

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

22. März 2010

Organisatorisches

- ▶ Zeit: Do, 14-16 Uhr (14 Wochen, Feiertage 13.5,3.6.)
- ▶ Ort: Raum 006, Schellingstr. 3
- ▶ Veranstaltung Nr.: 14545
- ▶ Voraussetzung: Kurs über Formale Sprachen und Automaten, Programmierkenntnisse.
- ▶ Scheinkriterium: Vortrag mit Ausarbeitung und aktive Teilnahme

Literatur

1. A.W.Appel: Modern Compiler Implementation in ML/Java/C. Cambridge University Press 1997.
2. A.W.Appel,D.R.Tarditi: ML-Yacc User's Manual. 2000.
SMLNJ-Distribution, <http://www.smlnj.org/doc/ML-Yacc/>
Überarbeitung: R.Price, User's Guide to ML-Lex and ML-Yacc,
<http://rogerprice.org>
3. G.Klein, H.Leiß: ML-GLR User's Manual. CIS, 2009
4. M-J.Nederhof, G.Satta: Efficient Tabular LR Parsing.
Proceedings of the 34th Annual Meeting of the ACL, 1996.
5. A.V.Aho, R.Sethi, J.D.Ullman: Compilers. Principles,
Techniques, and Tools. Addison Wesley, 1986
6. S.Sippu,E.Soisalon-Soininen: Parsing. Vol 1. Springer 1983
7. M.Lange: To CNF or not to CNF? An Efficient Yet Presentable
Version of the CYK Algorithm Informatica Didactica 8, 2009

Ziele des Seminars

1. Theoretisches Ziel: Verständnis des LR-Parsens und eine Implementierung (ML-Yacc)
2. Praktisches Ziel: Umgang mit einem Parsergenerator

Themen

- 1 Themenerläuterung
- 2 Der LR(0)-Automat einer CF-Grammatik. (Aho,Appel u.a.)
- 3 Theorie des LR(0)-Parsens (Aho u.a., Appel u.a.)
- 4 Das Format von ML-Yacc-Grammatiken (Beispiele, Bedienung)
- 5 Berechnung des LR(0)-Automaten in ML-Yacc (Quelldateien)
- 6 ParseGenParser und UserParser (Quelldateien)
- 7 Theorie des GLR-Parsers (Nederhof/Satta)
- 8 ML-GLR: Implementierung des GLR-Parsers (Quelldateien)
- 9 Theorie/Implementierung des GLR-Parsers 2
- 10 Vereinfachung mit Relationen $A \Rightarrow^* \epsilon$, $A \Rightarrow^* B$ (Soisalon,Lange)
- 11 Implementierung der Vereinfachung (ohne sem.Attribute)
- 12 Fehlerbehandlung in ML-Yacc (Quelldateien)
- 13 Fehlerbehandlung für ML-GLR (Theorie/Implementierung)

- 14 MA-Themen: Lexikonbasierter Lexer; Fehlerkorrektur für ML-GLR

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

Theorie des LR(0)-Parsens

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

5. Mai 2010 (mit Nachtrag vom 5.6.2010)

LR-Parsen

Ein LR(0)-Erkenner zur kontextfreien Grammatik $G = (N, \Sigma, P, S)$ ist eine „Keller“-Maschine M_G , deren Konfigurationen die Form

$$(\$ \text{Stapel}, \text{Eingaberest} \$)$$

haben, wobei für die Eingabe $a_1 \cdots a_n = vw$ nach Lesen von v neben den Endmarken $\$$

- ▶ der Stapel die Wurzeln $\alpha \in (N \cup \Sigma)^*$ eines Walds für v enthält,
- ▶ die Resteingabe das ungelesene Suffix $w \in \Sigma^*$ der Eingabe ist.

M_G hat folgende Konfigurationsübergangsmöglichkeiten:

- ▶ $(\$ \alpha, aw \$) \vdash_{\text{shift}_a} (\$ \alpha a, w \$)$, für Eingabesymbole $a \in \Sigma$,
- ▶ $(\$ \alpha \beta, w \$) \vdash_{\text{reduce}_{B \rightarrow \beta}} (\$ \alpha B, w \$)$, für Regeln $(B \rightarrow \beta)$ von G .

M_G erkennt $w \in \Sigma^*$, wenn $(\$, w \$) \vdash^* (\$ S, \$)$.

M_G heißt *shift-reduce*-Erkenner und „ist“ das reguläre Programm

$$(\sum_{a \in \Sigma} \text{shift}_a + \sum_{(A \rightarrow \alpha) \in P} \text{reduce}_{A \rightarrow \alpha})^*.$$

Beispiel (Herbert Lange) Erfolgreiches Parsen

Stapel	Eingabe	Aktion	Regeln von G
\$	(x,(x,x))\$	Shift	1 S \rightarrow (L)
\$(x,(x,x))\$	Shift	2 S \rightarrow x
\$(x	,(x,x))\$	Reduce, Regel 2	3 L \rightarrow S
\$(S	,(x,x))\$	Reduce, Regel 3	4 L \rightarrow L,S
\$(L	,(x,x))\$	Shift	
\$(L,	(x,x))\$	Shift	
\$(L,(x,x))\$	Shift	
\$(L,(x	,x))\$	Reduce, Regel 2	
\$(L,(S	,x))\$	Reduce, Regel 3	
\$(L,(L	,x))\$	Shift	
\$(L,(L,	x))\$	Shift	
\$(L,(L,x))\$	Reduce, Regel 2	
\$(L,(L,S))\$	Reduce, Regel 4	
\$(L,(L))\$	Shift	
\$(L,(L))\$	Reduce, Regel 1	
\$(L,S)\$	Reduce, Regel 4	
\$(L)\$	Shift	
\$(L)	\$	Reduce, Regel 1	
\$S	\$	Accept	

Mögliche Stapelinhalte

Im allgemeinen ist M_G nicht-deterministisch: er kann wählen

- ▶ zwischen verschiedenen *reduce*-Anwendungen (*reduce-reduce-Konflikt*) oder
- ▶ zwischen einer *shift*- und einer *reduce*-Anwendung (*shift-reduce-Konflikt*).

Eine LR(0)-Grammatik ist eine Grammatik, bei in M_G keine Konflikte auftreten, wo also der nächste Schritt eindeutig bestimmt ist, d.h. M_G ist *deterministisch*.

Grundlage für eine effiziente Implementierung von M_G ist, daß man nicht ständig prüft, ob der Stapel γ so in $\alpha\beta$ zerlegt werden kann, daß er mit einer Regel ($B \rightarrow \beta$) auf αB reduzierbar ist. Man erkennt Eigenschaften des Stapels mit einem endlichen Automaten und speichert dessen Zustände.

Reduzierte Linkskontexte von B bzw. $(B \rightarrow \beta)$

Ein $\alpha \in (N \cup \Sigma)^*$ ist ein *reduzierter Linkskontext* von B , wenn αB ein Stapelinhalt einer akzeptierenden Berechnung mit M_G ist, d.h. wenn es eine Rechtsableitung $S \Rightarrow_{rm}^* \alpha B \delta \Rightarrow_{rm}^* w \in \Sigma^*$ gibt.

Beim Parsen braucht man einen Stapel $\alpha\beta$ nur dann mit $(B \rightarrow \beta)$ zu αB zu reduzieren, wenn α ein reduzierter Linkskontext von B ist.

Lemma Sei G eine kontextfreie Grammatik ohne überflüssige Symbole, d.h. $L(B) \neq \emptyset$ für alle $B \in N$. Dann ist

$$rlc(B) := \{ \alpha \in (N \cup \Sigma)^* \mid S \Rightarrow_{rm}^* \alpha B v, v \in \Sigma^* \}$$

eine reguläre Menge.

Beweis: Die $rlc(B)$ kann man simultan induktiv definieren durch:

- ▶ $\epsilon \in rlc(S)$,
- ▶ $\alpha \in rlc(B), (B \rightarrow \gamma C \delta) \in P \Rightarrow \alpha \gamma \in rlc(C)$.

Aber das entspricht einem endlichen System von Bedingungen

$$\begin{aligned} \{\epsilon\} &\subseteq rlc(S), \\ rlc(B)\gamma &\subseteq rlc(C), \quad \text{falls } (B \rightarrow \gamma C\delta) \in P, \end{aligned} \quad (1)$$

wo die Unbekannten $rlc(B)$ nur ganz *links* in den definierenden Ausdrücken $rlc(B)\gamma$ auftreten. Ein linkslineares System über $(N \cup \Sigma)$ definiert aber nur reguläre Teilmengen von $(N \cup \Sigma)^*$. \square

Die Stapelinhalte, in denen eine Reduktion mit $(B \rightarrow \beta)$ möglich ist, bilden dann ebenfalls eine reguläre Menge, nämlich

$$rlc(B \rightarrow \beta) := rlc(B)\beta.$$

Das System (1) mit den Übergängen $rlc(B) \xrightarrow{\gamma} rlc(C)$, erweitert um Zwischenzustände $rlc(B) \cdot X_1 \cdots X_k$ für Präfixe $X_1 \cdots X_k$ der γ , mit $X_i \in N \cup \Sigma$, bildet einen endlichen Automaten, an dessen

Zuständen $rlc(S)\gamma$ man ablesen kann, mit welchen Regeln der Stapel γ reduziert werden kann.

Zustände repräsentieren die für M_G relevanten Eigenschaften der Stapelinhalte. Erkennt man nach Lesen von $\gamma = \alpha\beta$ am erreichten Zustand

$$rlc(B \rightarrow \beta) = rlc(B)\beta,$$

daß mit $(B \rightarrow \beta)$ reduziert werden kann, geht man $|\beta|$ Kanten zurück zu $rlc(B)$, wo die Eigenschaften von α repräsentiert sind, und von dort zum Zustand $rlc(B)B$, der die von αB repräsentiert. Man muß also das möglicherweise sehr lange Stapelpräfix α nicht nochmal lesen, sondern nur die $|\beta| + 1$ Kanten durchlaufen!

Die folgende Konstruktion entspricht einem minimalen deterministischen Automaten des hier skizzierten Automaten.

Konstruktion des LR(0)-Automaten

Aus der Grammatik G berechnet man einen endlichen Automaten.
Sei $P^\dagger := P \cup \{(S^\dagger \rightarrow S)\}$ mit neuem $S^\dagger \notin V := N \cup \Sigma$.

$$I_{LR} := \{(A \rightarrow \alpha \cdot \beta) \mid (A \rightarrow \alpha\beta) \in P^\dagger\}$$

die Menge der *gepunkteten Regeln* oder *LR-items* von G .

Der *Abschluß* einer Menge $q \subseteq I_{LR}$ ist die kleinste Menge $cl(q)$, die abgeschlossen ist unter

$$\frac{I \in q}{I \in cl(q)} (cl_1) \quad \frac{(B \rightarrow \alpha \cdot A\beta) \in cl(q), (A \rightarrow \gamma) \in P}{(A \rightarrow \cdot \gamma) \in cl(q)} (cl_2)$$

Die Funktion $goto : 2^{I_{LR}} \times V \rightarrow 2^{I_{LR}}$ sei

$$goto(q, X) := \{(A \rightarrow \alpha X \cdot \beta) \mid (A \rightarrow \alpha \cdot X\beta) \in cl(q)\}.$$

Der endliche Automat \mathcal{A}_G der LR(0)-Zustände

Der Automat $\mathcal{A}_G = (Q_{LR}, goto, q_{in}, q_{fin})$ von G besteht aus

- ▶ der Zustandsmenge Q_{LR} , der kleinsten Menge $\subseteq I_{LR}$, die unter

$$\frac{q_{in} := \{(S^\dagger \rightarrow \cdot S)\}}{q_{in} \in Q_{LR}}, \quad \frac{q \in Q_{LR}, X \in V, goto(q, X) \neq \emptyset}{goto(q, X) \in Q_{LR}}$$

abgeschlossen ist,

- ▶ der Übergangsfunktion $goto : Q_{LR} \times V \rightarrow Q_{LR}$,
- ▶ dem Anfangs- bzw. Endzustand q_{in} bzw. $q_{fin} := goto(q_{in}, S)$.

\mathcal{A}_G ist ein partieller deterministischer endlicher Automat; statt in den Zustand \emptyset zu gehen, meldet man einen Fehler.

An $(A \rightarrow \alpha \cdot) \in q$ liest man ab, daß M_G mit $(A \rightarrow \alpha)$ reduzieren kann; gibt es in q kein $(A \rightarrow \alpha \cdot)$, muß M_G von der Eingabe lesen.

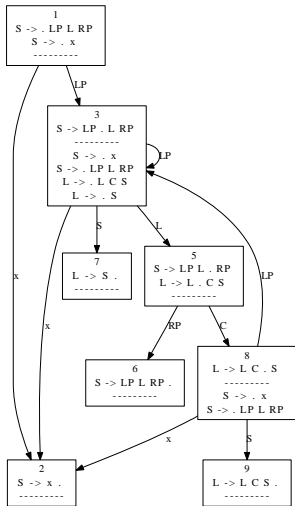
Spezifikation von G mit ML-Yacc

```

%%
%term
    x | LP | RP | C | EOF
%nonterm
    S | L
%pos int
%verbose

%% (* ohne Semantik:() *)
S : LP L RP      ( )
  | x             ( )
L  : S           ( )
  | L C S       ( )
    
```

Automat \mathcal{A}_G



Der Kellerautomat $\mathcal{A}_{LR}(G)$

Statt Symbole von G stapelt man gleich Zustände von Q_{LR} .

Der LR-Automat $\mathcal{A}_{LR} = (\Sigma, Q_{LR}, T_{LR}, q_{in}, q_{fin})$ von G besteht aus

- ▶ dem Eingabealphabet Σ der Terminalsymbole von G ,
- ▶ der Zustandsmenge Q_{LR} aller LR(0)-Zustände,
- ▶ dem Anfangs- bzw. Endzustand q_{in} bzw. $q_{fin} := goto(q_{in}, S)$,
- ▶ der Übergangsrelation T_{LR} , der kleinsten Relation \mapsto mit

$$\frac{a \in \Sigma, q \in Q_{LR}, goto(q, a) = p}{q \xrightarrow{a} qp} \text{ (shift}_a\text{)}$$

$$\frac{A \in N, A\text{-redex } q\delta, goto(q, A) = p}{q\delta \xrightarrow{\epsilon} qp} \text{ (reduce)}.$$

Ein *A-Redex* ist dabei eine Folge $q_0q_1 \cdots q_m \in Q_{LR}^+$, sodaß für geeignete $X_1, \dots, X_m \in V := \Sigma \cup N$

- ▶ $(A \rightarrow \cdot X_1 \cdots X_m) \in cl(q_0)$, und
- ▶ $goto(q_{k-1}, X_k) = q_k$ für $1 \leq k \leq m$.

