GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

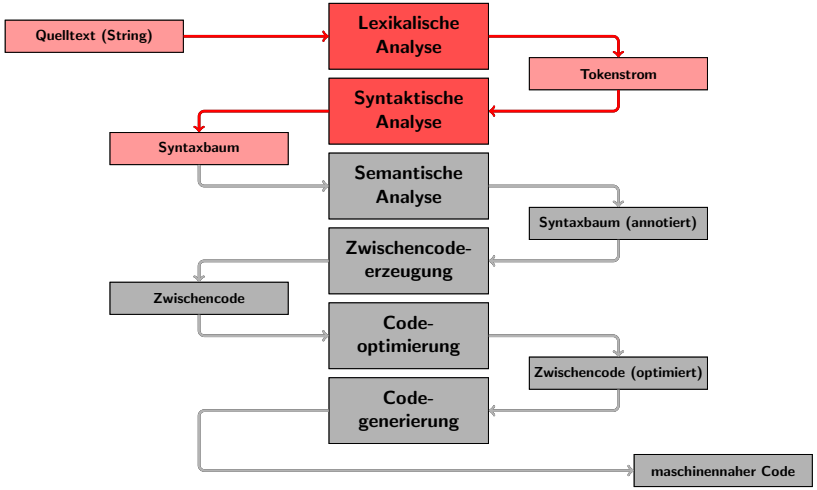
Grundlagen der Programmierung 2

Schiebe-Reduziere-Parser

Prof Dr. Manfred Schmidt-Schauß
(unter Mithilfe von PD Dr. David Sabel)

Sommersemester 2017

Phasen eines Compilers



Programm für heute

Bereits behandelt:

- Rekursiv-absteigende Parser
- Rekursive-prädiktive Parser
- Beide sind Top-Down Parser!

Heute: Shift-Reduce-Parser (Schiebe-Reduziere)

- Bottom-Up Parser
- Shift-Reduce Parser für **Operatorgrammatiken**
- SLR(1)-Parser (spezielle Shift-Reduce Parser)
- **Parsergeneratoren** für SLR(1)-Parser:
  Happy für Haskell

LR-Parser, Shift-Reduce-Verfahren

Bottom-Up-Syntaxanalyse

- **Bottom-Up**: Vom Wort zurück zum Startsymbol
  z.B. $(1+5) \leftarrow (1+E) \leftarrow (E+E) \leftarrow (E) \leftarrow E$

LR-Parser

- **L**: Die Eingabe wird von **links** nach rechts verarbeitet
- **R**: **Rechtsherleitung**
- $(1+5) \leftarrow (E+5) \leftarrow (E+E) \leftarrow (E) \leftarrow E$
  ist **eine** Rechtsherleitung!

Shift-Reduce-Verfahren

- Kein Backtracking
- Nicht für jede Grammatik anwendbar
- Schieben und Reduzieren: . . . gleich . . .

Beispiel: arithmetische Ausdrücke (Expressions)

Kontextfreie Grammatik:

$$\begin{array}{lcl} \mathbf{E} & ::= & \mathbf{E} + \mathbf{E} \\ \mathbf{E} & ::= & \mathbf{E} * \mathbf{E} \\ \mathbf{E} & ::= & (\mathbf{E}) \\ \mathbf{E} & ::= & \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \end{array}$$

Eine **Rechtsherleitung** von `c + b * a`

$$\begin{array}{lcl} \mathbf{E} & \rightarrow & \mathbf{E} + \mathbf{E} \\ & \rightarrow & \mathbf{E} + \mathbf{E} * \mathbf{E} \\ & \rightarrow & \mathbf{E} + \mathbf{E} * \mathsf{a} \\ & \rightarrow & \mathbf{E} + \mathsf{b} * \mathsf{a} \\ & \rightarrow & \mathsf{c} + \mathsf{b} * \mathsf{a} \end{array}$$

Vorsicht: es gibt eine weitere Rechtsherleitung

Begriff: Handle

Handle v in Wort w:

- rechte Seite v einer Regel $A ::= v$

Genauer bzw. verfeinert:

- Vorkommen von v in w, so dass man rückwärts Rechtsherleiten kann:
  $w = uvu' \leftarrow uAu'$ (u' enthält keine Nichtterminale!)
- genauer: v ist Handle im Wort w, wenn es eine Rechtsherleitung gibt: $\sigma \stackrel{*}{\rightarrow} uAu' \rightarrow uvu' = w$

Beispiel: $\mathbf{E} ::= \mathsf{a} \mid \mathbf{E} + \mathbf{E} \mid (\mathbf{E} + \mathbf{E})$

- $\mathbf{E} + \mathbf{E} + \mathbf{E}$ ist kein Handle: $\underline{\mathbf{E}} + \mathbf{E} \rightarrow \underline{\mathbf{E} + \mathbf{E}} + \mathbf{E}$ ist kein Rechtsherleitungsschritt $\mathbf{E} + \mathbf{E} + \mathbf{E}$ ist ein Handle: $\mathbf{E} + \underline{\mathbf{E}} \rightarrow \mathbf{E} + \underline{\mathbf{E} + \mathbf{E}}$ ist ein Rechtsherleitungsschritt ($\mathbf{E} \stackrel{*}{\rightarrow} \mathbf{E} + \mathbf{E}$)
- $(\mathbf{E} + \mathbf{E}) + (\mathbf{E} + \mathbf{E})$ kein Handle kein Handle:
  $\mathbf{E} + \mathbf{E} + (\mathbf{E}) \rightarrow \mathbf{E} + \mathbf{E} + (\mathbf{E} + \mathbf{E})$, aber $\mathbf{E} \stackrel{*}{\not\rightarrow} \mathbf{E} + \mathbf{E} + (\mathbf{E})$ ist ein Handle

