

Syntaxanalyse: LR-Parser (LR(0), LALR)

BC George (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Wiederholung

Top-Down-Analyse

- Baumaufbau von oben nach unten
- eine Möglichkeit: recursive-descent parser
- alternativ: tabellengesteuerter Parser
- *First*- und *Follow*-Mengen bestimmen Wahl der Ableitungen
- nicht mehr rekursiv, sondern mit PDA

$$S \rightarrow A \mid B \mid C \mid D$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$C \rightarrow c \mid \epsilon$$

$$D \rightarrow d \in \text{Follow}(C)$$

Motivation

LL ist nicht alles

Die Menge der *LL*-Sprachen ist eine echte Teilmenge der deterministisch kontextfreien Sprachen.

Bei *LL*-Sprachen muss man nach den ersten k Eingabezeichen entscheiden, welche Ableitung ganz oben im Baum als erste durchgeführt wird, also eine, die im Baum ganz weit weg ist von den Terminalen, die die Entscheidung bestimmen. Das ist nicht bei allen deterministisch parsebaren Grammatiken möglich und erschwert die Fehlerbehandlung.

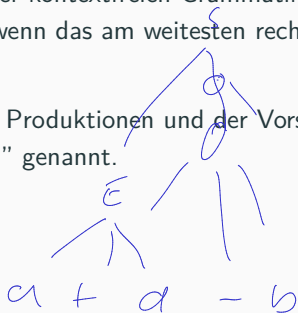
Von unten nach oben

Von unten nach oben

Bei der Bottom-Up-Analyse wird der Parse Tree von unten nach oben aufgebaut, von links nach rechts. Dabei entsteht eine *Rechtsableitung*.

Def.: Bei einer kontextfreien Grammatik G ist die *Rechtsableitung* von $\alpha \in (N \cup T)^*$ die Ableitung, die man erhält, wenn das am weitesten rechts stehende Nichtterminal in α abgeleitet wird. Man schreibt $\alpha \xRightarrow{*}_r \beta$.

Mit Hilfe der Produktionen und der Vorschautoken werden die Ableitungen “rückwärts” angewandt und “Reduktionen” genannt.



Versuchen wir es einmal

$$S' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Terminale
↑

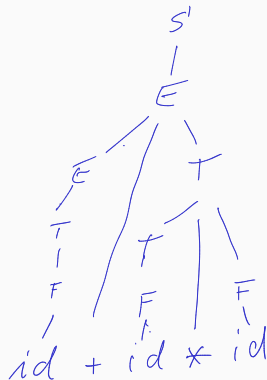
zu parsen $id + id * id$



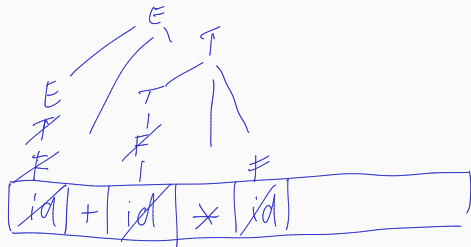
Strategie: reduzieren, wenn möglich
nicht gut

richtig wäre:

reduzieren
+ shiften
(Eingabe
mitlesen)



2) Stack benutzen:

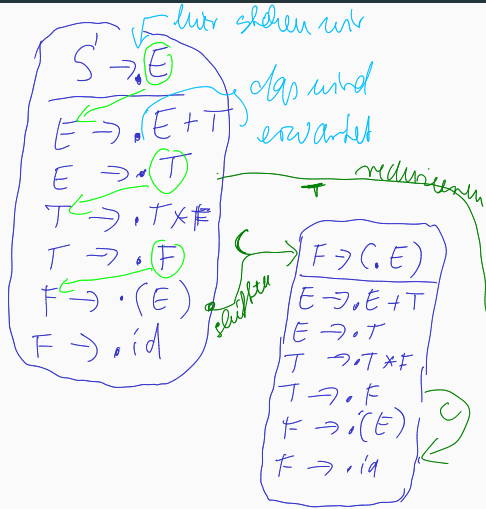


offene Unterbäume mit Stack
halten lässt auch nicht
(umgekehrt zu LL)

Kann ein Stack helfen?