Beachte, daß dann $(A \rightarrow X_1 \cdots X_m \cdot) \in q_m$, da für $1 \leq k \leq m$ gilt:

$$(A \rightarrow X_1 \cdots X_k \cdot X_{k+1} \cdots X_m) \in q_k \subseteq cl(q_k);$$

Durch T_{LR} wird eine Übergangsrelation $\vdash := \vdash_{shift} \cup \vdash_{reduce}$ auf der Menge $Q_{LR}^+ \times \Sigma^*$ der *Konfigurationen* von \mathcal{A}_{LR} definiert:

- ▶ $(\gamma q, av) \vdash_{shift} (\gamma qp, v) : \iff q \xrightarrow{a} qp \in T_{LR}$
- ▶ $(\gamma q\delta, w) \vdash_{reduce} (\gamma qp, w) : \iff q\delta \xrightarrow{\epsilon} qp \in T_{LR}$.

\mathcal{A}_{LR} erkennt w , wenn $(q_{in}, w) \vdash^* (q_{in}q_{fin}, \epsilon)$.

Erfolgreiches Parsen (H.Lange, Symbol mit LR-Zustand auf dem Stack)

Stack	Eingabe	Aktion
\$1	(x,(x,x))\$	Shift
\$1 (3	x,(x,x))\$	Shift
\$1 (3 x2	,(x,x))\$	Reduce, Rule 2
\$1 (3 S	,(x,x))\$	Goto
\$1 (3 S7	,(x,x))\$	Reduce, Rule 3
\$1 (3 L	,(x,x))\$	Goto
\$1 (3 L5	,(x,x))\$	Shift
\$1 (3 L5 ,8	(x,x))\$	Shift
\$1 (3 L5 ,8 (3	x,x))\$	Shift
\$1 (3 L5 ,8 (3 x2	,x))\$	Reduce, Rule 2
\$1 (3 L5 ,8 (3 S	,x))\$	Goto
\$1 (3 L5 ,8 (3 S7	,x))\$	Reduktion, Rule 3
\$1 (3 L5 ,8 (3 L	,x))\$	Goto
\$1 (3 L5 ,8 (3 L5	,x))\$	Shift
\$1 (3 L5 ,8 (3 L5 ,8	x))\$	Shift
\$1 (3 L5 ,8 (3 L5 ,8 x2))\$	Reduce, Rule 2
\$1 (3 L5 ,8 (3 L5 ,8 S))\$	Goto
\$1 (3 L5 ,8 (3 L5 ,8 S9))\$	Reduce, Rule 4

\$1 (3 L5 ,8 (3 L))\$	Goto
\$1 (3 L5 ,8 (3 L5))\$	Shift
\$1 (3 L5 ,8 (3 L5)6)\$	Reduce, Rule 1
\$1 (3 L5 ,8 S)\$	Goto
\$1 (3 L5 ,8 S9)\$	Reduce, Rule 4
\$1 (3 L)\$	Goto
\$1 (3 L5)\$	Shift
\$1 (3 L5)6	\$	Reduce, Rule 1
\$1 S	\$	Goto
\$1 S4	\$	Accept

Erfolgreiches Parsen, nur mit LR(0)-Zuständen auf dem Stapel:

Stapel	Eingabe	Aktion
\$1	(x,x)\$	Shift
\$1 3	x,x)\$	Shift
\$1 3 2	,x)\$	Reduce, Rule 2
\$1 3 7	,x)\$	Reduce, Rule 3
\$1 3 5	,x)\$	Shift
\$1 3 5 8	x)\$	Shift
\$1 3 5 8 2)\$	Reduce, Rule 2
\$1 3 5 8 9)\$	Reduce, Rule 4
\$1 3 5)\$	Shift
\$1 3 5 6	\$	Reduce, Rule 1
\$1 4	\$	Accept

Beispielparsen eines falschen Satzes

Stack	Eingabe	Aktion
\$1	(x,x),x\$	Shift
\$1 (3	x,x),x\$	Shift
\$1 (3 x2	,x),x\$	Reduce, Rule 2
\$1 (3 S	,x),x\$	Goto
\$1 (3 S7	,x),x\$	Reduce, Rule 3
\$1 (3 L	,x),x\$	Goto
\$1 (3 L5	,x),x\$	Shift
\$1 (3 L5 ,8	x),x\$	Shift
\$1 (3 L5 ,8 x2),x\$	Reduce, Rule 2
\$1 (3 L5 ,8 S),x\$	Goto
\$1 (3 L5 ,8 S9),x\$	Reduce, Rule 4
\$1 (3 L),x\$	Goto
\$1 (3 L5),x\$	Shift
\$1 (3 L5)6	,x\$	Reduce, Rule 1
\$1 S	,x\$	Goto
\$1 S4	,x\$	Fehler

Der Kellerautomat \mathcal{A}_{LR} ist nicht-deterministisch, wenn

- ▶ ein LR(0)-Zustand a zwei verschiedene Regeln $(A \rightarrow \alpha \cdot) \neq (A' \rightarrow \alpha' \cdot)$ enthält (reduce-reduce-Konflikt), oder
- ▶ ein LR(0)-Zustand sowohl eine Regel $(A' \rightarrow \alpha' \cdot a\beta)$ mit $a \in \Sigma$ als auch eine Regel $(A \rightarrow \alpha \cdot)$ enthält (shift-reduce-Konflikt).

Parsergeneratoren wie ML-Yacc melden solche Konflikte, die in begrenztem Umfang durch Operatordeklarationen (implizites Ausschließen von Lesarten) behoben werden können.

Parsergeneratoren wie Yacc geben zu einer LR(0)-Grammatik eine *LR-Steuertabelle*

$Q \setminus V$	a	-	A
q	<i>sn</i>		
p		<i>rm</i>	
q'			<i>p'</i>
	(shift)	(reduce)	(goto)

für den (determ.) Kellerautomaten aus, die wie folgt zu lesen ist:

sn: ist q oben auf dem Stapel und a am Anfang der Eingabe, so schiebe Zustand n auf den Stapel: $goto(q, a) = n$;

rm: ist p oben auf dem Stapel, so reduziere' mit Regel $m = (A \rightarrow \alpha)$, d.h. entferne $|\alpha|$ viele Zustände vom Stapel;

p': ist nach Reduktion' mit $(A \rightarrow \alpha)$ Zustand q' oben auf dem Stapel, so schiebe p' auf den Stapel: $goto(q', A) = p'$.

Steuertabelle nach ML-Yacc: .grm.verbose

```
state 0:
  S : . LP L RP
  S : . x

  x      shift 2
  LP     shift 1
  S      goto 8
  .      error

state 1:
  S : LP . L RP

  x      shift 2
  LP     shift 1
  S      goto 4
  L      goto 3
  .      error

state 2:
  S : x . (reduce)
  .      reduce by rule 1

state 3:
  S : LP L . RP
  L : L . C S

  RP     shift 6
  C      shift 5
  .      error

state 4:
  L : S . (reduce)
  .      reduce by rule 2
```

```
state 5:                                (* Zustandsnummern anders
    L : L C . S                        als im Bild oben *)
```

```
    x      shift 2
    LP     shift 1
    S      goto 7
    .      error
```

```
state 6:
    S : LP L RP . (reduce)
```

```
    .      reduce by rule 0
```

```
state 7:
    L : L C S . (reduce)
```

```
    .      reduce by rule 3
```

```
state 8:
    .      error
```

```
4 of 17 action table entries left after compaction
4 goto table entries
```

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

ML-Yacc: Beispiel und LR(0)-Automat

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

19. Mai 2010

ML-Yacc und ML-Lex

ML-Yacc ist ein Parsergenerator, der

- ▶ aus einer Grammatikspezifikation *G.grm* einen LALR(1)-Parser für die spezifizierte Grammatik erzeugt, *G.grm.sml*
- ▶ aus den in *G.grm* erklärten Terminalen und ihren Werttypen einen Datentyp von Tokens abliest,
- ▶ zum Parsen einen Strom von Token braucht, die zu *G.grm* passen müssen.

ML-Lex ist ein Lexergenerator, der

- ▶ aus einer Lexerspezifikation *G.lex* einen Lexer *G.lex.sml* erzeugt, der einen Zeichenstrom in einen Tokenstrom umformt,
- ▶ mit der Tokendefinition aus einer Grammatik *G.grm* arbeitet.

Der Programmierer muß Lexer *G.lex.sml* und Parser *G.grm.sml* miteinander verbinden.

ML-Yacc: Grammatikspezifikation

Eine mit ML-Yacc spezifizierte Grammatik

- ▶ ist eine deterministische kontextfreie Grammatik (LALR(1)),
- ▶ Terminale und Nonterminale tragen „semantische“ Attribute,
- ▶ Grammatikregeln enthalten Auswertungsregeln für Attribute,
- ▶ Attributwerte sind ML-Objekte, Auswertungsregeln ML-Code

$\langle \text{Format einer Grammatikspezifikation } G.grm \rangle \equiv$

ML-Yacc Benutzerdeklarationen (StandardML-Code)

%%

ML-Yacc Deklarationen (Terminale, Nicht-Terminale, ...)

%%

ML-Yacc Grammatikregeln (A : B C B (Wert(B1,C,B2)))

Beispiel: ml-yacc/examples/calc.grm

(Auszug aus calc.grm, kommentiert)≡

```
fun lookup "bogus" = 10000 (* -Benutzerdeklaration- *)
  | lookup s = 0
%% (* --Yacc-Deklarationen-- *)
%name Calc (* Name der Grammatik *)
%eop EOF SEMI (* end-of-parse symbole *)
%pos int (* Typ der Position eines Symbols im Text *)

%left SUB PLUS (* Dekl. zur Assoziativität *)
%left TIMES DIV
%right CARAT
%subst PRINT for ID (* Dekl. zur Fehlerkorrektur *)
%noshift EOF
%value ID ("bogus")

%verbose (* Erstelle lesbaren Automaten *)
```

(Auszug aus calc.grm, kommentiert)+≡

```
%term ID of string | NUM of int (* Terminale mit bzw.*)  
      | PLUS | TIMES | PRINT |      (* ohne Attributtyp *)  
      SEMI | EOF | CARAT | DIV | SUB
```

```
%nonterm EXP of int (* Nonterminale *)  
      | START of int option
```

```
%% (* --Yacc-Grammatikregeln-- *)
```

```
START : PRINT EXP      (print (Int.toString EXP);  
                        print "\n"; SOME EXP)  
      | EXP (SOME EXP) | (NONE)  
EXP   : NUM (NUM)      | ID (lookup ID)  
      | EXP PLUS EXP   (EXP1+EXP2)  
      | EXP TIMES EXP  (EXP1*EXP2)  
      | EXP DIV EXP    (EXP1 div EXP2)  
      | EXP SUB EXP    (EXP1-EXP2)  
      | EXP CARAT EXP  (let fun e (m,0) = 1  
                        | e (m,1) = m*e(m,1-1)  
                        in e (EXP1,EXP2) end)
```

Spezifikation eines Lexers

Ein mit ML-Lex spezifizierter Lexer

- ▶ muß folgende Typen und Werte deklarieren:
 - ▶ semantische Werte: `type svalue = ...`,
 - ▶ Positionstyp von Terminalen: `type pos = ...`,
 - ▶ Tokentyp: `type ('a,'b) token = ...`
 - ▶ Ergebnis: `type lexresult = (svalue,pos) token`
 - ▶ Funktion zum Stoppen: `val eof : unit -> lexresult`
- ▶ wird durch Match-Regeln `regExp => Token`; beschrieben

$\langle \text{Format einer Lexerspezifikation } G.\text{lex} \rangle \equiv$

ML-Lex Benutzerdeklarationen (* Typen/Werte s.o. *)

%%

ML-Lex Definitionen (* Abkürzungen für regExp's *)

%%

ML-Lex Regeln (* regExp => Token *)

Beispiel: ml-yacc/examples/calc.lex

<Auszug aus der Lexerspezifikation calc.lex>≡

```
                                (* -Benutzerdeklarationen- *)
structure Tokens = Tokens      (* aus der Grammatik abgel.*)

type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult= (svalue,pos) token

val pos = ref 0
fun eof () = Tokens.EOF(!pos,!pos)
fun error (err,line : int,_) = TextIO.output (...)
```

<Auszug aus der Lexerspezifikation calc.lex>+≡

```
%%                                (* ---Lex-Definitionen--- *)
%header (
    functor CalcLexFun(structure Tokens: Calc_TOKENS));
alpha=[A-Za-z];                    (* Abkürzungen für die Regeln *)
digit=[0-9];
white=[\ \t];                       (* -----Lex-Regeln----- *)
%%
\n      => (pos := (!pos) + 1; lex());
{white}+ => (lex());
{digit}+ => (Tokens.NUM (valOf (Int.fromString yytext),
                             !pos, !pos));
"+"      => (Tokens.PLUS(!pos, !pos));
"*"      => (Tokens.TIMES(!pos, !pos));
...
"^"      => (Tokens.CARAT(!pos, !pos));
{alpha}+ => (...);
"."      => (error ("ignore bad character "^yytext,
                  !pos, !pos); lex());
```

Mit der *%header*-Deklaration sagt man, daß die Definition der Token aus einer Struktur `Tokens` genommen werden soll, die aus einer bestimmten Grammatik (siehe *%name G* in *G.grm*) kommt:

```
<Header>≡  
  %header  
  (functor <name>LexFun (structure Tokens:<name>_TOKENS  
                        ... ) : LEXER);
```

Dann wird der Lexer aus dieser Struktur `Tokens` und weiteren Argumenten ... mit einem Funktor `LexFun` gebaut.

ML-Lex erzeugt eine Ausgabedatei `G.lex.sml` mit einem Funktor `GLexFun`, der aus den Tokens der Grammatik eine `makeLexer`-Funktion baut:

$\langle \text{Lexer Funktor zu } G.\text{lex} \rangle \equiv$

```
functor GLexFun(structure Tokens: G_TOKENS ...) : LEXER
  struct
    structure UserDeclarations =
      struct <ML-Lex Benutzerdeklarationen> end
    ... <definiert mit ML-Lex Definitionen
        und Match-Regeln> ...
    fun makeLexer (reader: int -> string) = ...
  end
```

Bem.: Ohne `%header`-Deklaration definiert `G.lex.sml` eine Struktur `Mlex:LEXER` und nimmt dafür `Tokens` aus der Umgebung.

```

<Signatur des Lexers, vgl. ml-yacc/lib/base.sig>≡
signature LEXER =
  sig
    structure UserDeclarations :
      sig
        type ('a,'b) token
        type pos
        type svalue
      end
    val makeLexer : (int -> string) -> unit ->
      (UserDeclarations.svalue,
       UserDeclarations.pos)
      UserDeclarations.token
  end
end

```

D.h. die Funktion `makeLexer` hat den Typ `(int -> string) -> unit -> (svalue, pos) token`, dessen Ergebnistyp der Benutzer festgelegt hat.