Beispiel

$$\begin{array}{llcl} \mathsf{R1} & \mathbf{E} & ::= & \mathbf{E} + \mathbf{E} \\ \mathsf{R2} & \mathbf{E} & ::= & \mathbf{E} * \mathbf{E} \\ \mathsf{R3} & \mathbf{E} & ::= & (\mathbf{E}) \\ \mathsf{R4} & \mathbf{E} & ::= & \mathsf{a} \mid \mathsf{b} \mid \mathsf{c} \end{array}$$

(Eine) Rechtsherleitung von `c + b * a` mit Handles

$$\begin{array}{lcl} \mathbf{E} & \stackrel{R1}{\rightarrow} & \mathbf{E} + \mathbf{E} \\ & \stackrel{R2}{\rightarrow} & \mathbf{E} + \mathbf{E} * \mathbf{E} \\ & \stackrel{R4.a}{\rightarrow} & \mathbf{E} + \mathbf{E} * \mathsf{a} \\ & \stackrel{R4.b}{\rightarrow} & \mathbf{E} + \mathsf{b} * \mathsf{a} \\ & \stackrel{R4.c}{\rightarrow} & \mathsf{c} + \mathsf{b} * \mathsf{a} \end{array}$$

Handle: Benutzung

- Die Handles werden zur Konstruktion einer Rechtsherleitung in umgekehrter Reihenfolge (rückwärts) benutzt
- Handles sind jedoch nicht immer eindeutig

Fortsetzung des Beispiels

Konstruiere Rechtsherleitung von $c + b * a$
durch Ersetzen von Handles

Rechtsherleitung ↑

hergeleitetes Wort	Handle	Produktion
$c + b * a$	c	$\mathbf{E} \to c$
$\mathbf{E} + b * a$	b	$\mathbf{E} \to b$
$\mathbf{E} + \mathbf{E} * a$	a	$\mathbf{E} \to a$
$\mathbf{E} + \mathbf{E} * \mathbf{E}$	$\mathbf{E} * \mathbf{E}$	$\mathbf{E} \to \mathbf{E} * \mathbf{E}$
$\mathbf{E} + \mathbf{E}$	$\mathbf{E} + \mathbf{E}$	$\mathbf{E} \to \mathbf{E} + \mathbf{E}$
$\mathbf{E}$		

Methode: Schieben-Reduzieren (1)

Shift-Reduce Syntax-Analysemethode
Zustand während der Analyse:

- Rest der Eingabe
- Aktuelles Zeichen
- Stack (das bisher rückwärts hergeleitete Wort)

Stack + aktuelles Zeichen + Rest der Eingabe ≡ Gesamtwort

Methode: Schieben-Reduzieren (2)

Wesentliche Aktionen der Analyse:

Schieben: Lesen eines Zeichens der Eingabe und Ablage auf den Stack

Reduzieren: Ersetzen eines obersten Teils des Stacks, das Handle, mittels einer Produktion rückwärts

Die Auswahl der Produktion ist abhängig vom Anfang der Resteingabe und vom Inhalts des Stacks.

Beispiel

$ bezeichnet das Ende der Eingabe und den Boden des Stacks

Stack	Eingabe	Aktion
$\$$	$c + b * a\$$	schiebe
$\$c$	$+ b * a\$$	reduziere mit $\mathbf{E} ::= c$
$\$\mathbf{E}$	$+ b * a\$$	schiebe
$\$\mathbf{E} +$	$b * a\$$	schiebe
$\$\mathbf{E} + b$	$* a\$$	reduziere mit $\mathbf{E} ::= b$
$\$\mathbf{E} + \mathbf{E}$	$* a\$$	schiebe
$\$\mathbf{E} + \mathbf{E} *$	$a\$$	schiebe
$\$\mathbf{E} + \mathbf{E} * a$	$\$$	reduziere mit $\mathbf{E} ::= a$
$\$\mathbf{E} + \mathbf{E} * \mathbf{E}$	$\$$	reduziere mit $\mathbf{E} ::= \mathbf{E} * \mathbf{E}$
$\$\mathbf{E} + \mathbf{E}$	$\$$	reduziere mit $\mathbf{E} ::= \mathbf{E} + \mathbf{E}$
$\$\mathbf{E}$	$\$$	akzeptiere

Die 4 Parser-Aktionen:

Schieben:	Zeichen von Eingabe auf Stack
Reduzieren:	Handle, d.h. ein oberes Stück des Stacks durch ein Nichtterminal ersetzen.
Akzeptieren:	Wenn Stack = Startsymbol und die Eingabe leer
Fehlererkennung:	Wenn weder Schiebe- noch Reduzieraktion möglich.

Shift-Reduce-Parser: Eigenschaften

Shift-Reduce-Parser sind deterministisch:
Berechnung der nächsten Aktion auf der Basis
des Stackinhalts
und des ersten Symbols der Eingabe

- Shift-Reduce-Parser arbeiten i.a. tabellengesteuert
  Normalerweise wird diese Tabelle automatisch erzeugt
  D.h. Parser als eine Stack-Maschine
- Die manuelle Erzeugung der Tabellen aus der Grammatiken ist i.a. zu komplex
- ε-Produktionen werden in Shift-Reduce-Parsern vermieden.

