4. Syntaktische Analyse

Eingabe: Symbolfolge

Aufgaben:

Ableitung bilden mit konkreter Grammatik (parsing)

Strukturbaum aufbauen mit abstrakter Grammatik

Fehlerbehandlung

Ausgabe: Strukturbaum

Übersetzermodul Parser (dt. Zerteiler):

Kellerautomat; hier nur deterministische

kann bei Anwenden einer Produktion Aktion ausführen

(Baumknoten erzeugen)

spezifiziert durch konkrete KFG mit Aktionen

Zielbezogene Parser:

Linksableitung; Baum top-down oder bottom-up

Quellbezogene Parser:

Rechtsableitung rückwärts; Baum bottom-up

Strukturbaum (vergröberter Ableitungsbaum):

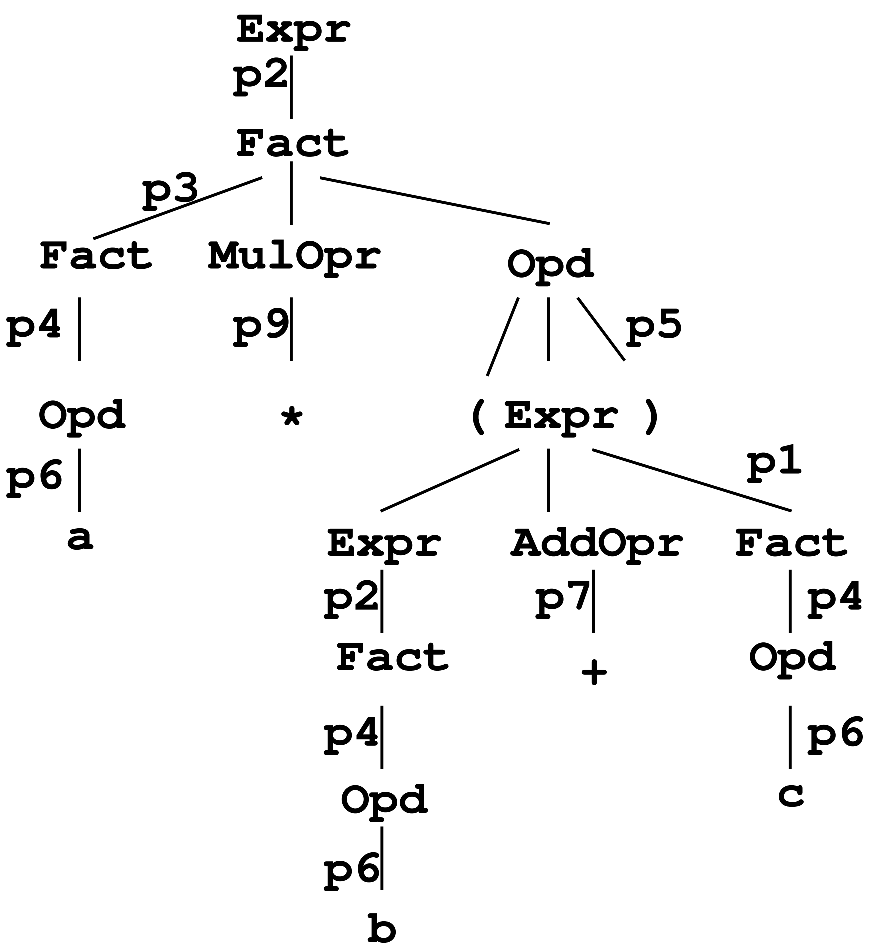
als Datenstruktur für Translationsphase, oder

implizit, Aufrufe von Operationen der Translationsphase

Beispiel: Ausdrucksgrammatik

Name	Produktion			Aktion
p1:	Expr	::=	Expr AddOpr Fact	BinEx
p2:	Expr	::=	Fact	
p3:	Fact	::=	Fact MulOpr Opd	BinEx
p4:	Fact	::=	Opd	
p5:	Opd	::=	'(' Expr ')'	
p6:	Opd	::=	Ident	IdEx
p7:	AddOpr	::=	'+'	PlusOpr
p8:	AddOpr	::=	'-'	MinusOpr
p9:	MulOpr	::=	'*'	TimesOpr
p10:	MulOpr	::=	'/'	DivOpr