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

SML-Strukturen und Aufbau von ML-Yacc

Georg Klein, Hans Leiß
Universität München
Centrum für Informations- und Sprachverarbeitung

22. März 2010

Signaturen, Strukturen und Funktoren

- ▶ Signatur: Typen für Funktionen und Werte einer Struktur
- ▶ Struktur: enthält Funktionen, Werten, Typen, Unterstrukturen
- ▶ Funktor: Funktion, die aus Strukturen eine neue Struktur baut

Struktur MathOps

```
signature MATHSIG =
sig
  val plus: int * int -> int
  val minus: int * int -> int
  val mal: int * int -> int
  val geteilt: int * int -> real
  val fakultaet : int -> int
  exception SmallerThanZero
end
structure MathOps : MATHSIG =
struct
  exception SmallerThanZero
  fun plus(x,y) = x+y
  fun minus(x,y) = x-y
  fun mal(x,y) = x*y
  fun geteilt(x,y) = x div y      (* 7 div 4 = 1 *)
  fun fakultaet 0 = 1
    | fakultaet n =
      if n < 1 then raise SmallerThanZero
      else n * fakultaet(n-1)
end
```

Funktor ListenOpsFun

```
signature LISTENOPS =
sig
  val listensumme: int list -> int
  val listenprodukt: int list -> int
end

functor ListenOpsFun (structure MathematicalOps : MATHSIG)
  : LISTENOPS =
struct
  fun listensumme liste = foldr MathematicalOps.plus 0 liste
  fun listenprodukt liste = foldr MathematicalOps.mal 1 liste
end

(* Funktoraufruf *)
structure ListenOps =
  ListenOpsFun(structure MathematicalOps = MathOps)
```

Aufbau ML-Yacc

- ▶ Library: wird benötigt um die mit ML-Yacc erzeugten Parser auszuführen
 - ▶ abstrakte Lr-Tabelle
 - ▶ abstrakter Lr-Parser für Lr-Tabelle
 - ▶ Join-Funktor um den Parser zusammenzubauen
 - ▶ parse-Funktion um den Parser aufzurufen
- ▶ ML-Yacc Parsergenerator: erzeugt aus einer Grammatikbeschreibung einen Parser
 - ▶ Parser für .grm-Dateien
 - ▶ konkrete Lr-Tabelle
 - ▶ LR(0) Graph
 - ▶ Lookahead
 - ▶ Grammatik (für Reduce)
 - ▶ Programm mit print-Funktionen für <spec>.grm.sml-Datei

```

signature LR_TABLE =
  sig
    datatype ('a,'b) pairlist =
      EMPTY | PAIR of 'a * 'b * ('a,'b) pairlist
    datatype state = STATE of int
    datatype term = T of int
    datatype nonterm = NT of int
    datatype action = SHIFT of state | REDUCE of int
      | ACCEPT | ERROR

    type table
    ...
    val action : table -> state * term -> action
    val goto : table -> state * nonterm -> state
    val mkLrTable :
      {actions : ((term,action) pairlist * action) array,
       gotos : (nonterm,state) pairlist array,
       numStates : int, numRules : int,
       initialState : state} -> table
  end
end

```

link.sml: Baue Parser, der .grm-Dateien verarbeiten kann

```
(* create parser *)
```

```
structure LrVals = MlyaccLrValsFun(  
    structure Token = LrParser.Token  
    structure Hdr = Header)  
structure Lex = LexMlyacc(  
    structure Tokens = LrVals.Tokens  
    structure Hdr = Header)  
structure Parser = JoinWithArg(  
    structure Lex=Lex  
    structure ParserData = LrVals.ParserData  
    structure LrParser= LrParser)  
structure ParseGenParser = ParseGenParserFun(  
    structure Parser = Parser  
    structure Header = Header)
```

link.sml: Tabelle & Verbose-Strukturen bauen

```
(* create structure for computing LALR table from a grammar *)
```

```
structure MakeLrTable = mkMakeLrTable(  
    structure IntGrammar = IntGrammar  
    structure LrTable = LrTable)
```

```
(* create structures for printing LALR tables: *)
```

```
structure Verbose =  
    mkVerbose(structure Errs = MakeLrTable.Errs)  
structure PrintStruct =  
    mkPrintStruct(structure LrTable = MakeLrTable.LrTable  
        structure ShrinkLrTable =  
            ShrinkLrTableFun(structure LrTable=LrTable))
```

link.sml: Zusammenbauen liefert ParseGen.parseGen Funktion

```
(* returns function which takes a file name,  
invokes the parser on the file,  
does semantic checks, creates table, and prints it *)
```

```
structure ParseGen =  
  ParseGenFun(  
    structure ParseGenParser = ParseGenParser  
    structure MakeTable = MakeLrTable  
    structure Verbose = Verbose  
    structure PrintStruct = PrintStruct  
    structure Absyn = Absyn)
```

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

Verallgemeinertes LR-Parsen

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

10. Juni 2010

Generalized LR-Parser: GLR

M.Tomita hat 1986 das LR-Parsen auf nichtdeterministische Grammatiken verallgemeinert:

- ▶ das obere Stapelende wird in Varianten aufgetrennt, wenn mehrere Operationen möglich sind,
- ▶ gleiche Stapelenden werden wieder verschmolzen.

Das führt zu einem *graph-structured stack*, einer Datenstruktur, durch die gemeinsame Teilberechnungen von M_G für mehrere Berechnungswege geteilt werden. Allerdings:

- ▶ Die Implementierung gilt als etwas kompliziert (in GNU Bison ist eine GLR-Option eingebaut).
- ▶ Die worst-case-Komplexität des GLR-Erkennens ist $O(|w|^3)$, wie beim Earley-Algorithmus.

GLR-Parsen mit einer Parsetabelle

J.M.Nederhof und G.Satta haben 1996 eine Vereinfachung vorgeschlagen, die angeblich (praktisch) effizienter ist:

- ▶ um Doppelberechnungen zu vermeiden, wird statt eines Graph-Stapels eine Parsetabelle (Chart) von Zwischenergebnissen benutzt,
- ▶ die Eingabegrammatik G wird in eine binäre Grammatik G' transformiert und G' -Analysen in G -Analysen umgerechnet,
- ▶ zum Parsen wird der übliche Chart-Parser CYK mit einem Wegfiltern unerwünschter Analysen verwendet.

Die Transformation $G \mapsto G'$ beruht auf einer Transformation $\mathcal{A}_{LR} \mapsto \mathcal{A}_{2LR}$ des Kellerautomaten in eine „binäre Form“ und einer Beschreibung der Berechnungen mit \mathcal{A}_{2LR} durch G' . Insofern ist das noch LR-Parsen.

Die Vorteile sollen sein:

1. konzeptuell einfacher: kein Graph-Stapel, sondern nur Grammatiktransformation, CYK-Algorithmus, Parsetabelle
2. kleiner \mathcal{A}_{2LR} : daher schnellere Parsegeneration, schnellerer Parser, weniger Speicherbedarf

Der Kellerautomat \mathcal{A}_{LR} von $G = (N, \Sigma, P, S)$

Erweitere P zu $P^\dagger = P \cup \{(S^\dagger \rightarrow \triangleright S \triangleleft)\}$, $q_{in} := \{(S^\dagger \rightarrow \triangleright \cdot S \triangleleft)\}$.

Der **LR-Automat** $\mathcal{A}_{LR} = (\Sigma, Q_{LR}, T_{LR}, q_{in}, q_{fin})$ von G besteht aus

- ▶ dem Eingabealphabet Σ der Terminalsymbole von G ,
- ▶ der Zustandsmenge Q_{LR} aller LR(0)-Zustände (zu P^\dagger),
- ▶ dem Anfangs- bzw. Endzustand q_{in} bzw. $q_{fin} := goto(q_{in}, S)$,
- ▶ der Übergangsrelation T_{LR} , der kleinsten Relation \mapsto mit

$$\frac{a \in \Sigma, q \in Q_{LR}, goto(q, a) = p}{q \xrightarrow{a} qp} \text{ (shift)}$$

$$\frac{A \in N, A\text{-redex } q\delta, goto(q, A) = p}{q\delta \xrightarrow{\epsilon} qp} \text{ (reduce)}.$$

Ein **A-Redex** ist dabei eine Folge $q_0 q_1 \cdots q_m \in Q_{LR}^+$, sodaß für geeignete $X_1, \dots, X_m \in V := \Sigma \cup N$

- ▶ $(A \rightarrow \cdot X_1 \cdots X_m) \in cl(q_0)$, und \neq Nederhof/Satta
- ▶ $goto(q_{k-1}, X_k) = q_k$ für $1 \leq k \leq m$.

Beachte, daß dann $(A \rightarrow X_1 \cdots X_m \cdot) \in q_m$, da für $1 \leq k \leq m$ gilt:

$$(A \rightarrow X_1 \cdots X_k \cdot X_{k+1} \cdots X_m) \in q_k \subseteq cl(q_k); \quad (2)$$

Durch T_{LR} wird eine Übergangsrelation $\vdash := \vdash_{shift} \cup \vdash_{reduce}$ auf der Menge $Q_{LR}^+ \times \Sigma^*$ der **Konfigurationen** von \mathcal{A}_{LR} definiert:

- ▶ $(\gamma q, av) \vdash_{shift} (\gamma qp, v) : \iff q \xrightarrow{a} qp \in T_{LR}$
- ▶ $(\gamma q\delta, w) \vdash_{reduce} (\gamma qp, w) : \iff q\delta \xrightarrow{\epsilon} qp \in T_{LR}$.

\mathcal{A}_{LR} erkennt w , wenn $(q_{in}, w) \vdash^* (q_{in}q_{fin}, \epsilon)$.

Beispiel $G \mapsto \mathcal{A}_{LR}$

$G = (N, \Sigma, P, S)$ mit Regeln

$S \rightarrow NP VP$

$NP \rightarrow DET As N \mid PN$

$VP \rightarrow V NP$

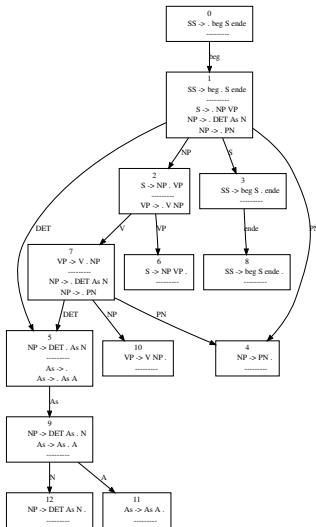
$As \rightarrow \epsilon \mid As A$

implizit :

$S^\dagger \rightarrow \triangleright S \triangleleft$

$\Sigma = \{DET, N, PN, V, A\}$

Der endliche Automat \mathcal{A}_G :



Beispiel: \mathcal{A}_{LR} zu G

Schreibe T_{LR} mit den Nummern der Kernzustände statt der Itemmengen, z.B. $1 = \{S^\dagger \rightarrow \triangleright \cdot S \triangleleft\} = q_{in}$, $3 = \{S^\dagger \rightarrow \triangleright S \cdot \triangleleft\} = q_{fin}$.

Nach (shift) gehören zu T_{LR} : Nach (reduce) gehören zu T_{LR} :

1	\xrightarrow{DET}	1 5
7	\xrightarrow{DET}	7 5
1	\xrightarrow{PN}	1 4
7	\xrightarrow{PN}	7 4
2	\xrightarrow{V}	2 7
9	\xrightarrow{A}	9 11
9	\xrightarrow{N}	9 12

1 2 6	$\xrightarrow{\epsilon}$	1 3
1 5 9 12	$\xrightarrow{\epsilon}$	1 2
1 4	$\xrightarrow{\epsilon}$	1 2
2 7 10	$\xrightarrow{\epsilon}$	2 6
7 5 9 12	$\xrightarrow{\epsilon}$	7 10
7 4	$\xrightarrow{\epsilon}$	7 10
5	$\xrightarrow{\epsilon}$	5 9
5 9 11	$\xrightarrow{\epsilon}$	5 9

Binäre und kompakte Form von \mathcal{A}_{LR}

\mathcal{A}_{LR} durch einen Tabellenparser effizient zu simulieren stößt auf zwei Probleme:

- P1** Reduktionen $q\delta \xrightarrow{\epsilon} qp$ mit $|\delta| > 2$ führen zu intensivem Zurücksetzen/Durchsuchen vieler Felder einer Parsetabelle,
- P2** das Kelleralphabet Q_{LR} kann groß sein (bis $O(2^{|\dot{P}|})$).

Zu P1: Bei Eingabe $w = a_1 \cdots a_n$ und Suffix $v = a_{j+1} \cdots a_n$ entspricht einem Übergang $(\$ \gamma q q_1 \cdots q_m, v \$) \vdash (\$ \gamma q p, v \$)$ ein Tabelleneintrag (mit festem i) gemäß

$$\frac{i = i_1 < \cdots < i_m = j, \quad X_1 \in F(i_1, i_2), \dots, X_m \in F(i_{m-1}, i_m)}{A \in F(i, j)} \quad (A \rightarrow X_1 \cdots X_m \in P)$$

Man kann mehr Suchaufwand teilen, wenn nur eine Position $i < k < j$ und zwei Felder $F(i, k), F(k, j)$ zu durchsuchen sind.

Die „binäre Simulation“ $\mathcal{A}'_{LR} = (\Sigma, Q'_{LR}, T'_{LR}, q_{in}, q_{fin})$ von \mathcal{A}_{LR} teilt sich Eingabealphabet, Anfangs- und Endzustand mit \mathcal{A}_{LR} , hat

- ▶ $Q'_{LR} := Q_{LR} \cup I_{LR}$, und
- ▶ als T'_{LR} die kleinsten Relationen $\xrightarrow{a}, \xrightarrow{\epsilon} \subseteq Q'_{LR} \times Q'_{LR}$ mit

$$\frac{a \in \Sigma, q \in Q_{LR}, \text{goto}(q, a) = p}{q \xrightarrow{a} qp} \text{ (shift)}$$

$$\frac{q \in Q_{LR}, (A \rightarrow \alpha \cdot) \in cl(q)}{q \xrightarrow{\epsilon} q(A \rightarrow \alpha \cdot)} \text{ (initiate)}$$

$$\frac{q \in Q_{LR}, (A \rightarrow \alpha X \cdot \beta) \in q,}{q(A \rightarrow \alpha X \cdot \beta) \xrightarrow{\epsilon} (A \rightarrow \alpha \cdot X \beta)} \text{ (gather)}$$

$$\frac{q \in Q_{LR}, (A \rightarrow \cdot \alpha) \in cl(q), \text{goto}(q, A) = p}{q(A \rightarrow \cdot \alpha) \xrightarrow{\epsilon} qp} \text{ (move)}$$

\mathcal{A}'_{LR} erkennt w , wenn $(q_{in}, w) \vdash^* (q_{in}q_{fin}, \epsilon)$ mit T'_{LR} .

Lemma: \mathcal{A}_{LR} erkennt $w \in \Sigma^*$ genau dann, wenn \mathcal{A}'_{LR} w erkennt.