Später: Autom. Erzeugen von SR-Parsern mit Parsergeneratoren

SR-Parser-Generatoren: Fehlerbehandlung

- Schiebe-Reduziere-Konflikt:
  unklarer Zustand: Schieben oder Reduzieren?
  Konflikt muss beim Generieren aufgelöst werden durch Festlegung
- Reduziere-Reduziere-Konflikt:
  unklarer Zustand: Reduzieren mit welcher Regel?
  2 verschiedene Handles, oder ein Handle und 2 Regeln.
  Konflikt muss beim Generieren aufgelöst werden durch Festlegung
- Keine anwendbare Regel
  ergibt Fehlermeldung beim Parsen
  Besser: explizite Fehlermeldung vom Parsergenerator schon vorgesehen.

Shift-Reduce Konflikt: Beispiel (1)

Beispiel: Sprache mit `if-then` und `if-then-else`

$$\begin{aligned}\mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\ \texttt{else}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{True}\end{aligned}$$

Stack	Eingabe	Aktion
$	if True then if True then True else True$	schiebe
$if	True then if True then True else True$	schiebe
$if True	then if True then True else True$	reduziere mit $\mathbf{E} ::= \texttt{True}$
$if E	then if True then True else True$	schiebe
$if E then	if True then True else True$	schiebe
$if E then if	True then True else True$	schiebe
$if E then if True	then True else True$	reduziere mit $\mathbf{E} ::= \texttt{True}$
$if E then if E	then True else True$	schiebe
$if E then if E then	True else True$	schiebe
$if E then if E then True	else True$	reduziere mit $\mathbf{E} ::= \texttt{True}$
$if E then if E then E	else True$	

Zwei Möglichkeiten: Schiebe oder Reduziere mit $\mathbf{E} ::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}$

Shift-Reduce Konflikt: Beispiel (2)

Behandlung von Shift-Reduce-Konflikten,
Möglichkeiten:

- Man legt fest, ob geschoben oder reduziert werden soll
- Der Parser bricht mit einem Fehler ab
- Man legt eine Priorität der Aktion fest, z.B. aufgrund des aktuellen Zustands und des Lookahead-Symbols.
- Man macht „irgendwas“

Beste Möglichkeit: Grammatik verändern bzw. andere Vorkehrungen treffen, damit die Konflikte nicht auftreten.

Shift-Reduce Konflikt: Beispiel (3)

Zurück zum Beispiel, Annahme: Schiebe statt zu Reduzieren

$$\begin{aligned}\mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\ \texttt{else}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{True}\end{aligned}$$

Stack	Eingabe	Aktion
...	...	
$if E then if E then E	else True$	schiebe
$if E then if E then E else	True$	schiebe
$if E then if E then E else E	$	reduziere mit E ::= True
$if E then if E then E else E	$	reduziere mit E ::= if E then E else E
$if E then E	$	reduziere mit E ::= if E then E
$E	$	akzeptiere

Shift-Reduce Konflikt: Beispiel (4)

Andere Variante: Reduziere statt zu Schieben

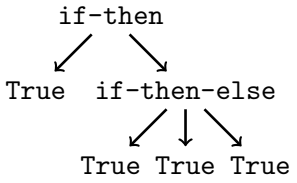
$$\begin{aligned}\mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\ \texttt{else}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{True}\end{aligned}$$

Stack	Eingabe	Aktion
...	...	
$if E then if E then E	else True$	reduziere mit E ::= if E then E
$if E then E	else True$	schiebe
$if E then E else	True$	schiebe
$if E then E else True	$	reduziere mit E ::= True
$if E then E else E	$	reduziere mit E ::= if E then E else E
$E	$	akzeptiere

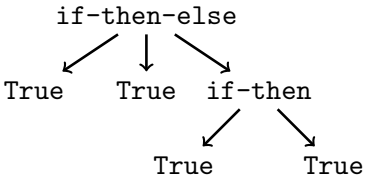
Shift-Reduce Konflikt: Beispiel (5)

Beachte: Die Syntaxbäume sind verschieden

Bei Schieben statt zu Reduzieren



Bei Reduzieren statt zu Schieben

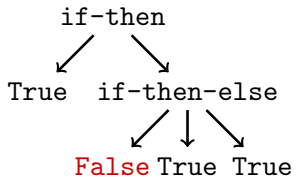


Shift-Reduce Konflikt: Beispiel (5)

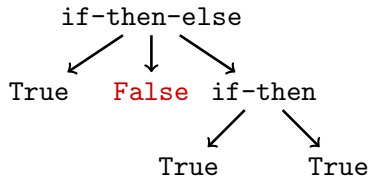
Beachte: Die Syntaxbäume sind verschieden

Bedeutung kann verschieden sein:

Bei Schieben statt zu Reduzieren



Bei Reduzieren statt zu Schieben



Reduce-Reduce Konfikt: Beispiel (1)

Mehrdeutige Grammatik:

$$\mathbf{A} \;::= \mathtt{a}\mathbf{A} \mid \mathtt{aa}\mathbf{A} \mid \mathtt{b}$$

Stack	Eingabe	Aktion
$	aab$	schiebe
$a	ab$	schiebe
$aa	b$	schiebe
$aab	$	reduziere mit $\mathbf{A} ::= \mathtt{b}$
aa\mathbf{A}$	$	???