Ableitungsbaum für `a * (b + c)`



Konkrete und abstrakte Syntax

konkrete Syntax	**abstrakte Syntax**
KFG	KFG
definiert Struktur der Quellprogramme	definiert Strukturbäume
eindeutig	i.a. mehrdeutig
spezifiziert Parser, Ableitung	Grundlage für Translationsphase
Aktionen spezifizieren	Baumaufbau
Kettenproduktionen `Expr ::= Fact`	unnötige weglassen
haben keine Aktion	
beteiligte Symbole	zu Symbolklasse vereinigt `Exp = {Expr, Fact}`

gleiche Aktion zu strukturgleichen Produktionen:

```
Expr ::= Expr AddOpr Fact &BinEx
Fact ::= Fact MulOpr Opd &BinEx
```

Terminale	unnötige weglassen, keine Baumknoten

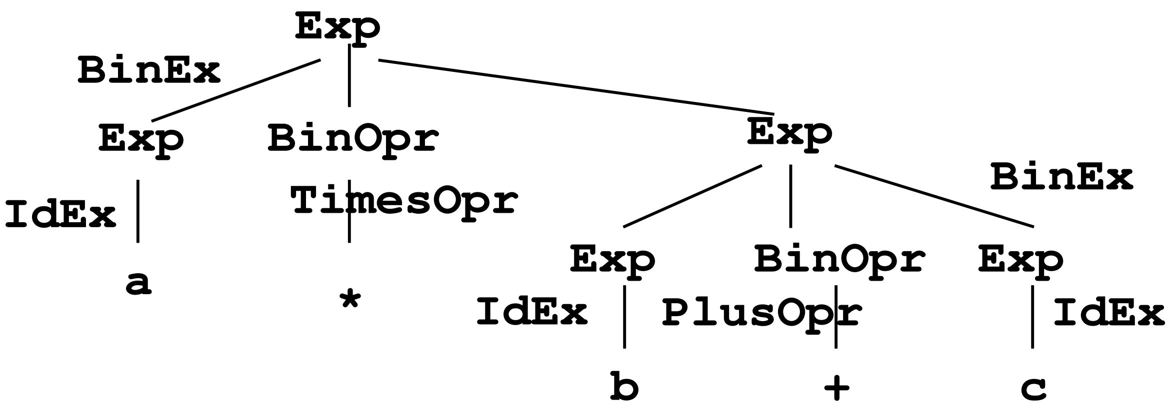
Aus konkreter Syntax und Symbolklassen sind die abstrakte Syntax und die Aktionen automatisch generierbar.

Beispiel: abstrakte Ausdrucksgrammatik

Name   Produktion

```
BinEx:     Exp    ::= Exp BinOpr Exp
IdEx:      Exp    ::= Ident
PlusOpr:   BinOpr ::= '+'
MinusOpr:  BinOpr ::= '-'
TimesOpr:  BinOpr ::= '*'
DivOpr:    BinOpr ::= '/'
```

Strukturbaum für `a * (b + c)`



Symbolklassen:

```
Exp = { Expr, Fact, Opd }
BinOpr = { AddOpr, MulOpr }
```

Aktionen der konkreten Syntax
Produktionen der abstrakten Syntax

keine Aktion an konkreter Produktion
kein Knoten im Strukturbaum

Recursive Descent Parser

dt.: rekursiver Abstieg; zielorientierte, prädiktive Methode

KFG in Menge von Funktionen transformieren:

Nichtterminal	Funktion
alternative Produktionen	Zweige im Rumpf
Nichtterminal auf rechter Seite	Aufruf
Terminal auf rechter Seite	akzeptiere Symbol und lies
Entscheidungsmenge zu Produktion	Verzweigungsentscheidung

Beispiel:

```
p1:  Stmt ::= Variable ':=' Expr
p2:       |   'while' Expr 'do' Stmt

void Stmt ()
{  switch (CurrSymbol)
   { case Entscheidungsmenge für p1:
           Variable ();
           accept (assignSym);
           Expr ();
           break;
     case Entscheidungsmenge für p2:
           accept (whileSym);
           Expr ();
           accept (doSym);
           Stmt ();
           break;
     default: Fehlerbehandlung;
}  }
```