Beweis: \Rightarrow : Sei $(q_{in}, w) \vdash^* (q_{in}q_{fin}, \epsilon)$ mit T_{LR} . Simuliere

$$(\delta q_0 q_1 \cdots q_m, v) \vdash_{reduce} (\delta q_0 p, v).$$

Es gibt $(A \rightarrow \cdot X_1 \cdots X_m) \in cl(q_0)$ mit $goto(q_{k-1}, X_k) = q_k$ für $1 \leq k \leq m$ und $goto(q_0, A) = p$. Nach (2) ist, für $m \geq 1$,

$$\begin{aligned} (\delta q_0 \dots q_m, v) &\vdash_{initiate} (\delta q_0 \dots q_m (A \rightarrow X_1 \cdots X_m \cdot), v) \\ &\vdash_{gather} (\delta q_0 \dots q_{m-1} (A \rightarrow X_1 \cdots X_{m-1} \cdot X_m), v) \\ &\quad \vdots \\ &\quad \vdots \\ &\vdash_{gather} (\delta q_0 (A \rightarrow \cdot X_1 \cdots X_m), v) \\ &\vdash_{move} (\delta q_0 p, v). \end{aligned}$$

For $m = 0$ ist $(A \rightarrow \epsilon \cdot) \in cl(q_0)$ und $goto(q_0, A) = p$, also

$$(\delta q_0, v) \vdash_{initiate} (\delta q_0 (A \rightarrow \epsilon \cdot), v) \vdash_{move} (\delta q_0 p, v).$$

\Leftarrow : Zeige, daß $\vdash_{initiate}, \vdash_{gather}, \vdash_{move}$ nur wie eben benutzt werden.

Beispiel: \mathcal{A}'_{LR} zu G

(shift)-Regeln von T'_{LR} : =
 (shift)-Regeln von T_{LR}

(move/goto)-Regeln von T'_{LR} :

(initiate)-Regeln von T'_{LR} :

	1 ($S \rightarrow \cdot NP VP$)	$\xrightarrow{\epsilon}$	1 3
6	$\xrightarrow{\epsilon}$ 6 ($S \rightarrow NP VP \cdot$)	1 ($NP \rightarrow \cdot DET As N$)	$\xrightarrow{\epsilon}$ 1 2
5	$\xrightarrow{\epsilon}$ 5 ($As \rightarrow \epsilon \cdot$)	1 ($NP \rightarrow \cdot PN$)	$\xrightarrow{\epsilon}$ 1 2
10	$\xrightarrow{\epsilon}$ 10 ($VP \rightarrow V NP \cdot$)	2 ($VP \rightarrow \cdot V NP$)	$\xrightarrow{\epsilon}$ 2 6
4	$\xrightarrow{\epsilon}$ 4 ($NP \rightarrow PN \cdot$)	7 ($NP \rightarrow \cdot DET As N$)	$\xrightarrow{\epsilon}$ 7 10
12	$\xrightarrow{\epsilon}$ 12 ($NP \rightarrow DET As N \cdot$)	7 ($NP \rightarrow \cdot PN$)	$\xrightarrow{\epsilon}$ 7 10
11	$\xrightarrow{\epsilon}$ 11 ($As \rightarrow As A \cdot$)	5 ($As \rightarrow \cdot \epsilon$)	$\xrightarrow{\epsilon}$ 5 9
		5 ($As \rightarrow \cdot As A$)	$\xrightarrow{\epsilon}$ 5 9

(gather)-Regeln von T'_{LR} :

$$2 (S \rightarrow NP \cdot VP) \xrightarrow{\epsilon} (S \rightarrow \cdot NP VP)$$

$$3 (S^\dagger \rightarrow \triangleright S \cdot \triangleleft) \xrightarrow{\epsilon} (S^\dagger \rightarrow \triangleright \cdot S \triangleleft)$$

$$7 (VP \rightarrow V \cdot NP) \xrightarrow{\epsilon} (VP \rightarrow \cdot V NP)$$

$$6 (S \rightarrow NP VP \cdot) \xrightarrow{\epsilon} (S \rightarrow NP \cdot VP)$$

$$5 (NP \rightarrow DET \cdot As N) \xrightarrow{\epsilon} (NP \rightarrow \cdot DET As N)$$

$$10 (VP \rightarrow V NP \cdot) \xrightarrow{\epsilon} (VP \rightarrow V \cdot NP)$$

$$4 (NP \rightarrow PN \cdot) \xrightarrow{\epsilon} (NP \rightarrow \cdot PN)$$

$$9 (NP \rightarrow DET As \cdot N) \xrightarrow{\epsilon} (NP \rightarrow DET \cdot As N)$$

$$9 (As \rightarrow As \cdot A) \xrightarrow{\epsilon} (As \rightarrow \cdot As A)$$

$$12 (NP \rightarrow DET As N \cdot) \xrightarrow{\epsilon} (NP \rightarrow DET As \cdot N)$$

$$11 (As \rightarrow As A \cdot) \xrightarrow{\epsilon} (As \rightarrow As \cdot A)$$

Kompakte Form \mathcal{A}_{2LR} von \mathcal{A}'_{LR}

Zu P2 Das Kelleralphabet Q_{LR} kann man verkleinern, denn

1. in $(A \rightarrow \alpha \cdot \beta) \in q$ ist α fürs Erkennen nicht nötig,
2. in $(A \rightarrow \alpha \cdot \beta) \in q$ ist auch A nicht nötig, wenn man es bei (move) mit dem neuen Zustand auf den Stapel schreibt.

Ein **2LR-Item** von G ist ein Suffix (β) einer Regel $(A \rightarrow \alpha\beta) \in P^\dagger$.

$$I_{2LR} := \{ (\beta) \mid (A \rightarrow \alpha\beta) \in P^\dagger \}.$$

Der *Abschluß*' von $q \subseteq I_{2LR}$ ist die kleinste Menge $cl'(q)$, die abgeschlossen ist unter

$$\frac{I \in q}{I \in cl'(q)} (cl'_1), \quad \frac{(A\beta) \in cl'(q), (A \rightarrow \gamma) \in P^\dagger}{(\gamma) \in cl'(q)} (cl'_2)$$

Die Übergangsfunktion $goto' : 2^{I_{2LR}} \times V \rightarrow 2^{I_{2LR}}$ ist gegeben durch

$$goto'(q, X) := \{ (\beta) \mid (X\beta) \in cl'(q) \}, \quad \text{für } X \in V, q \subseteq I_{2LR}.$$

Sei $\mathcal{R}_{2LR} \subseteq 2^{I_{2LR}}$ die kleinste Teilmenge, die abgeschlossen ist unter

$$\frac{\{(S\triangleleft)\} \in \mathcal{R}_{2LR}, \quad q \in \mathcal{R}_{2LR}, X \in V, \text{goto}'(q, X) \neq \emptyset}{\text{goto}'(q, X) \in \mathcal{R}_{2LR}}$$

Der 2LR-Automat $\mathcal{A}_{2LR} = (\Sigma, Q_{2LR}, T_{2LR}, q'_{in}, q'_{fin})$ zu G hat

- ▶ als Eingabealphabet Σ die Terminalsymbole von G ,
- ▶ als Kellularphabet die Menge $Q_{2LR} :=$

$$I_{2LR} \cup \{ (X, q) \in V \times \mathcal{R}_{2LR} \mid q \in \text{Bild}(\text{goto}'(\cdot, X)) \},$$

- ▶ als Anfangszustand $q'_{in} = (\triangleright, \{(S\triangleleft)\})$,
- ▶ als Endzustand $q'_{fin} = (S, \text{goto}'(\{(S\triangleleft)\}, S))$, und

- als Transitionen T_{2LR} die kleinsten unter folgenden Regeln abgeschlossenen Relationen $\mapsto, \mapsto^\epsilon \subseteq Q_{2LR}^{\leq 2} \times Q_{2LR}^{\leq 2}$:

$$\frac{(X, q) \in Q_{2LR}, a \in \Sigma, goto'(q, a) = p}{(X, q) \mapsto^a (X, q)(a, p)} \text{ (shift}_2\text{)}$$

$$\frac{(X, q) \in Q_{2LR}, (\epsilon) \in cl'(q)}{(X, q) \mapsto^\epsilon (X, q)(\epsilon)} \text{ (initiate}_2\text{)}$$

$$\frac{(X, q) \in Q_{2LR}, (\beta) \in q}{(X, q)(\beta) \mapsto^\epsilon (X\beta)} \text{ (gather}_2\text{)}$$

$$\frac{(X, q) \in Q_{2LR}, (\alpha) \in cl'(q), goto'(q, A) = p, (A \rightarrow \alpha) \in P}{(X, q)(\alpha) \mapsto^\epsilon (X, q)(A, p)} \text{ (move}_2\text{)}$$

\mathcal{A}_{2LR} erkennt w , wenn $(q'_{in}, w) \vdash^* (q'_{in}q'_{fin}, \epsilon)$ mit T_{2LR} .

Beachte:

- ▶ T_{2LR} entsteht aus T'_{LR} durch Vereinfachen der Items und Speichern des Übergangssymbols a/A in $shift_2/move_2$.
- ▶ Berechnungen mit \mathcal{A}_{2LR} erfordern i.a. weniger Rücksetzungen als solche mit \mathcal{A}'_{LR} , da Regeln $(A \rightarrow \alpha\beta), (A' \rightarrow \alpha'\beta)$ mit gleichem Suffix β in $gather_2$ simultan behandelt werden.

In \mathcal{A}'_{LR} würde ein reduce/reduce-Konflikt zu getrennten Berechnungen für das Suffix β führen.

Lemma: \mathcal{A}'_{LR} erkennt $w \in \Sigma^*$ genau dann, wenn \mathcal{A}_{2LR} w erkennt.

Beweis: (?) relativ mühsam und subtil.

Folgerung $w \in L(G)$ genau dann, wenn w von \mathcal{A}_{2LR} erkannt wird (oder von \mathcal{A}'_{LR} oder \mathcal{A}_{LR}).

Beispiel: \mathcal{A}_{2LR} zu G

(shift)-Regeln von T_{2LR} :

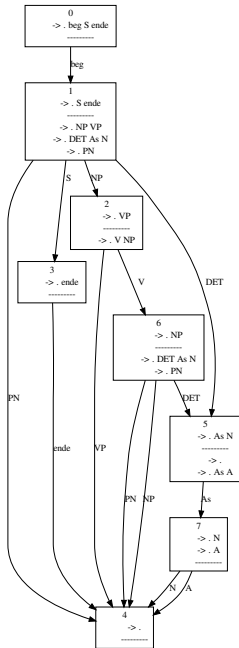
$(\triangleright, 1)$	\xrightarrow{DET}	$(\triangleright, 1) (DET, 5)$
$(V, 6)$	\xrightarrow{DET}	$(V, 6) (DET, 5)$
$(\triangleright, 1)$	\xrightarrow{PN}	$(\triangleright, 1) (P, 4)$
$(V, 6)$	\xrightarrow{PN}	$(V, 6) (PN, 4)$
$(NP, 2)$	\xrightarrow{V}	$(NP, 2) (V, 6)$
$(As, 7)$	\xrightarrow{A}	$(As, 7) (A, 4)$
$(As, 7)$	\xrightarrow{N}	$(As, 7) (N, 4)$

(initiate)-Regeln von T_{2LR} :

$(DET, 5)$	$\xrightarrow{\epsilon}$	$(DET, 5) (\epsilon)$
$(PN, 4)$	$\xrightarrow{\epsilon}$	$(PN, 4) (\epsilon)$
$(VP, 4)$	$\xrightarrow{\epsilon}$	$(VP, 4) (\epsilon)$
$(NP, 4)$	$\xrightarrow{\epsilon}$	$(NP, 4) (\epsilon)$
$(N, 4)$	$\xrightarrow{\epsilon}$	$(N, 4) (\epsilon)$
$(A, 4)$	$\xrightarrow{\epsilon}$	$(A, 4) (\epsilon)$

(gather)-Regeln von T_{2LR} :

$(S, 3)$	(\triangleleft)	$\xrightarrow{\epsilon}$	$(S \triangleleft)$
$(NP, 2)$	(VP)	$\xrightarrow{\epsilon}$	$(NP VP)$
$(V, 6)$	(NP)	$\xrightarrow{\epsilon}$	$(V NP)$
$(DET, 5)$	$(As N)$	$\xrightarrow{\epsilon}$	$(DET As N)$
$(As, 7)$	(N)	$\xrightarrow{\epsilon}$	$(As N)$
$(As, 7)$	(A)	$\xrightarrow{\epsilon}$	$(As A)$
$(PN, 4)$	(ϵ)	$\xrightarrow{\epsilon}$	(PN)
$(VP, 4)$	(ϵ)	$\xrightarrow{\epsilon}$	(VP)
$(NP, 4)$	(ϵ)	$\xrightarrow{\epsilon}$	(NP)
$(N, 4)$	(ϵ)	$\xrightarrow{\epsilon}$	(N)
$(A, 4)$	(ϵ)	$\xrightarrow{\epsilon}$	(A)



(move/goto)-Regeln von T_{2LR} :

$(\triangleright, 1) (NP VP) \xrightarrow{\epsilon} (\triangleright, 1) (S, 3)$

$(\triangleright, 1) (DET As N) \xrightarrow{\epsilon} (\triangleright, 1) (NP, 2)$

$(\triangleright, 1) (PN) \xrightarrow{\epsilon} (\triangleright, 1) (NP, 2)$

$(NP, 2) (V NP) \xrightarrow{\epsilon} (NP, 2) (VP, 4)$

$(V, 6) (DET As N) \xrightarrow{\epsilon} (V, 6) (NP, 4)$

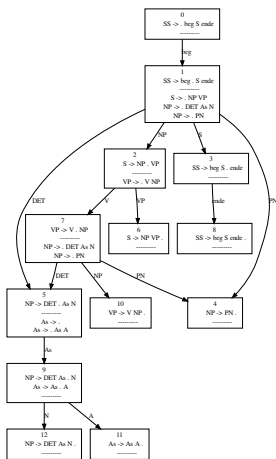
$(V, 6) (PN) \xrightarrow{\epsilon} (V, 6) (NP, 4)$

$(DET, 5) (\epsilon) \xrightarrow{\epsilon} (DET, 5) (As, 7)$

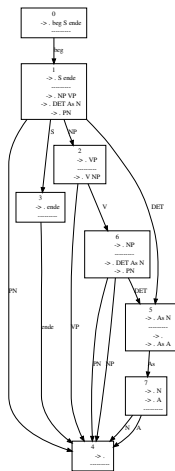
$(DET, 5) (As A) \xrightarrow{\epsilon} (DET, 5) (As, 7)$

Beispiel: LR- versus 2LR-Automat von G

Regeln: $S \rightarrow NP VP$; $NP \rightarrow PN$; $NP \rightarrow DET As N$; $VP \rightarrow V NP$; $As \rightarrow \epsilon$; $As \rightarrow As A$ und implizit: $S^\dagger \rightarrow beg S ende$