Reduce-Reduce Konflikt!

reduziere mit $\mathbf{A} ::= \mathtt{a}\mathbf{A}$ oder

reduziere mit $\mathbf{A} ::= \mathtt{aa}\mathbf{A}$

Reduce-Reduce Konflikte

- Reduce-Reduce-Konflikte sind auf jeden Fall zu vermeiden
- Zeigen Mehrdeutigkeit der Grammatik
- Grammatik sollte geändert werden
- Oder Prioritäten der Aktionen einführen z.B. aufgrund des aktuellen Stacks und des Lookahead-Symbols

Operatorgrammatiken

Operatorgrammatiken

Operatorgrammatiken

Operatorgrammatiken erlauben die (einfache) Konstruktion eines SR-Parsers aus

- Basisgrammatik (darf auch Mehrdeutigkeiten enthalten) und
- weitere Angaben zu den Operatoren

Definition: Operatorgrammatik

Eine **Operatorgrammatik** liegt vor, wenn

- keine ε-Produktion existiert
- keine rechte Seite einer Produktion hat direkt benachbarte Nichtterminale
- Terminale sind Operatoren oder Klammern, oder Bezeichner (eigentlich ein Nichtterminal)
- wesentliches Nichtterminal: Ausdrücke

Operatorgrammatik: Beispiele (1)

$$\mathbf{E} ::= \mathbf{E} + \mathbf{E} \mid -\mathbf{E} \mid \mathbf{E} - \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid \mathbf{E}/\mathbf{E} \mid (\mathbf{E}) \mid \mathbf{E}\,\hat{}\,\mathbf{E} \mid \mathbf{E}! \mid \mathbf{Id}$$

wobei $\mathbf{Id}$ für einen Bezeichner steht (Nichtterminal) (Konstanten oder Variablen)

- Operatoren sind: `+,-,*,\,^,!`
- Klammern sind vorhanden
- **E**: Nichtterminal für Ausdrücke.

Beachte, dass diese Operatorgrammatik mehrdeutig ist:
`"1-2-3"` hat 2 Parsebäume

Operatorgrammatik: Beispiele (2)

$$\begin{aligned} \mathbf{E} &::= \mathbf{E}\ \mathbf{BOP}\ \mathbf{E} \mid \mathbf{Id} \\ \mathbf{BOP} &::= + \mid * \mid - \mid / \end{aligned}$$

ist **keine** Operatorgrammatik, denn die Produktion

$$\mathbf{E} ::= \mathbf{E}\ \mathbf{BOP}\ \mathbf{E}$$

hat **direkt benachbarte Nichtterminale** (ohne Operator dazwischen)

Operatorgrammatiken: Eindeutigkeit

Weitere Angaben, um Eindeutigkeit zu erreichen

- Stelligkeit: wieviele Argumente bindet der Operator?
  I.a.: 1 oder 2.
- Infix, Prefix, Postfix: welchen Ausdruck (welche Ausdrücke) bindet der Operator?
- Prioritäten welcher Operator bindet stärker? I.a. sind die Prioritäten natürliche Zahlen
- Assoziativität: Ist der Operator rechtsassoziativ oder linksassoziativ, oder nicht assoziativ?
  Z.B. $a + b + c$ kann als $a + (b + c)$ (rechtsassoziativ) oder als $(a + b) + c$ (linksassoziativ) geklammert werden.

Beispiel

Basisgrammatik:

$$\mathbf{E} ::= \mathbf{E} + \mathbf{E} \mid -\mathbf{E} \mid \mathbf{E} - \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid \mathbf{E}/\mathbf{E} \mid (\mathbf{E}) \mid \mathbf{E}\hat{}\mathbf{E} \mid \mathbf{E}! \mid \mathbf{Id}$$

wird eindeutiger durch die Eigenschaften:

- $+, *, /, \hat{}$ sind zweistellige Infixoperatoren.
- $-$ kann zweistelliger Infix- oder einstelliger Präfixoperator sein.
- ! ist einstelliger Postfixoperator
- Prioritäten: ! vor $\hat{}$ vor einstelligem $-$ vor $*, /$ vor $+, -$.
- $+, *, /, -$ linksassoziativ, $\hat{}$ ist rechtsassoziativ.

Daher „weiß" man:

$$\begin{array}{lll} 1 + 2!\hat{}3 & \text{entspricht} & 1 + ((2!)\hat{}3) \\ 1 - 3 - 5 * 6! & \text{entspricht} & ((1 - 3) - (5 * (6!)) \end{array}$$

Eigenschaften

Es gilt:

- Basisgrammatik + Eigenschaften der Operatoren ist in eine CFG kodierbar
- Aber Datengrundlage eines Shift-Reduce Parser ist die Basisgrammatik und Eigenschaften der Operatoren
- nicht die volle Grammatik.

Beispiel

Gegeben: ! Postfix-Operator mit hoher Priorität,
$-$ einstellige mit geringerer Priorität
$+$ binär mit geringster Priorität, linksassoziativ
Zahl sei Nichtterminal für Zahlenkonstanten.

Eindeutige Grammatik dazu:

E	::=	**PlusE**
FakE	::=	**Zahl** | **FakE** ! | (**PlusE**)
MinE	::=	**FakE** | - **MinE**
PlusE	::=	**MinE** | **PlusE** + **FakE**

- `- - 1 ! !` ist als `- (- ((1!)!))` erkennbar.
- Vermutlich ist diese Grammatik eindeutig.
- Diese Grammatik ist linksrekursiv (kein Problem)

Vorgehen:
Auf dem Stack sind in der Reihenfolge der Eingabe:

- die bisherigen Operatoren,
- die offenen Klammern und
- Bezeichner und erkannte Ausdrücke (d.h. Herleitungsbäume)

Shift-Reduce Syntaxanalyse: Aktionen

- Wenn ein Bezeichner in der Eingabe ist, dann Schieben manchmal danach reduzieren, Z.B., wenn $-$;E auf dem Stack
- Wenn Tokenstrom zu Ende, dann reduzieren.
- Wenn „(“, dann schieben
- Bei „)“: Wenn der Stack noch offene Operatoren enthält, dann reduzieren.
  Am Ende muss auf dem Stack „(“ ; E stehen:
  Der Klammerausdruck (E) wird erkannt,
  und der Syntaxbaum für E bleibt auf dem Stack.

