \to aR \mid bR \mid \varepsilon\}$$

```
S('a':rest) = Z(rest)
S('b':rest) = Z(rest)

Z('a':rest) = R(rest)  oder if (rest==[]) then []  MEHRDEUTIG
Z('b':rest) = error("'a' expected")

R('a':rest) = R(rest)
R('b':rest) = R(rest)
R([]) = []
```

LL(1)

Für zwei unterschiedliche Produktionen

$$A \to \alpha \mid \beta$$

muss gelten:

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- Wenn $\alpha \xrightarrow{*} \epsilon$, dann $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Wenn $\beta \xrightarrow{*} \epsilon$, dann $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

LL(1)

Für zwei unterschiedliche Produktionen

$$A \to \alpha \mid \beta$$

muss gelten:

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- Wenn $\alpha \xrightarrow{*} \epsilon$, dann $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Wenn $\beta \xrightarrow{*} \epsilon$, dann $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

Unsere bekannte Grammatik ist nicht LL(1), weil für

$$Z \to aR \mid a$$

- $a \in FIRST(aR)$
- $a \in FIRST(a)$

Recursive-Decent–Parser

Theorem

Für jede LL(1)–Grammatik ist in einem Recursive-Decent–Parser jeder Folgeschritt eindeutig bestimmt.

Beispiel

$G = (N, T, \Pi, expr)$ mit $N = \{expr, expr', texpr\}$,
$T = \{+, =, var, let, in, digits\}$ und
$\Pi = \{expr \rightarrow texpr\ expr'$
$expr' \rightarrow +\ texpr\ expr' \mid \epsilon$
$texpr \rightarrow let\ var = expr\ in\ expr$
$\mid var$
$\mid digits\}$

```
module LLParser where
import Scanner
import Control.Arrow

-- (>>>) :: (a-> b) -> (b -> c) -> a -> c
-- (f >>> g) (x) = g(f(x))

expr :: [Token] -> [Token]
expr eing = (texpr >>> expr') eing

expr' (PlusToken : rest) = (texpr >>> expr') rest
expr' (x) = x --epsilon
-- Follow(expr') = Follow(expr) = { in } \cup Follow(texpr) = {in, +, $}
-- => Grammatik ist nicht LL(1), aber da + assoziativ ist es kein
-- Problem, dass immer der 1. Fall genommen wird.


expr'(err:rest) = error(" '+' expected " ++ show (err:rest))

texpr(LetToken : rest) = (isVar >>> (next (AssignToken)) >>> expr >>> (
     next InToken) >>> expr) rest
texpr ((VarToken v):rest) = rest
texpr ((IntToken d):rest) = rest
```

```
texpr (toks) = error("Instead of " ++ show toks ++ "\n'let' or 'var' or '
      digit' expected")

next expecttok (tok:toks) = if (expecttok == tok) then toks
                            else error ((show expecttok) ++ " expected "
                                 ++ show (tok:toks));

isVar ((VarToken x):rest) = rest
isVar toks = error ("'Var' expected " ++ show toks);

parser = expr . alexScanTokens

main = do
  s <- readFile "fst.mfe"
  --s <- getContents
  --print ((expr . alexScanTokens) s)
  print ("Eingabe: " ++ s)
  print ("Scanner: " ++ (show (alexScanTokens s)))
  print  $ "Parser: " ++ (show (parser s))
```

LL(k)–Grammatik

Jeder Ableitungsschritt ist eindeutig durch die nächsten k Symbole der Eingabe (Lookahead) bestimmt.

LL(k)–Grammatik

Jeder Ableitungsschritt ist eindeutig durch die nächsten k Symbole der Eingabe (Lookahead) bestimmt.

Die Grammatik mit den Produktionen

$$Z \rightarrow aR \mid a$$

wäre LL(2):

```
Z('a':x:rest) = R(x:rest)
Z('a':[]) = []
Z('b':x:rest) = error("'a' expected")
```

Abschlussfrage

Wie sehen die Sprachen einer LL(0)–Grammatik aus?

Abschlussfrage

Wie sehen die Sprachen einer LL(0)–Grammatik aus?

Sie enthalten genau ein Wort!