≤ 0

So geht es vielleicht



$S' \rightarrow E$
 $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

Zustände
eines PDA
(eigentlich aber
 DFA)
bilden

o o o (insgesamt 12)

Da wollen wir hin

Werte setzen das zu reduzierenden Regel, Reduzieren und in den normalen Zustand gehen

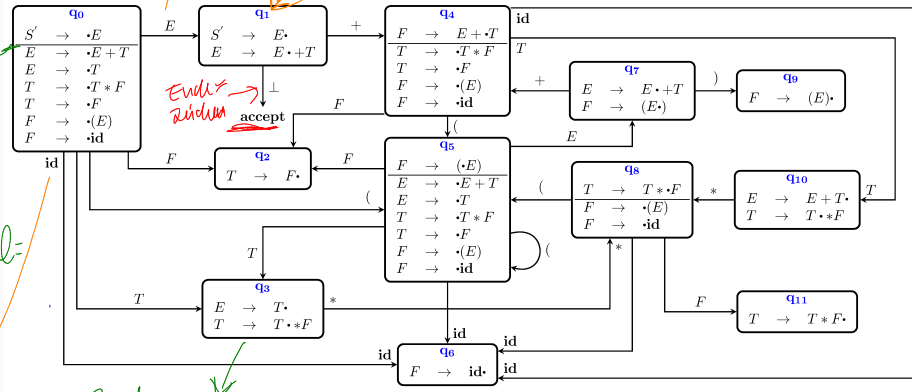


Abbildung 1: Parser-Automat

$$LR(0)$$

Zustände mit dem
Print hinter
T suchen

↑ Sprunghaken

24
schiffende
Terminale

close
ab hier
Füllbereich
lang

Theorie: LR(0)

Im Stack stehen nur Zustandsnummern, am Anfang die Nummer des Startzustandes (+ Bottomzeichen, oft auch \$).

- Lesen des obersten Stackelements ergibt Zustand q
- Lesen des nächsten Eingabezeichens ergibt Zeichen a
- Nachschlagen der Reaktion auf (q, a) in der Parse Table
- Durchführung der Reaktion

Mögliche "Actions" ohne Berücksichtigung von Vorschautoken

- Shift: Schiebe logisch das nächste Eingabesymbol auf den Stack (in Wirklichkeit Zustandsnummern)
- Reduce: (Identifiziere ein Handle oben auf dem Stack und ersetze es durch das Nichtterminal der dazugehörigen Produktion.) Das ist gleichbedeutend mit: Entferne so viele Zustände vom Stack wie die rechte Seite der zu reduzierenden Regel Elemente hat, und schreibe den Zustand, der im Goto-Teil für (q, a) steht, auf den Stack.
- Accept: Beende das Parsen erfolgreich
- Reagiere auf einen Syntaxfehler

$$E + T \Rightarrow E + T$$

nur Zustand
beim Redu-
zieren

Handle: rechte Seite von anzuwendenden Regeln

$E + T$ ist ein Handle der Produktion von E
($E \rightarrow E + T$)

Berechnung der Zustände: Items

Zustände

Def.: Ein (*dotted*) *Item* einer Grammatik G ist eine Produktion von G mit einem Punkt auf der rechten Seite der Regel vor, zwischen oder nach den Elementen.

Bsp.:

Zu der Produktion $A \rightarrow BC$ gehören die Items:

$[A \rightarrow \cdot BC]$

$[A \rightarrow B \cdot C]$

$[A \rightarrow BC \cdot]$

Das zu $A \rightarrow \epsilon$ gehörende Item ist $[A \rightarrow \cdot]$

Berechnung der $Closure_0$ von einer Menge I von Items

$$\begin{array}{c} A \rightarrow \cdot BC \\ B \rightarrow \cdot \gamma \end{array}$$

1. füge I zu $CLOSURE_0(I)$ hinzu
2. gibt es ein Item $[A \rightarrow \alpha \cdot B\beta]$ aus $CLOSURE_0(I)$ und eine Produktion $(B \rightarrow \gamma)$, füge $[B \rightarrow \cdot \gamma]$ zu $CLOSURE_0(I)$ hinzu

Berechnung der $GOTO_0$ -Sprungmarken

$$GOTO_0(I, X) = CLOSURE_0(\{[A \rightarrow \alpha X \cdot \beta] \mid [A \rightarrow \alpha \cdot X \beta] \in I\})$$

für eine Itemmenge I und $X \in N \cup T, A \in N, \alpha, \beta \in (N \cup T)^*$.