Operatorgrammatiken: Beispiele

Stack	Eingabe-Rest	
$E +E	$	Reduzieren, da keine Argumente mehr folgen
$(E +E	)* 5 + 6 ! $	Reduzieren, da keine Argumente mehr folgen
$(E	)* 5 + 6 ! $	Schieben, da Klammerausdruck beendet
$E +	(3 + 5 ! $	Schieben
$E : E	: 5 : [] $	Schieben da : rechtsassoziativ

Operatorgrammatiken: Beispiele

Stack	Rest der Eingabe	
$E *E	+ 5 * 6 ! $	Reduzieren, da $* > +$
$E +E	* 5 + 6 ! $	Schieben, da $* > +$
$E + E	+ 5 * 6 ! $	Reduzieren, da + linksassoziativ
$E : E	: 5 : [] $	Schieben da : rechtsassoziativ

Operatorgrammatiken: Beispielablauf

$	1 - 3 - 5 * 6 ! $	S
$1	- 3 - 5 * 6 ! $	R
$E	- 3 - 5 * 6 ! $	S
$E -	3 - 5 * 6 ! $	S
$E - 3	- 5 * 6 ! $	R
$E - E	- 5 * 6 ! $	R
$E	- 5 * 6 ! $	S
$E -	5 * 6 ! $	S
$E - 5	* 6 ! $	R
$E - E	* 6 ! $	S
$E - E*	6 ! $	S
$E - E*6	! $	R
$E - E*E	! $	S
$E - E*E!	$	R
$E - E*E	$	R
$E - E	$	R
$E	$	

Erkannt wurde der Ausdruck: $((1-3)-(5*(6!)))$

Operatorgrammatiken: Bemerkungen

- Der Parser baut nicht direkt auf einer Grammatik auf.
  Definition der erkannten formalen Sprache?
- Fehler in der Eingabe sind gut erkennbar:
  Wenn es keine erlaubte Aktion mehr gibt
- Mehrdeutigkeiten werden durch die Implementierung des Parsers aufgelöst.
  Bei gleicher Priorität „gewinnt“ der frühere Operator.
- Zweideutigkeit von '$-$' (einstellig Präfix und zweistellig Infix) wird entschieden durch folgende Vereinbarung: wenn links von $-$ kein Operand, dann einstellig, sonst zweistellig.
- Operatoren, die links mehr als ein Argument konsumieren, sind i.a. nicht zugelassen.
- Die Semantik kann direkt aus dem Syntaxbaum abgelesen werden.

Operatorgrammatiken: Bemerkungen

- Vorteil: Die Operatoren und deren Eigenschaften können vom Benutzer definiert werden (siehe Haskell)
- Nachteil einer klammerfreien Eingabe:
  vom Benutzer gewollter und vom Parser erkannter Ausdruck können verschieden sein.
  auch unter strengem Typsystem usw.
  Abhilfe: wenn man unsicher ist, mehr Klammern setzen.
- Beim Testen muss man auch falsche Eingaben mit testen
- Unproblematische Erweiterung in Haskell :
  `(+)` , `(+ 1)` `(1 +)` sind ebenfalls Ausdrücke

SLR(1)-Parser

- SLR(1) = Simple LR(1)
- Shift-Reduce Parser, die mehr können als Operatorgrammatiken
- Konstruktion aufwändig
- aber Konstruktion automatisch möglich mit

Parsergeneratoren

SLR(1)-Parser: Komponenten

- endliche Menge von Zuständen (Zahlen), ein Startzustand
- Stack der Form $s_1 X_1 s_2 X_2 \ldots s_n X_n s_{n+1}$ wobei: X_i Terminale / Nichtterminale und s_i Zustände (i.A. Zahlen)
- Eingabestrom (von Token)
- zwei Steuerungstabellen (bzw. Funktionen):
  - *Aktionstabelle* $aktion(s_i, a_i)$ (bzw. auch $aktion(s_i, \$)$) für Zustand s_i und Eingabesymbol a_i (bzw. leere Eingabe)
    - Schiebe und gehe zu Zustand s_k
    - Reduziere mit Produktion
    - Akzeptiere
    - Fehler
  - Goto-Tabelle $goto(s_i, A)$ für Zustand s_i und Nichtterminal A: Nachfolgezustand (Funktion ist partiell)

SLR(1)-Parser: Funktionsweise (1)

Für Eingabe $a_1 a_2 \ldots a_m$ starte mit:

Stack	Eingabe
$\$s$	$a_1 a_2 \ldots a_m \$$

wobei s der Startzustand ist

Beachte: Oberster Zustand auf dem Stack ist stets der aktuelle „Zustand des Parsers“

SLR(1)-Parser: Funktionsweise (2)

Abarbeitung für:

Stack	Eingabe
$\$s_1 X_1 s_2 X_2 \ldots s_n X_n s_{n+1}$	$a_1 a_2 \ldots a_m \$$

berechne $aktion(s_{n+1}, a_1)$ (bzw. $aktion(s_{n+1}, \$)$, wenn Eingabe leer).