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
           (=[a] -> [(a, [b])])
failure = _ -> []                        -- liefert immer fail
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
           (=[a] -> [(a, [b])])
failure = _ -> []                        -- liefert immer fail

succeed :: a -> Parser tok a             -- Parser der Sprache des
                  (=[tok] -> [(a, [tok])])
succeed value toks = [(value, toks)] -- leeren Worts (ϵ)
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
           (=[a] -> [(a, [b])])
failure = _ -> []                        -- liefert immer fail

succeed :: a -> Parser tok a             -- Parser der Sprache des
                 (=[tok] -> [(a, [tok])])
succeed value toks = [(value, toks)] -- leeren Worts (ϵ)

-- bedingte Erkennung
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy cond [] = []          (=[tok] -> [(tok, [tok])])
satisfy cond (tok : toks) | cond tok  = succeed tok toks
                          | otherwise = failure toks
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
           (=[a] -> [(a, [b])])
failure = _ -> []                        -- liefert immer fail

succeed :: a -> Parser tok a             -- Parser der Sprache des
                 (=[tok] -> [(a, [tok])])
succeed value toks = [(value, toks)] -- leeren Worts (ϵ)

-- bedingte Erkennung
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy cond [] = []          (=[tok] -> [(tok, [tok])])
satisfy cond (tok : toks) | cond tok  = succeed tok toks
                          | otherwise = failure toks

-- erkennen eines bestimmten Lexems (Terminals)
lexem :: Eq tok => tok -> Parser tok tok
lexem tok toks = satisfy ((==) tok) toks
```

Parser–Kombinatoren II

Umsetzen der Produktionen

```
-- nacheinander Erkennen
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)
(p1 +.+ p2) toks = [((v1, v2), rest2) | (v1, rest1) <- p1 toks,
                                        (v2, rest2) <- p2 rest1]
```

Parser–Kombinatoren II

Umsetzen der Produktionen

```
-- nacheinander Erkennen
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)
(p1 +.+ p2) toks = [((v1, v2), rest2) | (v1, rest1) <- p1 toks,
                                        (v2, rest2) <- p2 rest1]

 -- Alternative
 (|||) :: Parser tok a -> Parser tok a -> Parser tok a
 (p1 ||| p2) toks = p1 toks ++ p2 toks
```

Parser–Kombinatoren II

Umsetzen der Produktionen

```
-- nacheinander Erkennen
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)
(p1 +.+ p2) toks = [((v1, v2), rest2) | (v1, rest1) <- p1 toks,
                                        (v2, rest2) <- p2 rest1]
```

```
-- Alternative
(|||) :: Parser tok a -> Parser tok a -> Parser tok a
(p1 ||| p2) toks = p1 toks ++ p2 toks
```

Transfomation der Ergebnisse

```
(<<<) :: Parser tok a -> (a -> b) -> Parser tok b
(p <<< f) toks = [ (f v, rest) | (v, rest) <- p toks]
```

Beispiel Parser–Kombinatoren

Lexeme

```
data Token = LetToken
           | InToken
           | SymToken Char
           | VarToken String
           | IntToken Int

isVar (VarToken x) = True
isVar _ = False

isSym x (SymToken y) = x == y
isSym _ _ = False

isInt (IntToken n) = True
isInt _ = False

date Maybe a = Just a
             | Nothing
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow\ +\ TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow\ +\ TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???

texp :: Parser Token ???
texp = ((lexem LetToken) +.+ (satisfy isVar)
  +.+ (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken)
  +.+ expr)
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???

texp :: Parser Token ???
texp = ((lexem LetToken) +.+ (satisfy isVar)
  +.+ (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken)
  +.+ expr)
 ||| (satisfy isVar)
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???

texp :: Parser Token ???
texp = ((lexem LetToken) +.+ (satisfy isVar)
  +.+ (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken)
  +.+ expr)
 ||| (satisfy isVar)
 ||| (satisfy isInt)
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???

texp :: Parser Token ???
texp = ((lexem LetToken) +.+ (satisfy isVar)
  +.+ (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken)
  +.+ expr)
 ||| (satisfy isVar)
 ||| (satisfy isInt)
```

Typfehler!!!

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr')
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True

texp :: Parser Token Bool
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True

texp :: Parser Token Bool
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
 <<< (\(_,(_, (_, (_, (_, _))))) -> True)
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True

texp :: Parser Token Bool
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
 <<< (\(_,(_, (_, (_, (_, _))))) -> True)
 ||| ((satisfy isVar)
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True

texp :: Parser Token Bool
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
 <<< (\(_,(_, (_, (_, (_, _))))) -> True)
 ||| ((satisfy isVar)  <<< (\_ -> True))
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True

texp :: Parser Token Bool
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
 <<< (\(_,(_, (_, (_, (_, _))))) -> True)
 ||| ((satisfy isVar)  <<< (\_ -> True))
 ||| ((satisfy isInt)
```