Ergebnis z.B.

$E \rightarrow T_e$
$T \rightarrow T_o \times F$

closure davon

dotted item

Konstruktion des $LR(0)$ - Automaten

1. Bilde die Hülle von $S' \rightarrow S$ und mache sie zum ersten Zustand.
2. Für jedes noch nicht betrachtete $\cdot X, X \in (N \cup T)$ in einem Zustand q des Automaten berechne $GOTO_0(q, X)$ und mache $GOTO_0(q, X)$ zu einem neuen Zustand r . Verbinde q mit einem Pfeil mit r und schreibe X an den Pfeil. Ist ein zu r identischer Zustand schon vorhanden, wird p mit diesem verbunden und kein neuer erzeugt.

Konstruktion der Parse Table

1. Erstelle eine leere Tabelle mit den Zuständen als Zeilenüberschriften. Für den Aktionstabellenteil überschreibe die Spalten mit den Terminalen, für den Sprungtabellenteil mit den Nonterminals.
2. Shift: Für jeden mit einem Terminal beschrifteten Pfeil aus einem Zustand erstelle in der Aktionstabelle die Aktion shift mit der Nummer des Zustands, auf den der Pfeil zeigt. Für Pfeile mit Nonterminals schreibe in die Sprungtabelle nur die Nummer des Folgezustands.
3. Schreibe beim Zustand $[S' \rightarrow S \cdot]$ ein *accept* bei dem Symbol \perp .
4. Für jedes Item mit $[A \rightarrow \beta \cdot]$ aus allen Zuständen schreibe für alle Terminals *reduce* und die Nummer der entsprechenden Grammatikregel in die Tabelle.

Ein Beispiel zum Nachvollziehen

$$(0) S' \rightarrow S$$

$$(1) S \rightarrow aAbScS$$

$$(2) S \rightarrow aAbS$$

$$(3) S \rightarrow d$$

$$(4) A \rightarrow e$$

Der LR(0)-Automat zu G1

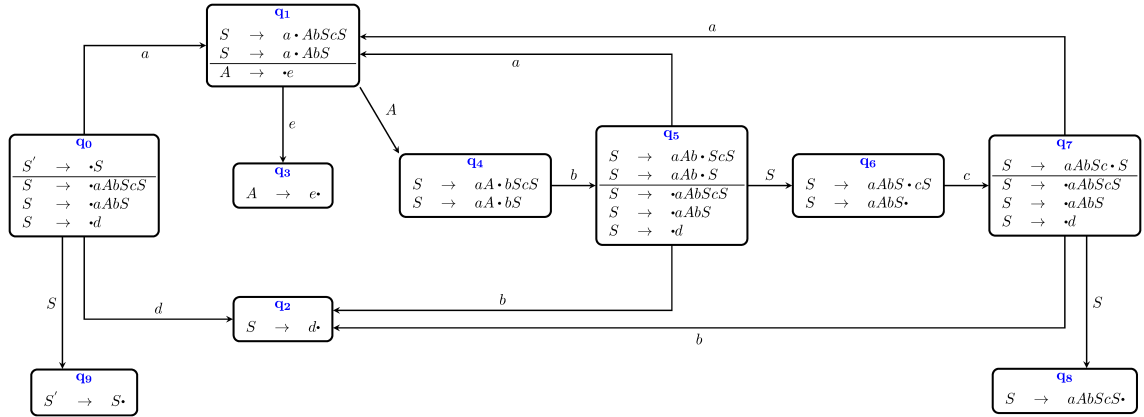


Abbildung 2: LR(0)-Automat

Die LR(0)-Parsertabelle zu G1

Zustand	Aktion						Sprung	
	a	b	c	d	e	\perp	S	A
q_0	s1			s2			q_9	
q_1					s3			q_4
q_2	r3	r3	r3	r3	r3	r3		
q_3	r4	r4	r4	r4	r4	r4		
q_4		s5						
q_5	s1	s2		s2			q_6	
q_6	r2	r2	s7,r2	r2	r2	r2		
q_7	s1	s2					q_8	
q_8	r1	r1	r1	r1	r1	r1		
q_9						acc		

r: reduce
+ displacement

S: shift +
Zustandsnr

Abbildung 3: LR(0)-Parsertabelle

shift-reduce
Konflikt

Es gibt auch noch reduce/reduce-Konflikt

Und was gibt es noch?