- Bei „akzeptiere“ bzw. „Fehler“ stoppe.

SLR(1)-Parser: Funktionsweise (3)

- Bei „schiebe und gehe zu Zustand s_j“:

Stack	Eingabe
$\$s_1 X_1 s_2 X_2 \ldots s_n X_n s_{n+1}$	$a_1 a_2 \ldots a_m \$$
$\$s_1 X_1 s_2 X_2 \ldots s_n X_n s_{n+1} a_1 s_j$	$a_2 \ldots a_m \$$

D.h. schiebe linkes Symbol der Eingabe auf den Stack und lege neuen Zustand s_j auf den Stack

SLR(1)-Parser: Funktionsweise (4)

- Bei „reduziere mit Produktion $E \to X_1 \dots X_m$“.

Stack	Eingabe
$\$s_1 Y_1 s_2 Y_2 \dots s_n \underbrace{X_1 \dots s_{n+m-1} X_m s_{n+m}}_{\text{entferne } 2*m \text{ Einträge}}$	$a_1 a_2 \dots a_m \$$
$\$s_1 Y_1 s_2 Y_2 \dots s_n E s_j$	$a_1 a_2 \dots a_m \$$

  wobei $goto(s_n, E) = s_j$

Beachte:

- Auf dem Stack werden die Zustände s_k gespeichert.
- Beim Reduzieren kann daher mit unterschiedlichen Zuständen fortgefahren werden
- Das bestimmt die Goto-Tabelle!

SLR(1)-Parser: Beispiel (1)

Grammatik: (Startsymbol $\mathbf{E}$, Z Token für eine Zahl):

$$\mathbf{E} ::= \mathbf{E} + \mathbf{E} \mid (\mathbf{E}) \mid Z$$

Annahme: $+$ als linksassoziativ bekannt

SLR(1)-Parser: Beispiel (2)

Aktions- und Goto-Tabelle (generiert, Erstellung ist kompliziert!)

Zustand	Aktion					Goto
Symbol	$	Z	(	)	+	$\mathbf{E}$
0		(s,3)	(s,4)			5
1		(s,3)	(s,4)			2
2					(s,6)	
3	(r,$\mathbf{E} \to Z$)			(r,$\mathbf{E} \to Z$)	(r,$\mathbf{E} \to Z$)	
4		(s,3)	(s,4)			7
5	akzeptiere				(s,6)	
6		(s,3)	(s,4)			9
7				(s,8)	(s,6)	
8	(r,$\mathbf{E} \to (\mathbf{E})$)			(r,$\mathbf{E} \to (\mathbf{E})$)	(r,$\mathbf{E} \to (\mathbf{E})$)	
9	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)			(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)	

- leere Einträge = Fehler
- (s,i) = Schiebe und gehe zu Zustand i
- (r,Produktion) = reduziere mit Produktion

SLR(1)-Parser: Beispiel (3)

Stack	Eingabe	Aktion
$\$0$	$a+b+(c+d)\$$	(s,3)
$\$0, a, 3$	$+b+(c+d)\$$	(r,$\mathbf{E} \to Z$)
$\$0, \mathbf{E}, 5a, 3$	$+b+(c+d)\$$	(s,6)
$\$0, \mathbf{E}, 5, +, 6$	$b+(c+d)\$$	(s,3)
$\$0, \mathbf{E}, 5, +, 6, b, 3$	$+(c+d)\$$	(r,$\mathbf{E} \to Z$)
$\$0, \mathbf{E}, 5, +, 6, \mathbf{E}, 9b, 3$	$+(c+d)\$$	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)
$\$0, \mathbf{E}, 5\mathbf{E}, 5, +, 6, \mathbf{E}, 9$	$+(c+d)\$$	(s,6)

Zustand	Aktion					Goto
Symbol	$	Z	(	)	+	$\mathbf{E}$
0		(s,3)	(s,4)			5
1		(s,3)	(s,4)			2
2					(s,6)	
3	(r,$\mathbf{E} \to Z$)			(r,$\mathbf{E} \to Z$)	(r,$\mathbf{E} \to Z$)	
4		(s,3)	(s,4)			7
5	akzeptiere				(s,6)	
6		(s,3)	(s,4)			9
7				(s,8)	(s,6)	
8	(r,$\mathbf{E} \to (\mathbf{E})$)			(r,$\mathbf{E} \to (\mathbf{E})$)	(r,$\mathbf{E} \to (\mathbf{E})$)	

SLR(1)-Parser: Beispiel (4)

usw.:

Stack	Eingabe	Aktion
$\$0, \mathbf{E}, 5$	$+(c+d)\$$	(s,6)
$\$0, \mathbf{E}, 5, +, 6$	$(c+d)\$$	(s,4)
$\$0, \mathbf{E}, 5, +, 6, (, 4$	$c+d)\$$	(s,3)
$\$0, \mathbf{E}, 5, +, 6, (, 4, c, 3$	$+d)\$$	(r,$\mathbf{E} \to Z$)
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7$	$+d)\$$	(s,6)
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7, +, 6$	$d)\$$	(s,3)
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7, +, 6, d, 3$	$)\$$	(r,$\mathbf{E} \to Z$)
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7, +, 6, \mathbf{E}, 9$	$)\$$	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7$	$)\$$	(s,8)
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7,), 8$	$\$$	(r,$\mathbf{E} \to (\mathbf{E})$)
$\$0, \mathbf{E}, 5, +, 6, \mathbf{E}, 9$	$\$$	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)
$\$0, \mathbf{E}, 5$	$\$$	akzeptiere