Automat mit ML-Yacc-graph



Automat mit ML-GLR

Beispiel: Erkennung von $PN V DET N$

\mathcal{A}_{LR} :	(1, $PN V DET N$)	\mathcal{A}'_{LR} :	(1, $PN V DET N$)
\vdash_{shift}	(1 4, $V DET N$)	\vdash_{shift}	(1 4, $V DET N$)
\vdash_{reduce}	(1 2, $V DET N$)	$\vdash_{initiate}$	(1 4 ($NP \rightarrow PN \cdot$), $V DET N$)
\vdash_{shift}	(1 2 7, $DET N$)	\vdash_{gather}	(1 ($NP \rightarrow \cdot PN$), $V DET N$)
\vdash_{shift}	(1 2 7 5, N)	\vdash_{move}	(1 2, $V DET N$)
\vdash_{reduce}	(1 2 7 5 9, N)	\vdash_{shift}	(1 2 7, $DET N$)
\vdash_{shift}	(1 2 7 5 9 12, ϵ)	\vdash_{shift}	(1 2 7 5, N)
\vdash_{reduce}	(1 2 7 10, ϵ)	$\vdash_{initiate}$	(1 2 7 5 ($As \rightarrow \epsilon \cdot$), N)
\vdash_{reduce}	(1 2 6, ϵ)	\vdash_{move}	(1 2 7 5 9, N)
\vdash_{reduce}	(1 3, ϵ)	\vdash_{shift}	(1 2 7 5 9 12, ϵ)
		$\vdash_{initiate}$	(1 2 7 5 9 12 ($NP \rightarrow DET As N \cdot$), ϵ)
		\vdash_{gather}	(1 2 7 5 9 ($NP \rightarrow DET As \cdot N$), ϵ)
		\vdash_{gather}	(1 2 7 5 ($NP \rightarrow DET \cdot As N$), ϵ)
		\vdash_{gather}	(1 2 7 ($NP \rightarrow \cdot DET As N$), ϵ)
		\vdash_{move}	(1 2 7 10, ϵ) $\vdash \dots \vdash_{move}$ (1 3, ϵ)

A_{2LR} : $((\triangleright, 1), PN\ V\ DET\ N)$
 \vdash_{shift} $((\triangleright, 1) (PN, 4), V\ DET\ N)$
 $\vdash_{initiate}$ $((\triangleright, 1) (PN, 4) (\epsilon), V\ DET\ N)$
 \vdash_{gather} $((\triangleright, 1) (PN), V\ DET\ N)$
 \vdash_{move} $((\triangleright, 1) (NP, 2), V\ DET\ N)$
 \vdash_{shift} $((\triangleright, 1) (NP, 2) (V, 6), DET\ N)$
 \vdash_{shift} $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5), N)$
 $\vdash_{initiate}$ $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5) (\epsilon), N)$
 \vdash_{move} $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5) (As, 7), N)$
 \vdash_{shift} $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5) (As, 7) (N, 4), \epsilon)$
 $\vdash_{initiate}$ $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5) (As, 7) (N, 4) (\epsilon), \epsilon)$
 \vdash_{gather} $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5) (As, 7) (N), \epsilon)$
 \vdash_{gather} $((\triangleright, 1) (NP, 2) (V, 6) (DET, 5) (As\ N), \epsilon)$
 \vdash_{gather} $((\triangleright, 1) (NP, 2) (V, 6) (DET\ As\ N), \epsilon)$
 \vdash_{move} $((\triangleright, 1) (NP, 2) (V, 6) (NP, 4), \epsilon) \vdash \dots$
 \vdash_{move} $((\triangleright, 1) (S, 3), \epsilon)$

Die binäre Überdeckungsgrammatik G_{2LR} von G

Eine Grammatik $G = (N, \Sigma, P, S)$ heißt **binär**, wenn für alle $(A \rightarrow \alpha) \in P$ die Einschränkung $\alpha \in \{\epsilon\} \cup \Sigma \cup N \cup NN$ gilt.

Insbesondere ist jede Grammatik in Chomsky-Normalform binär.

Lemma Zu G kann man in $\mathcal{O}(|G|)$ Schritten ein äquivalentes binäres $G' = (N', \Sigma, S, P')$ berechnen, mit $|G'| = \mathcal{O}(|G|)$.

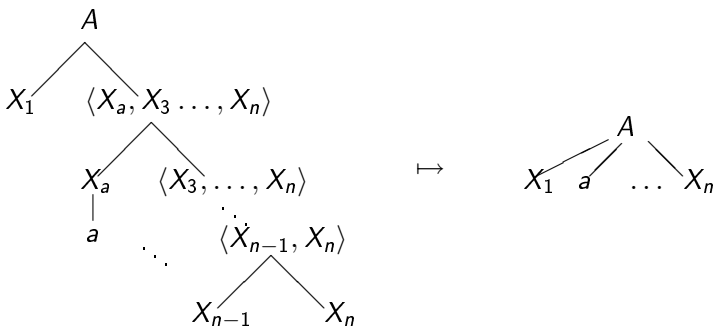
Beweis Ersetze $A \rightarrow \alpha a \beta$ mit $\alpha \beta \neq \epsilon$ durch $A \rightarrow' \alpha X_a \beta$ und $X_a \rightarrow' a$. Dann kürze Suffixe rechter Regelseiten durch neue Nonterminale ab: Ersetze $A \rightarrow X_1 X_2 \cdots X_n$ mit $n > 2$ durch

$$\begin{aligned} A \rightarrow' X_1 \langle X_2, \dots, X_n \rangle, \quad \langle X_2, \dots, X_n \rangle \rightarrow' X_2 \langle X_3, \dots, X_n \rangle, \\ \dots, \\ \langle X_{n-1}, X_n \rangle \rightarrow' X_{n-1} X_n. \end{aligned}$$

Man sieht leicht, daß die Grammatiken äquivalent sind (s.Bild) \square

Bem. Ein $G' \equiv G$ in CNF zu berechnen kostet i.a. $|G'| = \mathcal{O}(|G|^2)$.

Daß G und G' äquivalent sind, zeigt die Baumtransformation:



G' ist eine **Überdeckung** von G : für diese Baumtransformation h gilt, daß für jedes w

$$\{ h(t') \mid t' \text{ ist } G'\text{-Parsebaum von } w \}$$

die Menge der G -Parsebäume von w ist.

Dem binären Kellerautomaten $\mathcal{A}_{2LR} = (\Sigma, Q_{2LR}, T_{2LR}, q'_{in}, q'_{fin})$ von G entspricht eine binäre Überdeckungsgrammatik von G :

Die 2LR-Überdeckung $C_{2LR} = (\Sigma, Q_{2LR}, P_{2LR}, q'_{fin})$ von G hat

- ▶ als Terminale Σ die Terminale von G ,
- ▶ als Nonterminale die Kellersymbole von \mathcal{A}_{2LR} ,
- ▶ als Startsymbol den Endzustand q'_{fin} von \mathcal{A}_{2LR} ,
- ▶ als Regelmenge P_{2LR} die Menge der folgenden Regeln:

$$\begin{aligned}(a, p) &\rightarrow a, && \text{für } a \in \Sigma \text{ und } goto'(q, a) = p, \\(\epsilon) &\rightarrow \epsilon, && \text{für } (\epsilon) \in cl'(q), \\(X\beta) &\rightarrow (X, q)(\beta), && \text{für } (\beta) \in q, (X, q) \in Q_{2LR}, \\(A, p) &\rightarrow (\alpha), && \text{für } (A \rightarrow \alpha) \in P^\dagger \text{ und } goto'(q, A) = p.\end{aligned}$$

C_{2LR} ist nicht in CNF, da es Lösch- und Kettenregeln hat.

Mit $C_{2LR}(G)$ simulieren wir den Kellerautomaten \mathcal{A}_{2LR} : wir parsen bzgl. C_{2LR} mit dem Algorithmus von Cocke-Younger-Kasami (und einem Filter), wobei Zwischenergebnisse in die Parsetabelle gespeichert und wiederverwendet werden, um Rücksetzen zu vermeiden.

Aber: C_{2LR} erzeugt mehr als die Sprache von G : man muß beim Parsen noch Bäume wegfiltern, die keiner Berechnung mit \mathcal{A}_{2LR} entsprechen (d.h. wo die Zustände der (X, q) im Baum nicht „passen“.)

Daß C_{2LR} eine Überdeckung von G ist, muß noch gezeigt werden.

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

2LR-Überdeckungsgrammatik und CYK-Parser

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

17. Juni 2010

Die binäre Überdeckungsgrammatik G_{2LR} von G

Eine Grammatik $G = (N, \Sigma, P, S)$ heißt **binär**, wenn für alle $(A \rightarrow \alpha) \in P$ die Einschränkung $\alpha \in \{\epsilon\} \cup \Sigma \cup N \cup NN$ gilt.

Insbesondere ist jede Grammatik in Chomsky-Normalform binär.

Lemma Zu G kann man in $\mathcal{O}(|G|)$ Schritten ein äquivalentes binäres $G' = (N', \Sigma, S, P')$ berechnen, mit $|G'| = \mathcal{O}(|G|)$.

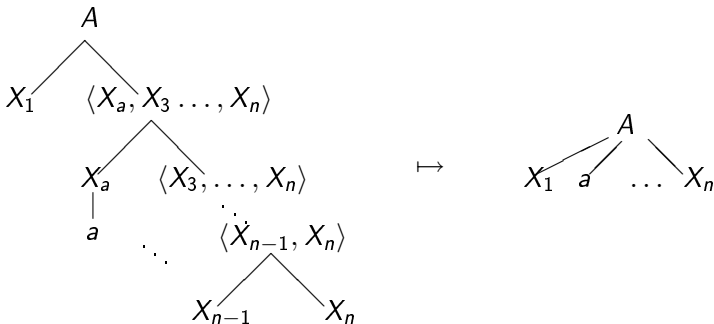
Beweis Ersetze $A \rightarrow \alpha a \beta$ mit $\alpha \beta \neq \epsilon$ durch $A \rightarrow' \alpha X_a \beta$ und $X_a \rightarrow' a$. Dann kürze Suffixe rechter Regelseiten durch neue Nonterminale ab: Ersetze $A \rightarrow X_1 X_2 \cdots X_n$ mit $n > 2$ durch

$$\begin{aligned} A \rightarrow' X_1 \langle X_2, \dots, X_n \rangle, \quad \langle X_2, \dots, X_n \rangle \rightarrow' X_2 \langle X_3, \dots, X_n \rangle, \\ \dots, \\ \langle X_{n-1}, X_n \rangle \rightarrow' X_{n-1} X_n. \end{aligned}$$

Man sieht leicht, daß die Grammatiken äquivalent sind (s.Bild) \square

Bem. Ein $G' \equiv G$ in CNF zu berechnen kostet i.a. $|G'| = \mathcal{O}(|G|^2)$.

Daß G und G' äquivalent sind, zeigt die Baumtransformation:



G' ist eine **Überdeckung** von G : für diese Baumtransformation h gilt, daß für jedes w

$$\{ h(t') \mid t' \text{ ist } G'\text{-Parsebaum von } w \}$$

die Menge der G -Parsebäume von w ist.

Dem binären Kellerautomaten $\mathcal{A}_{2LR} = (\Sigma, Q_{2LR}, T_{2LR}, q'_{in}, q'_{fin})$ von G entspricht eine binäre Überdeckungsgrammatik von G :

Die 2LR-Überdeckung $C_{2LR} = (\Sigma, Q_{2LR}, P_{2LR}, q'_{fin})$ von G hat

- ▶ als Terminale Σ die Terminale von G ,
- ▶ als Nonterminale die Kellersymbole von \mathcal{A}_{2LR} ,
- ▶ als Startsymbol den Endzustand q'_{fin} von \mathcal{A}_{2LR} ,
- ▶ als Regelmenge P_{2LR} die Menge der folgenden Regeln:

$$(a, p) \rightarrow a, \quad \text{für } a \in \Sigma \text{ und } goto'(q, a) = p,$$

$$(\epsilon) \rightarrow \epsilon, \quad \text{für } (\epsilon) \in cl'(q),$$

$$(X\beta) \rightarrow (X, q)(\beta), \quad \text{für } (\beta) \in q, \text{ o. } \beta = \epsilon \in cl'(q), (X, q) \in Q_{2LR},$$

$$(A, p) \rightarrow (\alpha), \quad \text{für } (A \rightarrow \alpha) \in P^\dagger \text{ und } goto'(q, A) = p.$$

C_{2LR} ist nicht in CNF, da es Lösch- und Kettenregeln hat.

Bem: Zur Berechnung von C_{2LR} braucht man nur \mathcal{A}'_G , nicht \mathcal{A}_{2LR} .

Mit $C_{2LR}(G)$ simuliert man den Kellerautomaten \mathcal{A}_{2LR} :

- ▶ zur Erkennung bzgl. C_{2LR} verwendet man den Algorithmus von Cocke-Younger-Kasami (CYK),
- ▶ Zwischenergebnisse werden in der Parsetabelle gespeichert und wiederverwendet, um Rücksetzen/Doppelarbeit zu vermeiden.

Daß C_{2LR} eine Überdeckung von G ist, muß noch gezeigt werden.

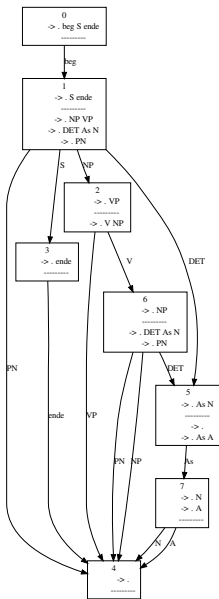
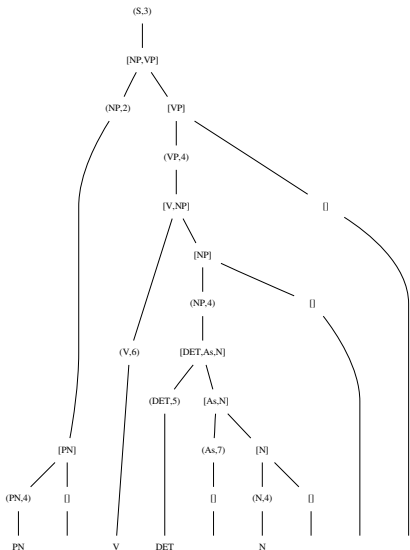
Aber:

- ▶ C_{2LR} erzeugt scheinbare Ambiguitäten; daher:
- ▶ man muß beim Parsen noch Bäume wegfiltern, die keiner Berechnung mit \mathcal{A}_{2LR} entsprechen (d.h. wo die Zustände der (X, q) im Baum nicht „passen“.)

Beispiel: C_{2LR} zu G

$(shift) + (move)$	$(initiate) + (gather)$
$(DET, 5) \rightarrow DET$	$(\epsilon) \rightarrow \epsilon$
$(P, 4) \rightarrow P$	$(S \triangleleft) \rightarrow (S, 3) (\triangleleft)$
$(V, 6) \rightarrow V$	$(NP VP) \rightarrow (NP, 2) (VP)$
$(A, 4) \rightarrow A$	$(V NP) \rightarrow (V, 6) (NP)$
$(N, 4) \rightarrow N$	$(DET A_s N) \rightarrow (DET, 5) (A_s N)$
$(S, 3) \rightarrow (NP VP)$	$(A_s N) \rightarrow (A_s, 7) (N)$
$(NP, 2) \rightarrow (DET A_s N)$	$(A_s A) \rightarrow (A_s, 7) (A)$
$(NP, 2) \rightarrow (PN)$	$(PN) \rightarrow (PN, 4) (\epsilon)$
$(VP, 4) \rightarrow (V NP)$	$(VP) \rightarrow (VP, 4) (\epsilon)$
$(NP, 4) \rightarrow (DET A_s N)$	$(NP) \rightarrow (NP, 4) (\epsilon)$
$(NP, 4) \rightarrow (PN)$	$(N) \rightarrow (N, 4) (\epsilon)$
$(A_s, 7) \rightarrow (\epsilon)$	$(A) \rightarrow (A, 4) (\epsilon)$
$(A_s, 7) \rightarrow (A_s A)$	

Beispiel: Analyse mit C_{2LR}



Erinnerung: Erkener für kontextfreie Grammatiken

Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, mit $S = A_1$,

$$\begin{aligned} A_1 &= \alpha_{1,1} + \dots + \alpha_{1,k_1} =: r_1(A_1, \dots, A_m), \\ &\vdots = \vdots \\ A_m &= \alpha_{m,1} + \dots + \alpha_{m,k_m} = r_m(A_1, \dots, A_m). \end{aligned} \tag{3}$$

- ▶ **Erkennungsproblem** Erkenne in Σ^* die Wörter aus $L(G)$.
Gegeben: G und $w \in \Sigma^*$. Stelle fest, ob $w \in L(G)$ ist.
- ▶ **Analyseproblem** Analysiere die Wörter von $L(G)$ syntaktisch.
Gegeben: G und $w \in L(G)$. Ermittle die Syntaxbäume von w .