Wenn LR(0) nicht reicht

Zunächst: Zu jeder LR(k)-Sprache gibt es eine LR(1)-Grammatik.

Ist eine Grammatik nicht LR(0), müssen nichtdeterministische Tabelleneinträge verhindert werden:

- ^{simple} SLR(1)-Parsing ($A \rightarrow \beta$ wird nur reduziert, wenn das Vorschautoken in der *FOLLOW*-Menge von A ist.)
- (kanonisches) LR(1)-Parsing (wie LR(0) mit einem Vorschautoken) *normales LR(1)*
- LALR(1)-Parsing (Zusammenfassung aller LR(1)-Zustände, die sich nur in den LOOKAHEAD-Mengen unterscheiden)
(Vorschau)

Mehrdeutige Grammatiken

Es gibt auch Auswege

Mehrdeutige Grammatiken sind oft leichter zu lesen und kleiner als die Grammatiken, die man erhält, wenn man die Mehrdeutigkeit auflöst, sofern möglich.

Folgendes kann bei Mehrdeutigkeiten helfen:

- Angabe von Vorrangregeln
- Angabe von Assoziativität
- Voreinstellung des Parsergenerators: z. B. Shiften bei Shift-Reduce-Konflikten
- Voreinstellung des Parsergenerators: z. B. Reduzieren nach der Regel, die in der Grammatik zuerst kommt bei Reduce-Reduce-Konflikten

/ das ist das Problem

Hierarchie der kontextfreien Sprachen

Hierarchie der kontextfreien Sprachen

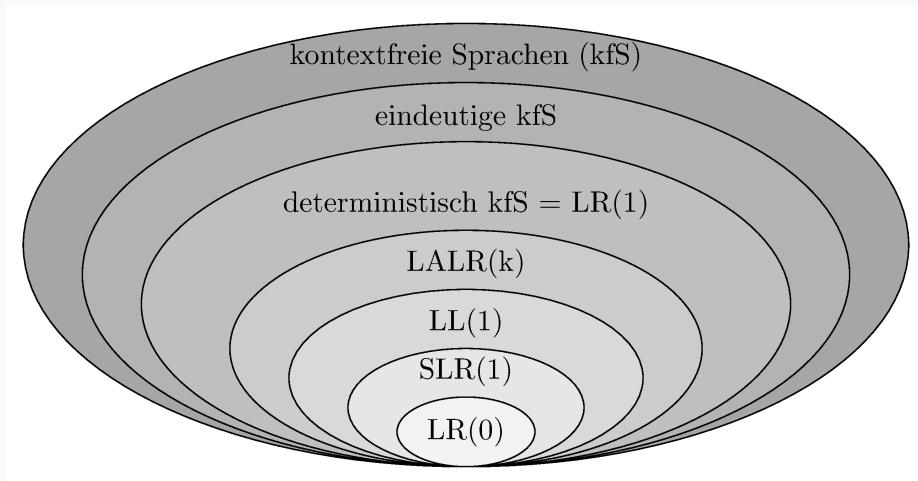


Abbildung 4: Sprachenhierarchie

Wrap-Up

- LR-Analyse baut den Ableitungbaum von unten nach oben auf
- es wird ein DFA benutzt zusammen mit einem Stack, der Zustände speichert
- eine Parse-Tabelle steuert über Aktions- und Sprungbefehle das Verhalten des Parsers
- die Tabelle wird mit (dotted) Items und Closures konstruiert
- mit Bottom-Up-Parsing LR(1) kann man alle deterministisch kontextfreien Sprachen parsen
- LR(0)-, SLR- und LALR- Parsing sind vereinfachte Verfahren für Teilmengen der LR-Sprachen



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.