Grammatikbedingung für Recursive Descent

Eine Grammatik ist **stark LL(1)**, wenn für alle Produktionenpaare mit gleicher linker Seite die Entscheidungmengen disjunkt sind:

	Entscheidungmengen
A ::= u	Anf (u Folge (A))
A ::= v	Anf (v Folge (A))

Anfangs- und Folgemengen:

Anf (u) := $\{t \in T \mid$ es gibt $u \Rightarrow^* tv\} \quad \cup \{\varepsilon\}$ falls $u \Rightarrow^* \varepsilon$ existiert

Folge (A) := $\{t \in T \mid$ es gibt $S \Rightarrow^* uAv$ und $t \in$ Anf (v)$\}$

Konsequenz:

Starke LL(1)-Grammatiken haben keine links-rekursiven Produktionen, und keine Produktionen mit gleichen Anfängen.

EBNF-Konstrukte, z.B.:

A ::= u (v)* w

in Funktion für A:

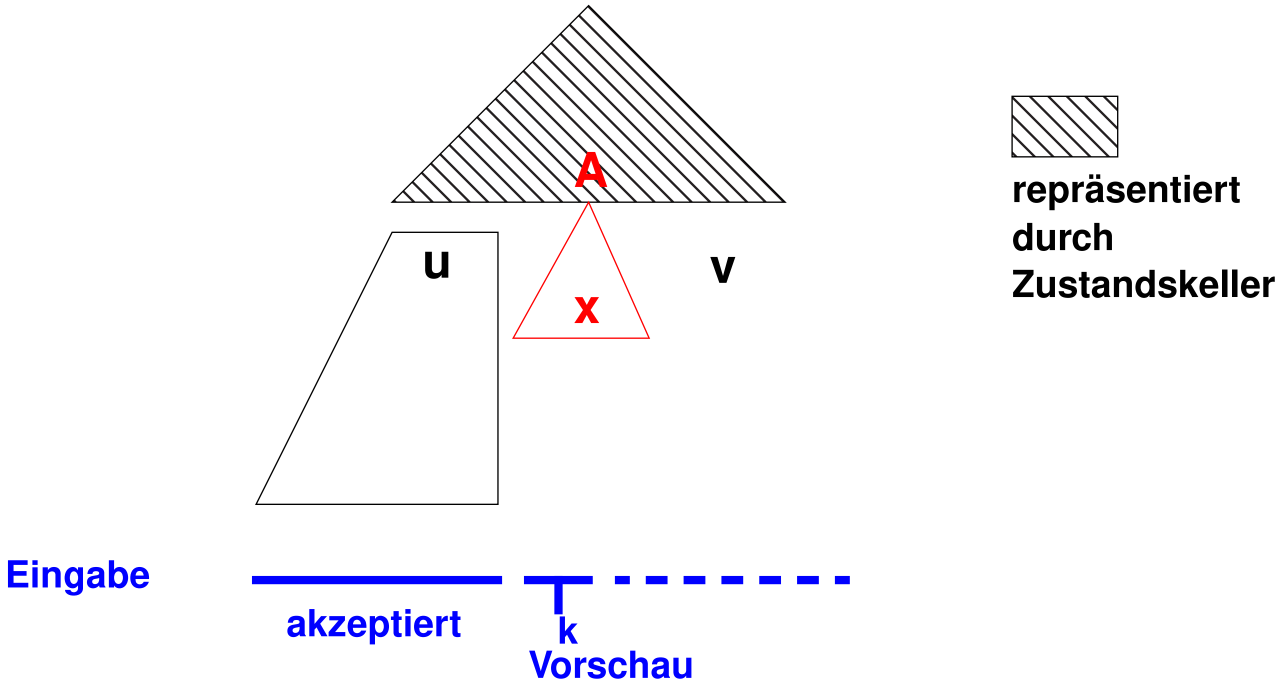
... u ... while (CurrSymbol in Anf (v)) { ... v ...} ...w ...

Es muß gelten: Anf (v) und Anf (w Folge (A)) sind disjunkt.