Das Erkennungs- oder Wortproblem einer CFG

Idee: Interpretiere G nicht in der KA der Wortmengen über Σ , sondern in der endlichen KA der *Mengen von Wortvorkommen in w* ,

$$\mathcal{T}(w) := (2^{\text{Tok}(w)}, \cup, \emptyset, \cdot, 1^{\mathcal{T}}, *, \langle a^{\mathcal{T}} \rangle_{a \in \Sigma}).$$

Ein *Vorkommen* von $v \in \Sigma^*$ in $w = a_1 \cdots a_n$ ist ein Tripel (i, v, j) mit $v = a_{i+1} \cdots a_j$. Die Operationen auf Tokenmengen sind

- ▶ $0^{\mathcal{T}} := \emptyset$,
- ▶ $1^{\mathcal{T}} = \{ (i, \epsilon, i) \mid 0 \leq i \leq n \}$,
- ▶ $a^{\mathcal{T}} := \{ (i, a, i+1) \mid 0 \leq i < n, a = a_{i+1} \}$,
- ▶ $A + B := A \cup B$,
- ▶ $A \cdot B := \{ (i, uv, j) \mid \exists k ((i, u, k) \in A \wedge (k, v, j) \in B) \}$,
- ▶ A^* ist wie immer der Abschluß von A unter \cdot und 1 .

Da $w = a_1 \dots a_n$ fest ist, kann man ein Token (i, v, j) mit seinen Randpositionen (i, j) identifizieren. Für $M = \{0, 1, \dots, n\}$ ist dann $\mathcal{T}(w)$ eine Unteralgebra der Algebra

$$\mathcal{R} = (2^{M \times M}, \cup, \emptyset, ;, \mathbf{1}_M, *, \langle a^{\mathcal{R}} \rangle_{a \in \Sigma})$$

aller zweistelligen Relationen auf M , wobei

$$a^{\mathcal{R}} = \{ (i, i+1) \mid 0 \leq i < n, a = a_{i+1} \}.$$

Wir berechnen die Stadien der Mengen $A_i^{\mathcal{R}}$ induktiv durch

$$A_{i,0} := \emptyset, \quad A_{i,s+1} := r_i^{\mathcal{R}}(A_{1,s}, \dots, A_{m,s})$$

was nach endlich vielen Schritten in den $A_i^{\mathcal{R}} = A_{i,s+1} = A_{i,s}$ endet.

Erkenner für CNF-Grammatiken (CYK)

Sei $G = (N, \Sigma, P, S)$ eine Grammatik in Chomsky-Normalform, wo die Regeln von der Form $A \rightarrow BC$, $A \rightarrow a$, und $S \rightarrow \epsilon$ sind, S auf keiner rechten Regelseite auftritt, und $A, B, C \in N$, $a \in \Sigma$ sind.

Statt $(i, j) \in A$ schreiben wir „Kanten“ $i \xrightarrow{A} j$. Das nächste Stadium berechnet sich dann aus den vorherigen nach den Regeln

$$\frac{(S \rightarrow \epsilon) \in P}{0 \xrightarrow{S} 0} (\text{Empty}), \quad \frac{i \xrightarrow{a} j, (A \rightarrow a) \in P}{i \xrightarrow{A} j} (\text{Scan})$$

$$\frac{i \xrightarrow{B} k, k \xrightarrow{C} j, (A \rightarrow BC) \in P, i < k < j}{i \xrightarrow{A} j} (\text{Complete})$$

Wenn keine neuen Kanten mehr hinzukommen, gilt:

$$i \xrightarrow{A} j \iff A \Rightarrow^* a_{i+1} \cdots a_j, \quad \text{also: } 0 \xrightarrow{S} n \iff w \in L(G).$$

Darstellung mit einer Parsetabelle („Chart“)

Oft stellt man die Berechnung der „Kanten“ in einer Tabelle

$$F = (F_{i,j})_{0 \leq i < j \leq n}$$

dar, deren Felder $F_{i,j}$, $i < j$, alle „grammatischen Eigenschaften“ (Kategorien) enthalten, die das Teilwort $a_{i+1} \cdots a_j$ hat, d.h.

$$F_{i,j} = \{ A \mid A \Rightarrow^* a_{i+1} \cdots a_j \}.$$

Beachte: $F_{i,j}$ hängt nur von den $F_{i,k}$ und $F_{k,j}$ mit $i < k < j$ ab.

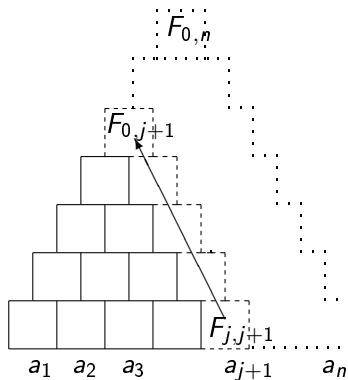
On-line: Hat man schon alle $F_{i',j'}$ mit $i' < j' \leq j$, dann berechnet man zuerst die Eigenschaften des nächsten Worts,

$$F_{j,j+1} := \{ A \mid (A \rightarrow a_{j+1}) \in P \},$$

und danach die Eigenschaften wachsender Suffixe von $a_1 \cdots a_{j+1}$:

$$F_{i,j+1} := \{ A \mid (A \rightarrow BC) \in P, i < k < j+1, B \in F_{i,k}, C \in F_{k,j+1} \}$$

Nach Lesen von $a_1 \cdots a_j$ werden alle Felder $F_{i,k}$ mit $0 \leq i < k \leq j$ gefüllt. Dann die Felder $F_{j,j+1}, F_{j-1,j+1}, \dots, F_{0,j+1}$:



Komplexität: Wegen CNF kann $F_{i,j}$ in $O(|j - i| \cdot |P| \cdot |N|)$ und damit F in $O(|w|^3 \cdot |G|^2)$ Schritten berechnet werden; mit $F_{i,j}$ als Bitvektor der Länge $|N|$ in $O(|w|^3 \cdot |G|)$ Schritten.

Aber: Die Umwandlung von G in CNF ergibt $|CNF(G)| = O(|G|^2)$.

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

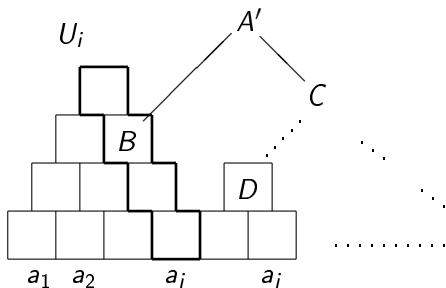
CYK-Parser mit LR-Filterung

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

17. Juni 2010

LR-Filter für CYK mit CNFs

Kennt man alle Eigenschaften B der Suffixe von $a_1 \cdots a_i$, so kann man die mit ihnen nicht kombinierbaren Eigenschaften D von Präfixen von $a_{i+1} \cdots a_n$ wegfiltern, d.h. gar nicht erst eintragen:



$$\text{Filter: } D \in F_{i,j} \Rightarrow (i \cdot D) := \exists (A' \rightarrow BC) \in P (B \in U_i \wedge D \text{ lc}^* C)$$

$$\text{lc} := \{ (X, C) \in V \times N \mid \exists (C \rightarrow \gamma X \delta) \in P \gamma \Rightarrow^* \epsilon \}$$

CYK für binäre CFGs mit LR-Filter

Sei $G' = (\Sigma, N', P', S')$ eine binäre CFG, $predict : 2^{N'} \rightarrow 2^{N'}$, $A_{init} \in N'$, und $w = a_1 \cdots a_n \in \Sigma^+$.

Die Parsetabelle F zu w enthält für $0 \leq i \leq j \leq n$ eine Menge $F(i, j) \subseteq N'$. Schreibe $A \in F(i, j)$ als „Kante“ $i \xrightarrow{A} j$. F ist die kleinste Menge von Kanten, die abgeschlossen ist unter:

$$\frac{}{0 \xrightarrow{A_{init}} 0} \text{ (start)} \quad \frac{(A \rightarrow \epsilon) \in P', \quad i \cdot A}{i \xrightarrow{A} i} \text{ (empty)}$$

$$\frac{i < n, \quad (A \rightarrow a_{i+1}) \in P', \quad i \cdot A}{i \xrightarrow{A} i+1} \text{ (scan)}$$

$$\frac{i \xrightarrow{B} j, \quad (A \rightarrow B) \in P', \quad i \cdot A}{i \xrightarrow{A} j} \text{ (close)}$$

$$\frac{i \xrightarrow{B} k, \quad k \xrightarrow{C} j, \quad (A \rightarrow BC) \in P', \quad i \cdot A, \quad i \leq k \leq j}{i \xrightarrow{A} j} \text{ (complete)}$$

Die „Filter“-Bedingung $i \cdot A$ wird beim Parsen berechnet, mit

$$\frac{A \in \text{predict}(\{ B \mid i' \xrightarrow{B} i \})}{i \cdot A} \text{ (predict)}.$$

Wegen Lösch- und Kettenregeln fehlt $i < k < j$ in (*combine*), und man braucht Felder $F(i, j)$ mit $0 \leq i \leq j \leq n$ (nicht bei G in CNF).

Dieser CYK ist korrekt (für nicht-leere Eingaben), und wenn (*predict*) „nicht zu viel wegfiltert“ auch vollständig; reichen sollte:

$$B \in U \wedge (A' \rightarrow BC) \in P' \wedge A \text{ lc}^* C \implies A \in \text{predict}(U)$$

wobei $\text{lc} := \{ (X, C) \mid (C \rightarrow \gamma X \delta) \in P, \gamma \Rightarrow^* \epsilon \}$ („Leftcorner“).

Beh: Wenn A_{init} nur in der Startregel $S' \rightarrow A_{init} S$ vorkommt, ist

$$a_1 \cdots a_n \in L(S) \iff 0 \xrightarrow{S} n \in F.$$

\mathcal{A}_G	T_{2LR}	C_{2LR}	$\text{predict}(U)$
$q \xrightarrow{a} q'$	$(X, q) \xrightarrow{a} (X, q)(a, q')$	$(a, q') \rightarrow a$	$(a, q')^*$
$\epsilon \in \text{cl}(q)$	$(X, q) \xrightarrow{\epsilon} (X, q)(\epsilon)$	$(\epsilon) \rightarrow \epsilon$	$(\epsilon)^*$
$\beta \in q$	$(X, q)(\beta) \xrightarrow{\epsilon} (X\beta)$	$(X\beta) \rightarrow (X, q)(\beta)$	$(X\beta)$
$q \xrightarrow{A} q'$	$(X, q)(\alpha) \xrightarrow{\epsilon} (X, q)(A, q')$	$(A, q') \rightarrow (\alpha)$	$(A, q')^+$

*): wenn $(X, q) \in U$, +): wenn $(X, q) \in U$ und $(A \rightarrow \alpha) \in P$

$\text{predict}(U)$

$:= \{ (a, q') \mid (X, q) \xrightarrow{a} (X, q)(a, q') \in T_{2LR}, (X, q) \in U \}$

$\cup \{ (\epsilon) \mid (X, q) \xrightarrow{\epsilon} (X, q)(\epsilon) \in T_{2LR} \}$

$\cup \{ (X\beta) \mid (X, q)(\beta) \xrightarrow{\epsilon} (X\beta) \in T_{2LR} \}$

$\cup \{ (A, q') \mid (X, q)(\alpha) \xrightarrow{\epsilon} (X, q)(A, q') \in T_{2LR}, (X, q) \in U \}$

Beispiel mit ML-GLR

Mit `ml-qlr <spec>.grm -debug` oder wie unten kann man die Erzeugung der Überdeckungsgrammatik inspizieren:

```
- CM.make "src/ml-qlr.cm";  
- ParseGen.parseGen "explMiniGer.grm" true; (* =: debug *)  
ParseGenParser.parse: parsing the grammar specification ...  
ParseGen.make_parser: computing the binary LR-cover-grammar
```

source grammar:

```
grammar (start:= <S'>, 8 nonterms, 8 terms) =  
(rule 0)      S : NP VP  
(rule 1)      NP : PN  
(rule 2)      NP : DET As N  
(rule 3)      VP : V NP  
(rule 4)      As : As A  
(rule 5)      As :  
(rule 6)      <S'> : beg S ende
```

compact	state 1	state 2	state 4
LR(0)-graph:	kernel:	kernel:	kernel:
	. S ende	. VP	.
state 0	non-kernel:	non-kernel:	non-kernel:
kernel:	. NP VP	. V NP	
. beg S ende	. DET As N		goto:
non-kernel:	. PN	goto:	
		2 -V-> 6	state 5
goto:	goto:	2 -VP-> 4	kernel:
0 -beg-> 1	1 -DET-> 5		. As N
	1 -PN-> 4	state 3	non-kernel:
	1 -S-> 3	kernel:	.
	1 -NP-> 2	. ende	. As A
		non-kernel:	
			goto:
		goto:	5 -As-> 7
		3 -ende-> 4	

		terminals:	8 (PN,4)
state 6	state 7	0 DET	9 (DET,5)
kernel:	kernel:	1 N	10 (V,6)
. NP	. N	2 PN	11 [NP]
non-kernel:	. A	3 V	12 [As,N]
. DET As N	non-kernel:	4 A	13 [N]
. PN		5 PUNKT	14 [A]
	goto:	6 beg	15 (NP,2)
goto:	7 -N-> 4	7 ende	16 (S,3)
6 -DET-> 5	7 -A-> 4	nonterminals:	17 [NP,VP]
6 -PN-> 4		0 [beg,S,ende]	18 (VP,4)
6 -NP-> 4		1 (beg,1)	19 [V,NP]
		2 [S,ende]	20 []
		3 (A,4)	21 (As,7)
		4 (N,4)	22 [As,A]
		5 [ende]	23 [DET,As,N]
		6 (ende,4)	24 (NP,4)
		7 [VP]	25 [PN]

cover grammar:

grammar (start:= (S,3), 26 nonterms, 8 terms) =

(rule 0) (PN,4) : PN
(rule 1) (N,4) : N
(rule 2) (A,4) : A
(rule 3) (DET,5) : DET
(rule 4) (V,6) : V
(rule 5) [] :
(rule 6) [beg,S,ende] : (beg,1) [S,ende]
(rule 7) [NP,VP] : (NP,2) [VP]
(rule 8) [S,ende] : (S,3) [ende]
(rule 9) [A] : (A,4) []
(rule 10) [N] : (N,4) []
(rule 11) [NP] : (NP,4) []
(rule 12) [ende] : (ende,4) []

(rule 13) [VP] : (VP,4) []
(rule 14) [PN] : (PN,4) []
(rule 15) [DET,As,N] : (DET,5) [As,N]
(rule 16) [V,NP] : (V,6) [NP]
(rule 17) [As,N] : (As,7) [N]
(rule 18) [As,A] : (As,7) [A]
(rule 19) (NP,2) : [DET,As,N]
(rule 20) (NP,2) : [PN]
(rule 21) (S,3) : [NP,VP]
(rule 22) (VP,4) : [V,NP]
(rule 23) (As,7) : []
(rule 24) (As,7) : [As,A]
(rule 25) (NP,4) : [DET,As,N]
(rule 26) (NP,4) : [PN]

Warning: noshift declarations ignored

lrfilter:	((_,4), [])
((_,0), (beg,1))	((_,5), [])
((_,1), (DET,5))	(_, [beg,S,ende])
((_,1), (PN,4))	(_, [NP,VP])
((_,1), (S,3))	(_, [S,ende])
((_,1), (NP,2))	(_, [A])
((_,2), (V,6))	(_, [N])
((_,2), (VP,4))	(_, [NP])
((_,3), (ende,4))	(_, [ende])
((_,5), (As,7))	(_, [VP])
((_,6), (DET,5))	(_, [PN])
((_,6), (PN,4))	(_, [DET,As,N])
((_,6), (NP,4))	(_, [V,NP])
((_,7), (N,4))	(_, [As,N])
((_,7), (A,4))	(_, [As,A])

Beispiel: Aufbauen der Parsetabelle

```
- CM.make "explMiniGer.cm";  
- Cnf.debug := true;  
- Cnf.parse();
```

Lisa frisst das Gras.