Rückgabetyp Bool

```
expr :: Parser Token Bool
expr = (texp +.+ expr') <<< (\(_, _) -> True)

expr' :: Parser Token Bool
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(_, _)) -> True)
 ||| succeed True

texp :: Parser Token Bool
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
 <<< (\(_,(_, (_, (_, (_, _))))) -> True)
 ||| ((satisfy isVar)  <<< (\_ -> True))
 ||| ((satisfy isInt)  <<< (\_ -> True))
```

Anpassung Alex-Spezifikation

```
{
module Scanner (alexScanTokens, Token(..)) where
}

%wrapper "basic"

$digit = 0-9                    -- digits
$alpha = [a-zA-Z]               -- alphabetic characters

tokens :-
  $white+                       ;
  "--".*                        ;
  let                           { \s -> LetToken }
  in                            { \s -> InToken }
  $digit+                       { \s -> IntToken (read s) }
  [\=\+\-\*\/\(\)]              { \s -> SymToken (head s) }
  $alpha [$alpha $digit \_ ']*  { \s -> VarToken s }
```

```
{
data Token =
   LetToken
|  InToken
|  SymToken Char
|  VarToken String
|  IntToken Int
  deriving (Eq,Show)
}
```

main Funktion

```
-- nur wenn keine Tokens übrig sind ist die Loesung korrekt
correctsols :: [(t, [a])] -> [(t, [a])]
correctsols sols =
  (filter (\(_, resttokens) -> null resttokens)) sols

parser :: String -> MiniFunkExpr
parser = fst . head . correctsols. expr . alexScanTokens
```

main Funktion

```
-- nur wenn keine Tokens übrig sind ist die Loesung korrekt
correctsols :: [(t, [a])] -> [(t, [a])]
correctsols sols =
  (filter (\(_, resttokens) -> null resttokens)) sols

parser :: String -> MiniFunkExpr
parser = fst . head . correctsols. expr . alexScanTokens


main = do
  s <- readFile "Pfad/fst.mfe"
  print (parser s)
```

main Funktion

```
-- nur wenn keine Tokens übrig sind ist die Loesung korrekt
correctsols :: [(t, [a])] -> [(t, [a])]
correctsols sols =
  (filter (\(_, resttokens) -> null resttokens)) sols

parser :: String -> MiniFunkExpr
parser = fst . head . correctsols. expr . alexScanTokens


main = do
  s <- readFile "Pfad/fst.mfe"
  print (parser s)
```

Mögliche Eingabe: `fst.mfe`

```
let x = 10
in let y = 20
   in x + y
```

Abschlussbemerkung Kombinator-Parsen

- Es werden immer alle möglichen Ableitungen gebildet
  $\Rightarrow$
  - keine Vorausschau zur Endscheidung bei Alternativen
  - Es muss kein Backtracking programmiert werden
  - `head` in der Funktion `parser` führt dazu, dass nur die erste Lösung bestimmt wird.
  - Nicht-strikte Auswertung führt zu trotzdem effizienten Ergebnissen.

Abschlussbemerkung Kombinator-Parsen

- Es werden immer alle möglichen Ableitungen gebildet
  $\Rightarrow$
  - keine Vorausschau zur Endscheidung bei Alternativen
  - Es muss kein Backtracking programmiert werden
  - `head` in der Funktion `parser` führt dazu, dass nur die erste Lösung bestimmt wird.
  - Nicht-strikte Auswertung führt zu trotzdem effizienten Ergebnissen.

- Linksrekursive Grammatiken können zu Endlosrekusionen führen
  $\Rightarrow$ Auflösung von Linksrekursionen

Rechtsableitungen (bottom–up)

$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

Eingabewort: $w = abc$

Rechtsableitungen (bottom–up)

$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

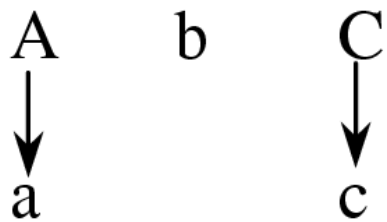
nexttoken$() = a$
wende an: $A \to a$

Rechtsableitungen (bottom–up)

$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$

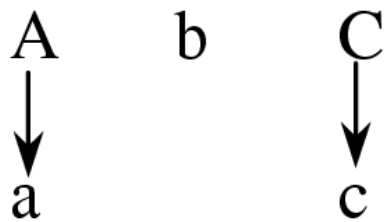


Rechtsableitungen (bottom–up)

