

5.5 Aufsteigende Analyse

5. Syntaxanalyse und der Parser-Generator yacc

- 5.1 Einleitung
- 5.2 Kontextfreie Grammatiken
- 5.3 Grundlagen von yacc
- 5.4 Absteigende Analyse
- 5.5 Aufsteigende Analyse

Absteigende Analyse versus aufsteigende Analyse

- **Absteigende Analyse** (top-down-Analyse)
 - Knoten des Ableitungsbaums werden von der Wurzel her konstruiert
 - leichter von Hand zu programmieren
 - „Praktische Informatik 2“
- **Aufsteigende Analyse** (bottom-up-Analyse)
 - Knoten des Ableitungsbaums werden von den Blättern her konstruiert
 - größere Klasse von Grammatiken
 - seltener Grammatiktransformation von Hand nötig
 - yacc

5.5 Aufsteigende Analyse

- 5.5.1 Prinzip der aufsteigenden Analyse
- 5.5.2 Algorithmus der LR-Syntaxanalyse
- 5.5.3 Konstruktion der Syntaxanalysetabellen
- 5.5.4 Konflikte
- 5.5.5 Präzedenzen
- 5.5.6 Fehlerbehandlung

Prinzip der aufsteigenden Analyse

- „reduzieren“ des Eingabeworts auf das Startsymbol
 - Reduktionsschritt:
 - wenn: ein Teilwort = rechte Seite einer Regel
 - ersetze Teilwort durch Symbol auf linker Seite
 - richtige Wahl des Teilworts nötig
 - liefert Rechtsableitung in umgekehrter Reihenfolge
 - Name auch: „shift-reduce-Syntaxanalyse“



Prinzip: Beispiel

- Grammatik:
 - $s \rightarrow AabE$
 - $a \rightarrow aBC \mid B$ ← (ist übrigens keine LL-Grammatik!)
 - $b \rightarrow D$
- Eingabe: ABBCDE
- Schritte:
 - ABBCDE
 - AaBCDE
 - AaDE
 - AabE
 - s
- Rechtsableitung:
 $s \Rightarrow_r AabE \Rightarrow_r AaDE \Rightarrow_r AaBCDE \Rightarrow_r ABBCDE$

Implementierung der aufsteigenden Analyse mit einem Stack

- Idee:
 - lies jeweils ein Zeichen und packe es oben auf den Stack
 - wenn ein passendes Muster im Stack steht, wende eine Grammatikregel rückwärts an
 - Stack:
 - repräsentiert die bisher gelesene Eingabe
 - bereits erkannte Teile sind zu Nichtterminalen verdichtet



Stack-Implementierung: Beispiel

Stack	Eingabe
	ABBCDE
A	BBCDE
AB	BCDE
Aa	BCDE
AaB	CDE
AaBC	DE
Aa	DE
AaD	E
Aab	E
AabE	
s	

Grundoperationen eines Shift-Reduce-Parsers

1. schieben (shift)
 - nächstes Eingabesymbol → Stack
 2. reduzieren (reduce)
 - Stack: oberste Symbole → Nichtterminal
 3. akzeptieren
 4. Fehler melden
- Satz:
Beim Reduzieren reicht es, nur die obersten Symbole auf dem Stack zu betrachten
 - Beweis: Drachenbuch

Konflikte bei der Shift-Reduce-Syntaxanalyse

- Parser kann nicht entscheiden, was zu tun ist
 - obwohl er den ganzen Stack kennt
 - obwohl er das nächste Eingabesymbol kennt
 - bzw. k nächste Symbole bei LR(k)-Grammatik
- mögliche Konflikte:
 - schieben oder reduzieren?
 - shift/reduce-Konflikt
 - welche mehrerer Reduktionen?
 - reduce/reduce-Konflikt
- Grammatik ist dann nicht LR (bzw. LR(k))

Shift/Reduce-Konflikt: Beispiel

- mehrdeutige Grammatik:
 - $statement ::= 'if' expression 'then' statement$
| $'if' expression 'then' statement 'else' statement$
| $other$
- aktuelle Parser-Konfiguration:

Stack	Eingabe
... 'if expression 'then' statement	'else' ...
- 'else' schieben oder 4 Symbole reduzieren??
 - es gibt mehrere Parse-Bäume
 - mehrdeutige Grammatik ist niemals LR

Reduce/Reduce-Konflikt: Beispiel

- mehrdeutige Grammatik:
 - $kinder ::= maedchen | jungen$
 $maedchen ::= KIRSTEN | BIRGIT | EIKE | KAI$
 $jungen ::= JAN | ULRICH | EIKE | KAI$
 - gehört EIKE zu *maedchen* oder *jungen*??

Stack	Eingabe
... EIKE	...