...

```
field(0,0): (beg,1){0,0}
```

```
field(1,1): []{1,1}
```

```
field(0,1): PN.1
```

```
    (PN,4){0,1} -> PN.1
```

```
    [PN]{0,1} -> (PN,4){0,1}    []{1,1}
```

```
    (NP,2){0,1} -> [PN]{0,1}
```

```
field(2,2):
```

```
field(1,2): V.1
```

```
    (V,6){1,2} -> V.1
```

```
field(0,2):
```

```
field(3,3): []{3,3}
```

```
    (As,7){3,3} -> []{3,3}
```

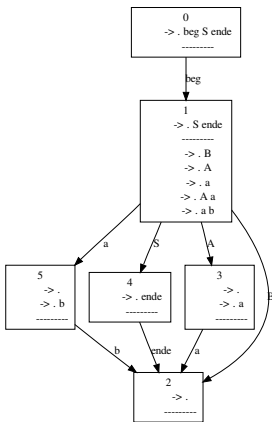
field(2,3): DET.1
 (DET,5){2,3} -> DET.1
 field(1,3):
 field(0,3):
 field(4,4): []{4,4}
 field(3,4): N.1
 (N,4){3,4} -> N.1
 [N]{3,4} -> (N,4){3,4} []{4,4}
 [As,N]{3,4} -> (As,7){3,3} [N]{3,4}
 field(2,4): [DET,As,N]{2,4} -> (DET,5){2,3} [As,N]{3,4}
 (NP,4){2,4} -> [DET,As,N]{2,4}
 [NP]{2,4} -> (NP,4){2,4} []{4,4}
 field(1,4): [V,NP]{1,4} -> (V,6){1,2} [NP]{2,4}
 (VP,4){1,4} -> [V,NP]{1,4}
 [VP]{1,4} -> (VP,4){1,4} []{4,4}
 field(0,4): [NP,VP]{0,4} -> (NP,2){0,1} [VP]{1,4}
 (S,3){0,4} -> [NP,VP]{0,4}

Wegfiltern scheinbarer Analysen

Die Grammatik G aus `examples/cmpl/lrf.cm` mit den Regeln

$$S \rightarrow A \mid B \quad A \rightarrow a \mid A a \quad B \rightarrow a b$$

hat folgenden kompakten endlichen LR(0)-Automaten \mathcal{A}'_G :



Die Überdeckungsgrammatik C_{2LR} von G ist

grammar (start:= (S,4), 18 nonterms, 3 terms) =

(rule 0) (a,2) : a

(rule 1) (b,2) : b

(rule 2) (a,5) : a

(rule 3) [] :

(rule 4) [beg,S,ende] : (beg,1) [S,ende]

(rule 5) [b] : (b,2) []

(rule 6) [ende] : (ende,2) []

(rule 7) [a] : (a,2) []

(rule 8) [B] : (B,2) []

(rule 9) [A] : (A,3) []

(rule 10) [A,a] : (A,3) [a]

(rule 11) [S,ende] : (S,4) [ende]

(rule 12) [a] : (a,5) []

(rule 13) [a,b] : (a,5) [b]

(rule 14) (B,2) : [a,b]

(rule 15) (A,3) : [a]

(rule 16) (A,3) : [A,a]

(rule 17) (S,4) : [B]

(rule 18) (S,4) : [A]

Die Eingabe *aa* erzeugt *ohne* den LR-Filter folgende Tabelle:

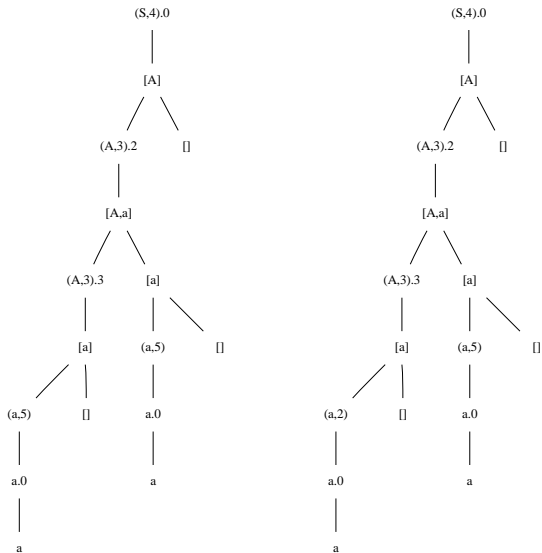
```
field(0,0): (beg,1){0,0}
             []{0,0}
field(1,1): []{1,1}
field(0,1): a.1
             (a,2){0,1} -> a.1
             (a,5){0,1} -> a.1
             [a]{0,1} -> (a,2){0,1}   []{1,1}
             [a]{0,1} -> (a,5){0,1}   []{1,1}
             (A,3){0,1} -> [a]{0,1}
             [A]{0,1} -> (A,3){0,1}   []{1,1}
             (S,4){0,1} -> [A]{0,1}
```

```

field(2,2): []{2,2}
field(1,2): a.1
            (a,2){1,2} -> a.1
            (a,5){1,2} -> a.1
            [a]{1,2} -> (a,2){1,2}   []{2,2}
            [a]{1,2} -> (a,5){1,2}   []{2,2}
            (A,3){1,2} -> [a]{1,2}
            [A]{1,2} -> (A,3){1,2}   []{2,2}
            (S,4){1,2} -> [A]{1,2}
field(0,2): [A,a]{0,2} -> (A,3){0,1}  [a]{1,2}
            (A,3){0,2} -> [A,a]{0,2}
            [A]{0,2} -> (A,3){0,2}   []{2,2}
            (S,4){0,2} -> [A]{0,2}

```

Ohne den LR-filter hat die Eingabe 4 C_{2LR} -Bäume, darunter:



Der LR-Filter ist

$(_ , [beg, S, ende])$	$((a, 2), [])$	$(_ , [a, b])$
$((a, 5), (b, 2))$	$((a, 5), [])$	$(_ , [a])$
$((S, 4), (ende, 2))$	$((B, 2), [])$	$((beg, 1), (A, 3))$
$((A, 3), (a, 2))$	$((A, 3), [])$	$(_ , [A, a])$
$(_ , [S, ende])$	$((beg, 1), (a, 5))$	$(_ , [B])$
$(_ , [ende])$	$(_ , [b])$	$((beg, 1), (S, 4))$
$((b, 2), [])$	$((beg, 1), (B, 2))$	$(_ , [A])$
$((ende, 2), [])$		

Mit dem LR-Filter \mathcal{F} wird die Tabelle kleiner:

- ▶ Feld (0,0) enthält $[]$ nicht mehr, da $((beg, 1), []) \notin \mathcal{F}$,
- ▶ Feld (0,1) enthält die Einträge mit Wurzel $(a, 2)$ und davon abhängende Einträge nicht mehr, da $((beg, 1), (a, 2)) \notin \mathcal{F}$;
- ▶ Feld (1,2) enthält die Einträge mit Wurzel $(a, 5)$ nicht mehr, da $((A, 3), (a, 5)) \notin \mathcal{F}$.

```

field(0,0): (beg,1){0,0}
field(1,1): []{1,1}
field(0,1): a.1
            (a,5){0,1} -> a.1
            [a]{0,1} -> (a,5){0,1}   []{1,1}
            (A,3){0,1} -> [a]{0,1}
            [A]{0,1} -> (A,3){0,1}   []{1,1}
            (S,4){0,1} -> [A]{0,1}
field(2,2): []{2,2}
field(1,2): a.1
            (a,2){1,2} -> a.1
            [a]{1,2} -> (a,2){1,2}   []{2,2}
field(0,2): [A,a]{0,2} -> (A,3){0,1}   [a]{1,2}
            (A,3){0,2} -> [A,a]{0,2}
            [A]{0,2} -> (A,3){0,2}   []{2,2}
            (S,4){0,2} -> [A]{0,2}

```

Die Folge ist, daß von den 4 ohne Filter möglichen Bäumen nur der eine übrigbleibt, dem eine Berechnung mit A_{2LR} entspricht.

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

Der Parsergenerator ML-GLR

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

24. Juni 2010

Parsergenerator ML-GLR

ML-GLR ist ein Parsergenerator, der

- ▶ Parser für azyklische kontextfreie Grammatiken erzeugt,
- ▶ die Eingabegrammatik in die 2LR-Überdeckung umwandelt und einen CYK-Tabellenparser mit LR-Filter verwendet,
- ▶ durch Änderung des Quellcodes von ML-Yacc entstanden ist,
- ▶ Operatordeklarationen und semantische Attribute unterstützt,
- ▶ lexikalische, syntaktische und semantische Ambiguität erlaubt,
- ▶ das Zusammenbauen von Lexer und Parser selbständig macht.

Teilweise realisiert und geplant sind:

- ▶ Übertragung der Fehlerkorrektur von ML-Yacc auf ML-GLR
- ▶ Ersetzen der 2LR- durch eine binäre Überdeckung und effizienteren CYK mit LR-Filter und Erreichbarkeitsrelationen
- ▶ Parsezeit von $O(|G| \cdot |w|^3)$ in der Quellgrammatik G

Bedienung von ML-GLR

Die Arbeitsweise von ML-GLR ist ähnlich wie die von ML-Yacc:

1. aus der Grammatikspezifikation `G.grm` werden der kompakte LR(0)-Automat \mathcal{A}'_G , die Überdeckungsgrammatik G_{2LR} , die Token und semantischen Aktionen berechnet, in `G.grm.sml`,
2. aus der Lexerspezifikation `G.lex` wird ein Lexer berechnet und nach `G.lex.sml` geschrieben,
3. eine Bibliothek wird benutzt, die einen CYK-Parser und Funktor zum Zusammenbauen der Teile enthält,
4. die Verbindung von Lexer und Grammatik/Parser wird in `G.grm.lnk.sml` gespeichert.

Am einfachsten steuert man das Erstellen des Parsers durch eine „Make“-Datei `G.cm` für den Compilation Manager von `smlnj`:

<'Makefile' G.cm für die Grammatik G>≡

Group

structure G (* Grammatikname: G *)

is

\$/basis.cm

\$/ml-qlr-lib.cm

G.lex

G.grm:MLGLR

G.parser.sml (* Parsen von der Konsole *)

Auflösung der Anker `$/...` und Dateisuffixe `.lex|grm|qlr` mit

<~/smlnj-pathconfig>≡

ml-qlr-lib.cm <absoluter Pfad zu>/hl-yacc/lib

mlqlr-tool.cm <absoluter Pfad zu>/hl-yacc/tool

qlr-ext.cm <absoluter Pfad zu>/hl-yacc/tool

ML-GLR erlaubt Grammatikspezifikationen, die zwei Parserdeklarationen mehr haben als die von ML-Yacc:

- ▶ die Deklaration `%graphview`, durch die eine `.dot`-Datei des kompakten LR(0)-Automaten erstellt, in `.ps` umgewandelt und mit `gv` angezeigt wird,
- ▶ die Deklaration

```
%lexheader (functor GLexFun(structure Tokens:G_TOKENS  
                                ... ) : LEXER)
```

die die `%header`-Deklaration aus der *Lexerspezifikation* `G.lex` enthält, und mit der die Datei `G.grm.lnk.sml` erzeugt wird.

ML-GLR arbeitet mit einem ML-Lex, dessen Matchregeln die Form

```
... => (...; ([Tokens.term(value, pos, pos'), ...], yytext));
```

haben, mit dem erkannten Text `yytext` und einer *Liste* seiner Lesarten, d.h. Token mit verschiedenen `term` und `value`.