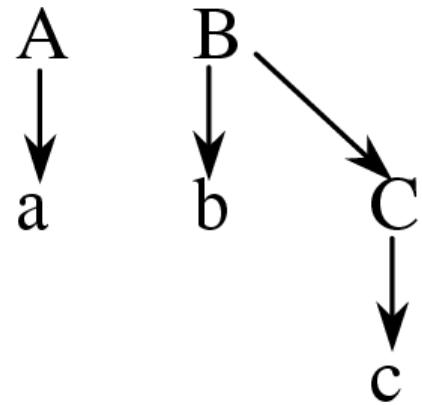
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$



wende an: $B \to bC$

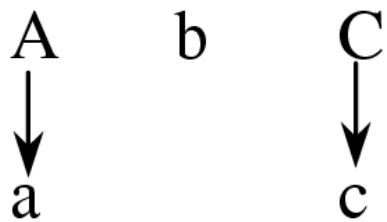


Rechtsableitungen (bottom–up)

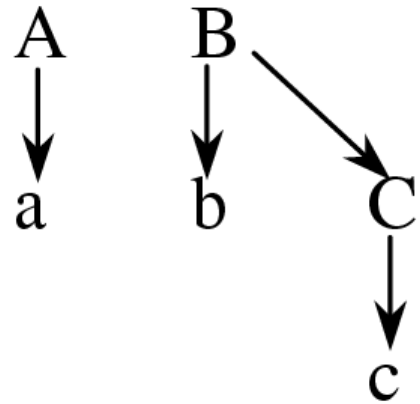
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

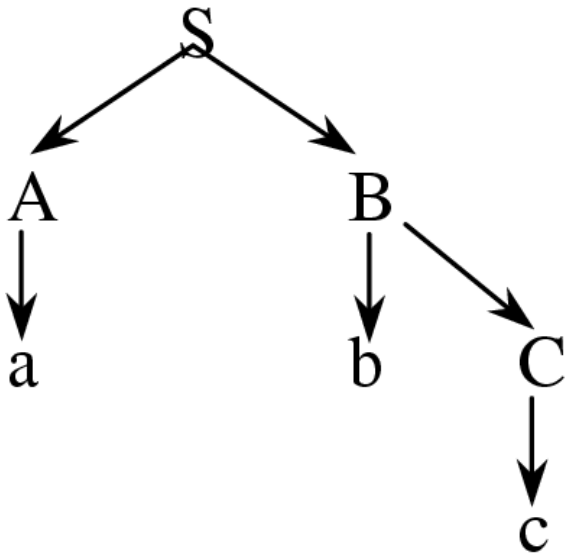
nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$



wende an: $B \to bC$



wende an: $S \to AB$

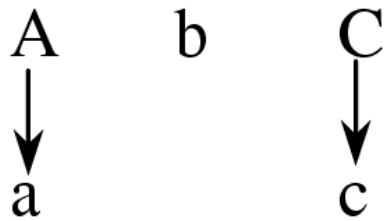


Rechtsableitungen (bottom–up)

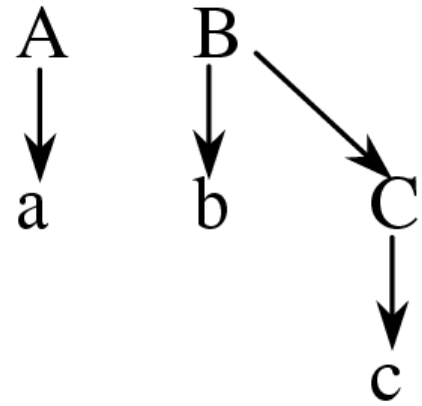
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

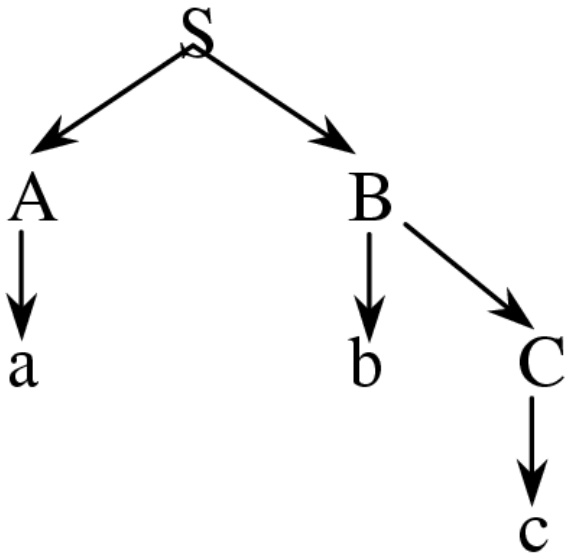
nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$



wende an: $B \to bC$



wende an: $S \to AB$



Betrachtet man die Konstruktion rückwärts:

$$\underline{S} \to A\underline{B} \to Ab\underline{C} \to \underline{A}bc \to abc,$$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de$