Reduce/Reduce-Konflikt: Auflösung

- mögliche eindeutige Grammatik:
 - $kinder ::= maedchen \mid jungen \mid maedchenOderJungen$
 - $maedchen ::= KIRSTEN \mid BIRGIT$
 - $jungen ::= JAN \mid ULRICH$
 - $maedchenOderJungen ::= EIKE \mid KAI$
 - ist *andere* Grammatik!

Absteigende Analyse versus aufsteigende Analyse (2)

- Parser muß rechte Seite einer Regel erkennen, wenn ...
 - aufsteigend, LR(k):
 - ... alles gesehen, was von dieser Regel abgeleitet wird, plus k Symbolen Vorschau
 - absteigend, LL(k):
 - ... die ersten k Symbole dessen gesehen, was von dieser Regel abgeleitet wird
 - daher LL(k)-Parser weniger mächtig

5.5 Aufsteigende Analyse

- 5.5.1 Prinzip der aufsteigenden Analyse
- 5.5.2 Algorithmus der LR-Syntaxanalyse
- 5.5.3 Konstruktion der Syntaxanalysetabellen
- 5.5.4 Konflikte
- 5.5.5 Präzedenzen
- 5.5.6 Fehlerbehandlung

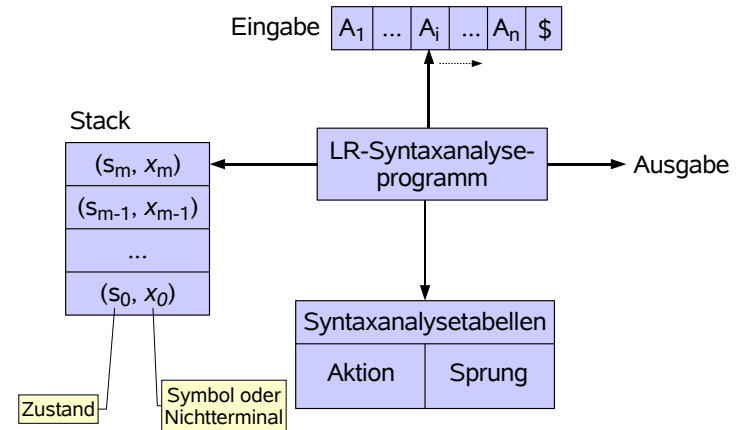
LR-Syntaxanalyse

- eine (von mehreren) aufsteigende Analyseverfahren
- yacc verwendet sie
 - genauer: LALR(1)
- LR(k)
 - links-nach-rechts lesen, Rechtsableitung umgekehrt, Vorausschau um k Symbole
 - LR(1) = LR

LR-Syntaxanalyse: Eigenschaften

- fast allgemeinste Methode für kontextfreie Grammatiken
- allgemeinste bekannte aufsteigende Analyse ohne Backtracking
- erkennt Syntaxfehler an frühestmöglicher Stelle beim Lesen von links
- LR-Parser sehr aufwendig von Hand
 - Lösung: Parser-Generator

LR-Parser



LR-Syntaxanalyseprogramm

- betrachtet
 - Zustand s_m oben auf Stack und
 - aktuelles Eingabesymbol A_i
- Aktion[s_m, A_i] kann sein
 - schiebe Zustand s
 - reduziere mit Regel $a \rightarrow \beta$
 - akzeptiere
 - Fehler
- Sprung
 - Zustand x Nichtterminal \rightarrow Zustand
 - deterministischer endlicher Automat

Schiebe Zustand s

- liest ein Eingabesymbol A_i
- legt Zustand s und Eingabesymbol A_i auf Stack
 - neu: Zustand mit auf Stack
 - Zustand oben auf Stack repräsentiert *gesamten* Stack
 - deshalb kein Vergleich von Regeln mit Stack nötig!

Reduziere mit Regel $a \rightarrow \beta$

- Eingabe unverändert
- löscht $|\beta|$ Symbole vom Stack
 - ohne sie anzusehen
 - s_f : freigelegter Zustand oben auf Stack
- legt Zustand Sprung[s_f, a] und Nichtterminal a auf Stack

Ausgabe des LR-Parsers

- nach Reduzier-Aktion
- durch semantische Aktion zur Syntaxregel
 - siehe später

Implementierung des Stacks

- für jedes Stackelement j gilt:
Zustand s_j bestimmt eindeutig
Symbol/Nichtterminal x_j
 - x_j muß nicht wirklich gespeichert werden

Algorithmus

- Initialisierung:
 - Stack: `push((s0, dummy));` (s_0 : Parser-Startzustand)
 - Eingabezeiger: `i := Anfang der Eingabe A;`
- Schleife:
 - `(s, x) := top;`
 - **case** Aktion[s, A_i] **of**
 - „schiebe s' “: `push((s', Ai)); i++;`
 - „reduziere $a \rightarrow \beta$ “:
`|\beta| x pop; (sf, y) := top; push((Sprung[sf, a], a));`
 - „akzeptiere“: **return**;
 - **sonst: Fehler**;