SLR(1)-Parser: Bemerkungen

- In der Implementierung reicht es, nur die Zustände auf dem Stack zu speichern (nicht die Nicht-(Terminale))
- Aktions- und Goto-Tabellen erstellen behandeln wir nicht
- Diese werden aber auch von Parsergeneratoren automatisch erstellt

Happy

Parsergenerator Happy

Parsergeneratoren

- Parsergeneratoren sind Werkzeuge um Parser automatisch erzeugen zu lassen
- Eingabe: Basisgrammatik + Festlegungen (Prioritäten, Assoziativitäten,...)
- Ausgabe: Meistens ein SLR(1)-Parser

Bekannteste Generatoren: Yacc und Bison für C

Es gibt jede Menge Parsergeneratoren:

`http://en.wikipedia.org/wiki/Comparison_of_parser_generators`

Happy: Ein Parsergenerator für Haskell

- Syntax der Spezifikation Yacc-ähnlich
- Homepage haskell.org/happy
- Erzeugt einen SLR(1)-Parser in Haskell
- Ausgabe des Parsers ist ein Syntaxbaum
  (definiert als Haskell-Datentyp)

Happybenutzung am Beispiel

Grammatik

$$
\begin{array}{rcl}
\mathbf{E} & ::= & \mathbf{E} + \mathbf{E} \\
& | & \mathbf{E} - \mathbf{E} \\
& | & \mathbf{E} * \mathbf{E} \\
& | & \mathbf{E} \,/\, \mathbf{E} \\
& | & (\ \mathbf{E}\ ) \\
& | & \mathit{zahl}
\end{array}
$$

Lexer

Selbst implementiert (einfach)

```
data Token = TokenInt Int
           | TokenSymbol Char
 deriving(Show)

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
 | c `elem` ['+','-','*','/','(',')'] =
                (TokenSymbol c) : lexer cs
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise = error ("error, can't lex symbol " ++ show c)

lexNum cs = TokenInt (read num) : lexer rest
             where (num,rest) = span isDigit cs
```


Lexer (2)

Erstellt aus der Eingabe einen Tokenstrom:

```
*> lexer "3+(4*5)"
[TokenInt 3,TokenSymbol '+',TokenSymbol '(',
 TokenInt 4,TokenSymbol '*',TokenInt 5,TokenSymbol ')']

*> lexer "1!+2"
*** Exception: parse error, can't lex symbol '!'
```

Happy: Aufbau der Parserspezifikation

- Dateiendung `.y` `Parser.y`
- Dateiinhalt

  Modulkopf (optional)
  Parserdirektiven
  %%
  Grammatik
  Modulschluss (optional)
- Aufruf `happy Parser.y` erzeugt SR-Parser in der Datei `Parser.hs`.

Aufbau der Parserspezifikation: Modulkopf

Modulkopf (optional)
Parserdirektiven
%%
Grammatik
Modulschluss (optional)

- Block Haskell-Code
- von geschweiften Klammern umschlossen
- Definition des Modulnamens, Exportliste
- Modulimporte

Für unser Beispiel:

```
{
module Parser where
import Data.Char
}
```


Parserspezifikation: Parserdirektiven

Modulkopf (optional)
Parserdirektiven
%%
Grammatik
Modulschluss (optional)

- `%name` *NAME* Namen der Parserfunktion festlegen. Unter diesem Namen kann der Parser später aufgerufen werden.
- `%tokentype` { *TYPE* } Typ der Tokens (Eingabe des Parsers)
- `%token` *MATCHLIST* Zuweisen der Token zu den Terminalen in der Grammatik
- Prioritäten / Assoziativitäten (später)

Beispiel: Parserdirektiven

```
%name parser
%tokentype { Token }
%token
        int         { TokenInt $$ }
        '+'         { TokenSymbol '+' }
        '-'         { TokenSymbol '-' }
        '*'         { TokenSymbol '*' }
        '/'         { TokenSymbol '/' }
        '('         { TokenSymbol '(' }
        ')'         { TokenSymbol ')' }
```

- links Terminale, rechts die Token
- Normalerweise: Wert ist das Token selbst
- Symbol $$ repräsentiert den Wert des Tokens
- Daher: Bei `TokenInt` *zahl* ist der Wert die Zahl selbst.

Parserspezifikation: Grammatik

Modulkopf (optional)
Parserdirektiven
%%
Grammatik
Modulschluss (optional)

- BNF-ähnliche Notation
- Auch Ausgabetyp des Parsers festlegen
- Verarbeitung definieren

Beispiel: Grammatik

```
%%
E :: { Expr }
E : E '+' E          { Plus  $1 $3 }
  | E '-' E          { Minus $1 $3 }
  | E '*' E          { Times $1 $3 }
  | E '/' E          { Div $1 $3 }
  | '(' E ')'        { $2 }
  | int              { Number $1 }
```

- Nichtterminal `E`
- Ausgabe Syntaxbaum vom Typ:
  ```
  data Expr = Plus Expr Expr  | Minus Expr Expr
            | Times Expr Expr | Div Expr Expr | Number Int
  ```
- anstelle von ::= wird `:` verwendet
- in `{ ... }` stehen die Aktionen bei Benutzen der Regel
- $\$i = i$. Symbol (Token) in der rechten Seite der Regel
- Aktion muss vom Ausgabetyp sein (`Expr`)