Shift–Reduce–Konflikt

$aAA.cde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt

$aAA.cde$

$aA.de \longleftarrow aAd.e$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt

$aAA.cde$

$aA.de \longleftarrow aAd.e \longleftarrow aAB.e$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt

$aAA.cde$

$aA.de \longleftarrow aAd.e \longleftarrow aAB.e$

Reduce–Reduce–Konflikt

$aAA.e$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt

$aAA.cde$

$aA.de \longleftarrow aAd.e \longleftarrow aAB.e \longleftarrow aABe \longleftarrow S$

Reduce–Reduce–Konflikt

$aAA.e$

Konfliktlösungsansätze

LR(0): Es muss ohne Vorausschau möglich sein zu entscheiden, ob geshiftet oder reduziert wird.

SLR(1): An Hand der Bildung der Menge aller möglichen folgenden Terminalsymbole auf ein Nichtterminal, wird entschieden ob reduziert wird.

LR(1): Für jeden möglichen Ableitungsschritt in einer Produktion wird die Menge der darauffolgenden Terminalsymbolde zur Unterscheidung betrachtet.

LALR(1): Es werden alle Mengen von LR(1)–Elementen zusammengefasst, die den gleichen Ableitungsschritt vollziehen.

$$LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) (= \mathcal{L}(\mathcal{DPDA}))$$

Parsergenerator Happy

- Yacc für Haskell
- **Aufruf:** `> happy -info JavaParser.y`
- Option `-info` erzeugt die Info-Datei: `nfo`

Das Happy-File

Haskell–Source–Code:

```
{
module Parser (parse-Funktion) where
}
```

- Das erzeugte Haskell–File definiert das Module *Parser*.
- Die Funktion *parse-Funktion* wird exportiert.

Deklarationen

```
%name { parse-Funktion }
%tokentype { Tokentyp }
%error { parseError-Funktion }
```

- `name`: Name der Parserfunktion.
- `tokentype`: Type der einzelnen Tokens, die der Parser liest.
- `error`: Name der Funktion, die bei einem Fehler aufgerufen wird.

Beispiel

Data–Deklaration des %tokentype's

Leicht modifizierte Datenstruktur in Scanner.x:

```
data Token =
  LetToken|
  InToken |
  PlusToken |
  AssignToken |
  VarToken String  |
  IntToken Int
  deriving (Eq,Show)
```

Tokens

```
%token
    Let { LetToken }
    In { InToken }
    Plus { PlusToken }
    Assign { AssignToken }
    Var  { VarToken $$  }
    Int { IntToken $$ }
```

- Die Tokens werden definiert durch das Paar
  - *Terminal* in der Grammatik (links)
  - *Haskell-Konstruktor des Typs* `%tokentype` (rechts in geschweifter Klammer)
- Wert des Tokens: normalerweise das Token selbst
  $$ bedeutet, der Wert ist das Argument des Tokens

Modifizierte Grammatik

Die Grammatik aus dem Kombinator-Parsen Beispiel

$$
\begin{aligned}
\Pi = \{\ & Exp \rightarrow TExp\ Exp' \\
& Exp' \rightarrow + TExp\ Exp' \\
& \quad\ \mid \epsilon \\
& TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \\
& \quad\ \mid var \\
& \quad\ \mid digits\ \}
\end{aligned}
$$

wird vereinfacht zu:

$$
\begin{aligned}
\Pi = \{\ & Exp \rightarrow let\ var\ =\ Exp\ in\ Exp \\
& \quad\ \mid Exp\ +\ Exp \\
& \quad\ \mid var \\
& \quad\ \mid digits\}
\end{aligned}
$$

Beim Bottom–Up Parsen dürfen Grammatiken linksrekursiv sein.

Grammatik im Happy–File

```
%% -- aus yacc-Tradition

expr              : Let Var Assign expr In expr { }
                  | expr Plus expr { }
                  | Var { }
                  | Int { }
```

Haskell-Code

Am Ende der Datei gibt es einen Abschnitt, in dem Haskell-Code programmiert werden kann.

```
{

parseError :: [Token] -> a
parseError _ = error "Parse error"

parser :: String -> MiniFunkExpr
parser = expr . alexScanTokens

main = do
  s <- readFile "Pfad/fst.mfe"
  print (parser s)


}
```