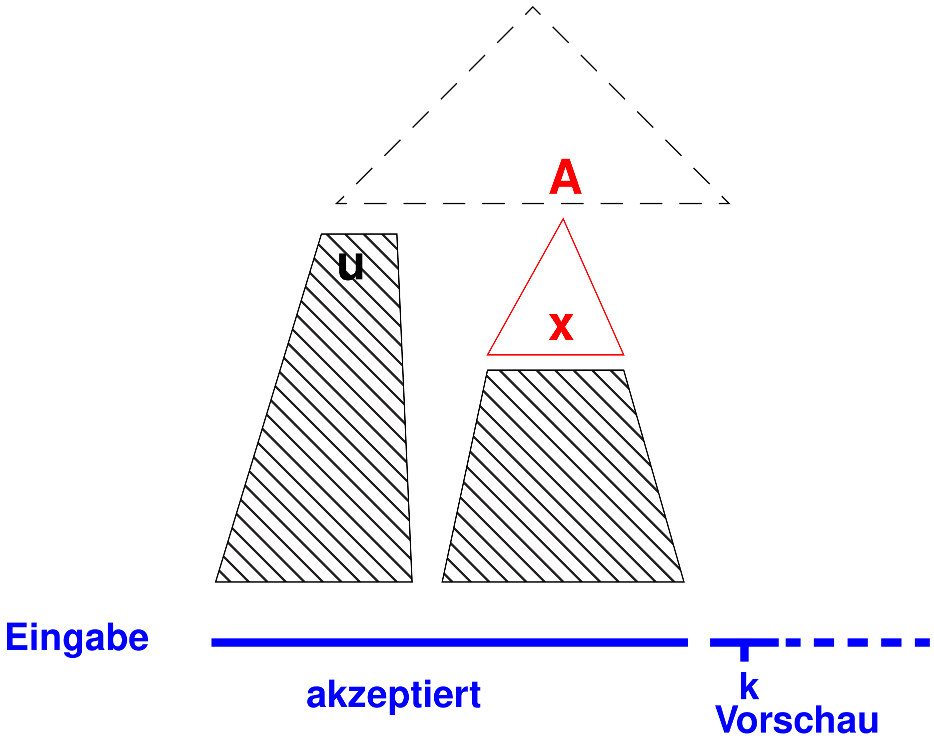
Vergleich: quellbezogen - zielbezogen

Information des Kellerautomaten bei der Entscheidung, die Produktion `A ::= x` anzuwenden.

zielbezogen, prädiktiv, Linksableitung:



quellbezogen, Rechtsableitung rückwärts:



LR(k)-Bedingung

Eine kontext-freie Grammatik ist eine LR(k)-Grammatik mit

$k \geq 0$, wenn für alle Paare von Rechtsableitungen

$$Z \Rightarrow^* u\ A\ w \Rightarrow u\ x\ w \qquad u, u', x, x' \in V^*$$

$$Z \Rightarrow^* u'\ B\ w' \Rightarrow u'\ x'\ w' \qquad w, w', v \in T^*$$

gilt:

falls es ein v gibt, so daß

$$u\ x\ v = u'\ x'\ w'$$

und die ersten k Zeichen von w und v gleich sind,

dann sind auch A = B, x = x', u = u'

(d. h. der letzte Ableitungsschritt ist gleich)

Diese Bedingung ist nicht konstruktiv und so i.a. nicht prüfbar!

Automat konstruieren, deterministisch g.d.w. LR(k).

Mächtigkeit der Grammatikklasse:

LL(k) => LR(k) => eindeutig

Praktische Bedeutung haben Unterklassen wie LALR(1).

LR(1)-Automat

Zustände werden gekellert.

Operationen: **shift** lesen und Zustand kellern

reduce mit Produktion reduzieren, Zustände entkellern

Ein **Zustand** repräsentiert eine **Menge von Situationen**.

Eine **Situation** repräsentiert den Analysestand bzgl. einer Produktion:

[A ::= u . v R]

Analyseposition **.**

R erwarteter **Rechtskontext**
Menge von Terminalen, die folgen können, wenn mit der Produktion reduziert ist.

Reduktionssituation:

[A ::= u v . R]

Automat reduziert, falls nächstes Zeichen der Eingabe in R