Parserspezifikation: Modulschluss

Modulkopf (optional)
Parserdirektiven
%%
Grammatik
Modulschluss (optional)

- Haskell-Code
- In geschweiften Klammern
- Kann z.B. den Lexer enthalten
- Sollte die Funktion `happyError ::[Token] -> a` definieren
- Wird im Fehlerfall aufgerufen

Beispiel: Modulschluss

```
{
happyError :: [Token] -> a
happyError [] = error "parse error: unerwartetes Ende"
happyError xs = error ("parse error:" ++ show xs)

...
}
```

Beispiel komplett

```
{
module Parser where
import Data.Char
}
%name parser
%tokentype { Token }
%token
        int         { TokenInt $$ }
        '+'         { TokenSymbol '+' }
        '-'         { TokenSymbol '-' }
        '*'         { TokenSymbol '*' }
        '/'         { TokenSymbol '/' }
        '('         { TokenSymbol '(' }
        ')'         { TokenSymbol ')' }
%%
E :: { Expr }
E : E '+' E          { Plus  $1 $3 }
  | E '-' E          { Minus $1 $3 }
  | E '*' E          { Times $1 $3 }
  | E '/' E          { Div $1 $3 }
  | '(' E ')'        { $2 }
  | int              { Number $1 }
{
happyError :: [Token] -> a
happyError [] = error "error: unerwartetes Ende"
happyError xs = error ("error:" ++ show xs)
data Token = TokenInt Int | TokenSymbol Char
 deriving(Show)

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
 | c `elem` ['+','-','*','/','(',')'] =
                 (TokenSymbol c) : lexer cs
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise =
    error ("error, can't lex symbol "
            ++ show c)

lexNum cs = TokenInt (read num) : lexer rest
              where (num,rest) = span isDigit cs

data Expr = Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
          | Number  Int
 deriving(Show)
}
```

Parser1.y

Beispiel: Happy-Aufruf

```
> happy Parser1.y
shift/reduce conflicts:  16
```

Aufruf mit -iDatei erstellt eine Info-Datei:

```
happy -iInfoDatei Parser1.y
```


Info-Datei

```
-----------------------------------------------------------------------------
Info file generated by Happy Version 1.18.6 from Parser1.y
-----------------------------------------------------------------------------

state 12 contains 4 shift/reduce conflicts.
state 13 contains 4 shift/reduce conflicts.
state 14 contains 4 shift/reduce conflicts.
state 15 contains 4 shift/reduce conflicts.

-----------------------------------------------------------------------------
Grammar
-----------------------------------------------------------------------------
%start_parser -> E                              (0)
E -> E '+' E                                    (1)
E -> E '-' E                                    (2)
E -> E '*' E                                    (3)
E -> E '/' E                                    (4)
E -> '(' E ')'                                  (5)
E -> int                                        (6)

-----------------------------------------------------------------------------
Terminals
-----------------------------------------------------------------------------
int             { TokenInt $$ }
'+'             { TokenSymbol '+' }
'-'             { TokenSymbol '-' }
'*'             { TokenSymbol '*' }
'/'             { TokenSymbol '/' }
'('             { TokenSymbol '(' }
')'             { TokenSymbol ')' }
```


Info-Datei (2)

```
-----------------------------------------------------------------------------
Non-terminals
-----------------------------------------------------------------------------
%start_parser   rule  0
E               rules 1, 2, 3, 4, 5, 6

-----------------------------------------------------------------------------
States
-----------------------------------------------------------------------------
State 0


int             shift, and enter state 3
'('             shift, and enter state 4

E               goto state 5

State 1


int             shift, and enter state 3
'('             shift, and enter state 4

E               goto state 2

State 2

E -> E . '+' E                                  (rule 1)
E -> E . '-' E                                  (rule 2)
E -> E . '*' E                                  (rule 3)
E -> E . '/' E                                  (rule 4)
```

Info-Datei (3)

```
State 12

E -> E . '+' E                                  (rule 1)
E -> E . '-' E                                  (rule 2)
E -> E . '*' E                                  (rule 3)
E -> E . '/' E                                  (rule 4)
E -> E '/' E .                                  (rule 4)

'+'             shift, and enter state 6
(reduce using rule 4)

'-'             shift, and enter state 7
(reduce using rule 4)

'*'             shift, and enter state 8
(reduce using rule 4)

'/'             shift, and enter state 9
(reduce using rule 4)

')'             reduce using rule 4
%eof            reduce using rule 4
...
```


Nochmal Direktiven

Modulkopf (optional)
Parserdirektiven
%%
Grammatik
Modulschluss (optional)

Prioritäten und Assoziativitäten

- `%left` *Terminal(e)* linksassoziativ
- `%right` *Terminal(e)* rechtsassoziativ
- `%nonassoc` *Terminal(e)* nicht assoziativ
- Priorität wird durch die Reihenfolge festgelegt: früher = niedrigere Priorität

Beispiel

Einfügen von

```
%left '+' '-'
%left '*' '/'
```

Anschließend hat der Parser keine Konflikte mehr und kann verwendet werden.
z.B.:

```
*> parser (lexer "12+999")
Plus (Number 12) (Number 999)
```


Fazit

- Funktionsweise von Shift-Reduce-Parsern
- SR-Parser können für Operatorgrammatiken leicht programmiert werden
- Basisgrammatik + Angaben (Prioritäten, Assoziativitäten) zu den Operatoren
- Im allgemeinen: SR-Parser besser generieren lassen, als selbst programmieren
- Beispiel für einen Parsergenerator: Happy