Beispiel: Eindeutige Ausdrücke

1. $e \rightarrow e + t$
2. $e \rightarrow t$
3. $t \rightarrow t * f$
4. $t \rightarrow f$
5. $f \rightarrow (e)$
6. $f \rightarrow ID$

Syntaxanalysetabellen

Zustand	Aktion						Zustand	Sprung		
	ID	+	*	()	\$		Nichtterminal	e	t
0	s5			s4			0	1	2	3
1		s6				akz	1			
2		r2	s7		r2	r2	2			
3		r4	r4		r4	r4	3			
4	s5			s4			4	8	2	3
5		r6	r6		r6	r6	5			
6	s5			s4			6		9	3
7	s5			s4			7			10
8		s6			s11		8			
9		r1	s7		r1	r1	9			
10		r3	r3		r3	r3	10			
11		r5	r5		r5	r5	11			



Demo



Beispielablauf

	Stack	Eingabe	Aktion
1	(0,?)	ID * ID + ID \$	schiebe 5
2	(0,?) (5,ID)	* ID + ID \$	reduziere durch 6. $f \rightarrow ID$
3	(0,?) (3,f)	* ID + ID \$	reduziere durch 4. $t \rightarrow f$
4	(0,?) (2,t)	* ID + ID \$	schiebe 7
5	(0,?) (2,t) (7,*)	ID + ID \$	schiebe 5
6	(0,?) (2,t) (7,*) (5,ID)	+ ID \$	reduziere durch 6. $f \rightarrow ID$
7	(0,?) (2,t) (7,*) (10,f)	+ ID \$	reduziere durch 3. $t \rightarrow t * f$
8	(0,?) (2,t)	+ ID \$	reduziere durch 2. $e \rightarrow t$
9	(0,?) (1,e)	+ ID \$	schiebe 6
10	(0,?) (1,e) (6,+)	ID \$	schiebe 5
11	(0,?) (1,e) (6,+) (5,ID)	\$	reduziere durch 6. $f \rightarrow ID$
12	(0,?) (1,e) (6,+) (3,f)	\$	reduziere durch 4. $t \rightarrow f$
13	(0,?) (1,e) (6,+) (9,t)	\$	reduziere durch 1. $e \rightarrow e + t$
14	(0,?) (1,e)	\$	akzeptiere



Das Beispiel mit bison (1)

• expr-unambig.l

```
%{
#include "expr-unambig.tab.h"
}%
%option noyywrap
%option nodefault
%%
"+"                |
"*"                |
"("                |
")"                | return *yytext; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+             /* ueberspringe White-Space */
\n                 { return 0; /* Nur eine Zeile lesen. */ }
.                  { return ILLEGAL_CHAR; }
```



Das Beispiel mit bison (2)

- expr-unambig.y

```
 %{
 #define YYERROR_VERBOSE
 void yyerror(char *);
 }
 %debug
 %verbose
 %token '+' '*' '(' ')' ID ILLEGAL_CHAR
 %%
 e:   e '+' t
 ;
 e:   t
 ;
 t:   t '*' f
 ;
 t:   f
 ;
 f:   '(' e ')'
 ;
 f:   ID
 ;
 %%
 void yyerror(char *msg) {
     printf("\nEingabefehler: %s\n", msg);
 }
 int main() {
     yydebug = 1;
     return yyparse();
 }
```



Das Beispiel mit bison (3)

- %verbose
 - erzeugt expr-unambig.output mit Informationen über generierten Parser
- YYERROR_VERBOSE
 - Parser gibt genauere Fehlermeldungen
- %debug und „yydebug = 1“ zusammen
 - schaltet Debug-Ausgabe ein

5.5 Aufsteigende Analyse

5.5.1 Prinzip der aufsteigenden Analyse

5.5.2 Algorithmus der LR-Syntaxanalyse

→ 5.5.3 Konstruktion der Syntaxanalysetabellen

5.5.4 Konflikte

5.5.5 Präzedenzen

5.5.6 Fehlerbehandlung

Idee der SLR-Methode

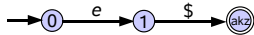
- Erinnerung:
 - Stack:
 - repräsentiert die bisher gelesene Eingabe
 - bereits erkannte Teile sind zu Nichtterminalen verdichtet
- Idee:
 - konstruiere endlichen Automaten, der einen korrekten verdichteten Stackinhalt erkennt





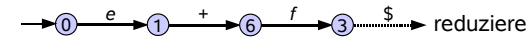
Idee der SLR-Methode (2)

- siehe Folie „Beispielablauf“ oben:
 - Zeile 13, während des Reduzierens:
 - Zustand s_f ist 0 (Startzustand)
 - Nichtterminal e wird auf leeren Stack gelegt
 - neuer Zustand: Sprung[0, e] = 1
 - Zeile 14:
 - Eingabe ist $\$$
 - Aktion[1, $\$$] = akz
 - Teilautomat dazu:



Idee der SLR-Methode (3)

- siehe Folie „Beispielablauf“ oben:
 - Zeile 12:
 - Stack enthält: $\langle e, +, f \rangle$
 - Eingabe ist $\$$
 - Zustände auf Stack: $\langle 0, 1, 6, 3 \rangle$
 - Sprung[0, e] = 1, Aktion[1, $+$] = s 6,
Sprung[6, f] = 3, Aktion[3, $\$$] = r 4
 - Teilautomat dazu:



Idee der SLR-Methode (4)

- endlicher Automat *nicht* für Eingabewort
 - weil keine reguläre Menge
- endlicher Automat für verdichtetes Eingabewort auf Stack
- nichtdeterministisch
 - weil viele Teilautomaten zusammen „geklebt“ werden
 - z.B. (mind.) zwei Aktionen für „e“ in Zustand 0
- Stack von Zuständen/Automaten
 - für richtige Fortsetzung nach einem Verdichtungsschritt
 - kann *ein* endlicher Automat nicht



Idee der SLR-Methode (5)

- nichtdeterministischer EA
 - deterministischer EA
 - durch bekannte Potenzmengenkonstruktion

LR(0)-Element

- eine Regel mit einem Punkt irgendwo auf der rechten Seite
 - Intuition:
 - wieviel bereits von Regel gesehen
- Beispiele:
 - Regel: $a \rightarrow x y z$
 - LR(0)-Elemente:
 - $a \rightarrow \cdot x y z$
 - $a \rightarrow x \cdot y z$
 - $a \rightarrow x y \cdot z$
 - $a \rightarrow x y z \cdot$

Regel: $a \rightarrow \epsilon$
LR(0)-Element:
 $a \rightarrow \cdot$



Idee der SLR-Methode (6)

- ein LR(0)-Element:
ein Zustand des NEA
- eine Menge von LR(0)-Elementen:
ein Zustand des DEA
- Potenzmengenkonstruktion
 - wieder nötige Operationen:
 - Hülle (ϵ -Hülle)
 - Sprung („move“)
 - siehe Kapitel „lexikalische Analyse“



Idee der SLR-Methode (7)

- Konstruktion der Funktionen für „Aktion“ und „Sprung“:
 - mit den Ergebnissen dieser Operationen
 - Details: siehe „Drachenbuch“

LR(1)-Grammatik, die nicht SLR(1) ist

- Grammatik (Zuweisung in C)
 - $s \rightarrow l = r$
 - $s \rightarrow r$
 - $l \rightarrow * r$
 - $l \rightarrow ID$
 - $r \rightarrow l$
- Shift/Reduce-Konflikt:
Zustand 2 des DEA
mit LR(0)-Elementen
 $s \rightarrow l \cdot = r$
 $r \rightarrow l \cdot$
bei Eingabe „=“:
Schieben oder Reduzieren?
nicht genug Information in
LR(0)-Elementen
- ist aber
 - nicht mehrdeutig
 - LALR(1)

Idee der kanonischen LR-Methode

- erweitere LR(0)-Element um ein zulässiges Vorschauzeichen zu „LR(1)-Element“
 - Zustände mit unterschiedlichem Vorschauzeichen werden getrennt
- Tabellen ca. 10 mal so groß
 - für Programmiersprache mit ca. 100 Regeln

Idee der LALR-Methode

- wie eben, aber identifiziere DEA-Zustände mit gleichem „Kern“
 - Kern:
 - Menge von LR(1)-Elementen für DEA-Zustand
 - LR(1)-Element: LR(0)-Element + Vorschauzeichen
 - Vorschauzeichen bei Kern-Vergleich ignoriert
 - Vorschauzeichen in DEA-Transitionen berücksichtigt
 - erzeugt nie Shift/Reduce-Konflikte
 - kann Reduce/Reduce-Konflikte erzeugen
- Tabellengröße: klein wie bei SLR
- für Programmiersprachen in der Regel ausreichend

5.5 Aufsteigende Analyse

- 5.5.1 Prinzip der aufsteigenden Analyse
- 5.5.2 Algorithmus der LR-Syntaxanalyse
- 5.5.3 Konstruktion der Syntaxanalysetabellen
- 5.5.4 Konflikte
- 5.5.5 Präzedenzen
- 5.5.6 Fehlerbehandlung