Grammatikspezifikation:

$\langle G.grm \rangle \equiv$

```
%% (* Parser declarations: *)
```

```
%name      G
```

```
%graphview
```

```
%pos       int
```

```
%lexheader (
```

```
functor GLexFun(structure Tokens: G_TOKENS) : LEXER)
```

```
%term      a | EOF
```

```
%nonterm   S | A | B
```

```
%eop       EOF
```

```
%% (* Grammar rules: *)
```

```
S : A A ( ) | B A ( )
```

```
A : a ( )
```

```
B : a ( )
```

Lexerspezifikation:

$\langle G.lex \rangle \equiv$

```
type pos = int
type ('a,'b) token = ('a,'b) Tokens.token
type svalue = Tokens.svalue
type lexresult = (svalue,pos) token list * string
fun eof() : lexresult = ([Tokens.EOF(0,0)],"")
fun error(msg,lin,col) =
    TextIO.output(TextIO.stdout,"GLex: "^ msg ^"\n")
val lin = ref 0
%%
%header
(functor GLexFun(structure Tokens: G_TOKENS):LEXER);
%%
"\n"=> (lin := (!lin + 1); lex());
"a" => ([Tokens.a(!lin,!lin)],yytext);
"." => (YYBEGIN INITIAL; eof());
.   => (error ("char "^yytext^" skipped",!lin,!lin));
```

Bedienung von der Konsole:

1. `> ml-lex G.lex` produziert `G.lex.sml`, was den Funktor
functor `GLexFun(structure Tokens:G_TOKENS):LEXER`
enthält, der aus den an `G` abzulesenden `Tokens:G_TOKENS` und
einem `reader:int->string` einen passenden Lexer baut;
`LEXER` verlangt eine Funktion

```
makeLexer : (int -> string) -> unit -> lexresult
```

2. `> ml-qlr G.grm` produziert drei Dateien:
 - ▶ `G.grm.sig` mit den zwei Signaturen: `G_TOKENS` und `G_VALS`,
 - ▶ `G.grm.sml` mit dem Funktor

```
GValsFun : TOKEN * GRAMMAR -> G_VALS
```
 - ▶ `G.grm.lnk.sml` mit drei Strukturen (in Pseudocode:)

```
GVals = GValsFun(CharParser.Token+Grammar)
GLex = GLexFun(GVals.Tokens:G_TOKENS)
GParser = Join(GVals.ParserData,GLex,CharParser)
```

Erzeugte Datei G.grm.sig:

```
signature G_TOKENS =
  sig
    type ('a,'b) token      (* abstrakte Typen *)
    type svalue
    (* Typen der Tokenkonstruktionsfunktionen: *)
    val EOF: 'a * 'a -> (svalue,'a) token
    val a: 'a * 'a -> (svalue,'a) token
  end

signature G_VALS =
  sig
    structure Tokens : G_TOKENS
    structure ParserData : PARSER_DATA
    sharing type ParserData.Token.token = Tokens.token
    sharing type ParserData.svalue = Tokens.svalue
  end
```

ParserData sammelt aus G die zum Parsen nötige Information:

<Auszug aus hl-yacc/lib/base.sig>≡

```
signature PARSE_DATA =
  sig
    type svalue  (* the type of semantic values *)
    type result  (* type of the result of a parse *)
    ...
    val spec : string          (* source grammar spec *)
    val grammar : Grammar.grammar (* cover grammar *)
    val lrfilter : Grammar.nonterm * Grammar.nonterm -> bool
    ...
    structure Actions : sig
      val void : svalue          (* default value *)
      val actions : (* semantic of source grammar rules *)
        int * pos * (svalue * pos * pos) list * arg ->
        Grammar.nonterm * (svalue * pos * pos)
      val extract : svalue -> result  (* from tree top *)
    end
  end
```

Join bildet aus GVals.ParserData und GLex einen Parser mit

<Auszug aus hl-acc/lib/base.sig>≡

```
signature PARSER =
  sig
    structure Token : TOKEN
    val sameToken : (svalue,pos) Token.token * (svalue,pos)
    structure Stream : STREAM
    ...
    type svalue      (* of semantic values at nodes of parse
    type lexresult = (svalue,pos) Token.token list * string
    val makeLexer : (int -> string) -> lexresult Stream.str

    val parse : (lexresult Stream.stream) *
                (string * pos * pos -> unit) * arg
                -> result list * (lexresult Stream.stream)
    val debug : bool ref
    val trees : bool ref
    val covertrees : bool ref
  end
```

CM-Makefile:

$\langle G.cm \rangle \equiv$

Group

structure GParser

is

\$/basis.cm

\$/ml-glr-lib.cm

G.lex

G.grm:MLGLR

$\langle \textit{Erzeugung des GParser interaktiv} \rangle \equiv$

> smlnj

- CM.make "G.cm";

...

- open GParser;

...

⟨Erzeugung des GParser von der Konsole⟩≡

```
> sml G.cm
...
[ml-qlr  G.grm]
[parsing (G.cm):G.grm.sig]
[parsing (G.cm):G.grm.lnk.sml]
[parsing (G.cm):G.grm.sml]
- open GParser;
...
val progress : bool ref
```

Mit `GParser.parse` kann man erst parsen, wenn man ihm passende Argumentfunktionen `reader`, `error` u.a. gibt.

Das wird in den `examples/grm/*`-Beispielen mit einer Datei `parser.sml` und einer Hilfsdatei `interface.sml` gemacht, die in der `G.cm` einzutragen sind (mit passendem Namen `G!`).

Ohne interface.sml und den Funktor in parse.sml geht es so:

$\langle G.parser.sml \rangle \equiv$

```
structure G =
```

```
  struct
```

```
    structure Parser = GParser
```

```
    structure Tokens = GVals.Tokens
```

```
  fun error (msg,lin,_) = () (* keine Fehlermeldung *)
```

```
  fun loop lexer =
```

```
    let val (result,lexer) =
```

```
        Parser.parse(lexer,error,())
```

```
        val ((nextTokens,nextWord),lexer) =
```

```
            Parser.Stream.get lexer
```

```
    in
```

```
      if Parser.sameToken(List.hd nextTokens,  
                           Tokens.EOF(0,0))
```

```
      then result else loop lexer
```

```
    end
```

```

⟨G.parser.sml⟩+≡
  fun reader (n:int) =
    case (TextIO.inputLine TextIO.stdIn) of
      SOME s => s
    | NONE => raise Undefined

  fun parse () = loop (Parser.makeLexer reader)

  val debug = Parser.debug
  val trees = Parser.trees
  val covertrees = Parser.covertrees
end

```

Wenn G.cm diese Datei G.parser.sml nennt und die Struktur G exportiert, kann man mit G.parse von der Konsole lesen:

```

⟨Beispiel nach CM.make "G.cm" ⟩≡
  - G.trees:=true; G.parse();
  - abac. (* ignoriert b,c und erkennt aa *)

```

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

Einblick in den Quellcode von ML-GLR

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

31. Juni 2010

Quellcode von ML-GLR

Dateistruktur: (siehe `hl-yacc/src/FILES.GLR`)

1. Bibliotheksdateien, die alle von ML-GLR erzeugten Parser brauchen und durch `lib/ml-glr-lib.cm` geladen werden:

`<hl-yacc/>≡`

Chart parser for 2LR-cover grammars:

`lib/grammar.sml`

`lib/chart.sml`

`lib/chartparser.sml`

Joining the chartparser with the parser data of the source grammar and the lexer fitting to the source grammar's terminals:

`lib/base.sig`

`lib/join.sml`

`lib/stream.sml`

2. Quelldateien des Parsergenerators:

`<hl-yacc/src/ >≡`

`parseGenParser-util.cm`

`parseGenParser.sig`

`parseGenParser.sml`

Parser for parsing the source grammar (with ML-Yacc)

`hdr.sml`

`yacc.grm`

`yacc.lex.sml`

`parse.sml`

Signatures for intermediate structures:

`utils.sig`

`utils.sml`

`sigs.sml` (PARSE_GEN, INTGRAMMAR, CORE, LRGRAPH ..

`<hl-yacc/src/>+≡`

Base definitions and acyclicity test:

`../lib/grammar.sml`

`look.sml (auch: nullable, leftcorner, ..)`

Compacted LR(0)-graph generator:

`core.sml`

`coreutils.sml`

`graph.sml`

2LR-cover grammar generator:

`mkcover.sml`

Rest of ML-GLR:

- to handle abstract syntax for actions and remove unused variable bindings from the abstract syntax (from ML-Yacc)

`absyn.sig`

`absyn.sml`

`<hl-yacc/src/>+≡`

- to check source grammar specification for errors, create the source grammar from specification (by ML-Yacc) print `<grm>.sig|sml|lnk` files and semantic actions for the parser:
 `glr.sml`
- module to hook everything together:
 `link.sml`
- exporting `parseGen`, `parseGen2`:
 `export-glr.sml`
- printing the source grammar, its compact LR(0) automaton, its 2LR-cover grammar, and errors in the source grammar to `<grm>.descr`:
 `verbose.sml`

Einblick in einige Programmteile

- ▶ `graph.sml`
baut den kompakten LR(0)-Graphen der Grammatik
- ▶ `mkcover.sml`
baut die 2LR-Überdeckungsgrammatik aus dem Graphen
- ▶ `chartparser.sml`
ein Chartparser für 2LR-Überdeckungsgrammatiken

```

<aus src/graph.sml>≡
  functor mkGraph(structure IntGrammar : INTGRAMMAR
                  structure Core : CORE ... ): LRGRAPH =
  struct
    ...
    type graph = {edges: {edge:symbol,to:core} list array,
                  nodes: core list,
                  nodeArray : core array}
    val shift =
      fn ({edges,...} : graph) => fn a as (i,sym) =>
      let fun find nil = raise (Shift a)
          | find ({edge,to=CORE (_,state)} :: r) =
            if gtSymbol(sym,edge) then find r
            else if eqSymbol(edge,sym) then state
                else raise (Shift a)
          in find (edges sub i)
      end
  end

```

```

<aus src/graph.sml>+≡
  val mkGraph =
    fn (g as (GRAMMAR {start,...})) =>
      let val {shifts,produces,rules,epsProds,nonKernel,
              findRuleToLhs,...} = CoreUtils.mkFuncs g
          fun f (nodes,node_list,edge_list,nil,nil,num) =
              let val nodes=rev node_list
                  in {nodes=nodes,
                      edges=Array.fromList (rev edge_list),
                      nodeArray = Array.fromList nodes} end
          | f (nodes,node_list,edge_list,nil,y,num) =
              f (nodes,node_list,edge_list,rev y,nil,num)
          | f (nodes,node_list,edge_list,h::t,y,num) =
              let val (nodes,edges,future,num) =
                  List.foldr add_goto (nodes,[],y,num)
                      (shifts h)
                  in f (nodes,h::node_list,
                      edges::edge_list,t,future,num)
              end
      end
end

```

```

<aus src/graph.sml>+≡
  in
    {graph = let val makeItem = fn (RULE {rhs,...}) =>
              ITEM{rhsAfter=rhs}
              val initialItemList =
                map makeItem (produces start)
              val orderedItemList = List.foldr
                Core.insert [] initialItemList
              val initial = CORE (orderedItemList,0)
              in f(empty,nil,nil,[initial],nil,1)
            end,
      produces = produces,  rules = rules,
      epsProds = epsProds,  nonKernel = nonKernel,
      findRuleToLhs = findRuleToLhs}
end

```

```

<aus src/mkcover.sml>≡
  functor makeCover (Grammar, IntGrammar : GRAMMAR ...) =
    struct
      ...
      structure Core = mkCore(IntGrammar)
      structure Graph = mkGraph(IntGrammar,Core ...)
      ...
      datatype term = datatype Grammar.term (* = T of int *)
      datatype symbol = datatype Grammar.symbol

      datatype coverNonterm = LIST of symbol list
        | TUPLE of (symbol * int) (* core state nr *)
      datatype coverSymbol = COVERTERM of term
        | COVERNONTERM of coverNonterm
      datatype coverRule = COVERRULE of {
        lhs: coverNonterm, rhs: coverSymbol list,
        rulenumSG: int option, precedence: int option}

```

```
<aus src/mkcover.sml>+≡
```

```
fun mkFilters (grammar as GRAMMAR{nonterms,...}) = ...
  let ... in
    {nullable = nullable, (* A s.th. A =>* epsilon
      leftcorner = leftcorner,
      unit = unit, (* Chain rules A => B *)
      unitTrImage = unitTrImage, ...}
  end
(* auxiliary functions for printing and comparing *)
...
val computeCoverRules =
  fn (graph, grammar as GRAMMAR{rules,...,symbolToString,
    nonKernel, findRuleFromLhs) =>
    let ...
      fun extractRulesShift (CORE(items,state)) = .
      fun extractRulesGather (CORE(items,state)) =
      fun extractRulesMove (edges) = ...
      fun extractRulesInitiate (CORE(items,state))
```

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

2NF-Überdeckung

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

8. Juli 2010

Vereinfachung von ml-glr zu ml-glr2NF

Ziel

1. Grammatiktransformation in binäre Normalform (2NF): nur Regeln der Form $(A \rightarrow XY), (A \rightarrow X), (A \rightarrow \epsilon)$
2. Vorberechnung der Relationen

$$Del := \{ A \mid A \Rightarrow^+ \epsilon \}, \quad Unit^{-1}(B) := \{ A \mid A \Rightarrow^+ B \}$$

$$lc := \{ (X, C) \mid (C \rightarrow \gamma X \delta) \in P, \gamma \Rightarrow^* \epsilon \},$$

3. einfachere Überdeckungsgrammatik: $(2NF, Del, Unit, lc)$
4. einfacherer und effizienterer CYK-Parser, der $(A \rightarrow \epsilon), (A \rightarrow B)$ zur Parsezeit ignoriert (dafür: $Del, Unit, lc^*$)
5. LR-Filter $F(B, A) \iff \exists(A' \rightarrow BC) \in P (A lc^* C)$
6. Effizienzgewinn: Größe von $C(G)$, Parsezeit (testen)

Sei $G' = (\Sigma, N', P', S')$ in 2NF, dazu vorberechnet: *Del*, *Unit*, *lc*.

Die Parsetabelle F zu $w = a_1 \cdots a_n \neq \epsilon$ enthält für $0 \leq i < j \leq n$ eine Menge von „Kanten“ $i \xrightarrow{A} j$: die kleinste Menge von Kanten, die abgeschlossen ist unter:

$$\frac{(A \rightarrow a_{j+1}) \in P', \quad j \cdot A, \quad j < n}{j \xrightarrow{A} j+1} \text{ (scan)}$$

$$\frac{i \xrightarrow{B} k, \quad k \xrightarrow{C} j, \quad (A \rightarrow BC) \in P', \quad i \cdot A, \quad i < k < j}{i \xrightarrow{A} j} \text{ (complete)}$$

$$\frac{i \xrightarrow{B} j, \quad A \Rightarrow^+ B, \quad i \cdot A}{i \xrightarrow{A} j} \text{ (close)}$$

$$\frac{A \text{ lc}^* S'}{0 \cdot A} \text{ (start)} \quad \frac{k \xrightarrow{B} j, \quad (A' \rightarrow BC) \in P', \quad A \text{ lc}^* C}{j \cdot A} \text{ (predict)}$$

Mit den obigen Regeln wäre der Erkennen das reguläre Programm

start ; ((scan ; close* ; complete* ; close*) ; predict*)**

Das ist so gemeint, daß bei der äußeren Schleife (...) * der Index j durchlaufen wird und bei den inneren Schleifen ... * die übrigen Parameter ($i, k, \text{Regeln}, \text{Kanten}, \text{Erwartungskategorien}$).

Problem: Auswertungsregeln

Für eine Grammatik mit Auswertung, d.h. mit Regeln $(A \rightarrow \alpha, f_{A \rightarrow \alpha})$, muß man bei den Vorberechnungen auch Symbole mit Werten oder Regeln mit Auswertungsfunktionen berechnen.

Wenn die Grammatik azyklisch ist, kann man die Vorberechnung auch mit Attributgrammatiken machen:

1. Für die löschbaren Symbole werden Werte mitberechnet:

$$\frac{(A \rightarrow \epsilon, v_A) \in P}{(A, v_A) \in Del} (Del_0) \quad \frac{(A \rightarrow B, f) \in P, (B, v_B) \in Del}{(A, f(v_B)) \in Del} (Del_1)$$

$$\frac{(A \rightarrow BC, f) \in P, (B, v_B) \in Del, (C, v_C) \in Del}{(A, f(v_B, v_C)) \in Del} (Del_2)$$

2. Für Kettenregeln werden Auswertungsfunktionen mitberechnet:

$$\frac{(A \rightarrow B, f) \in P}{(A \rightarrow B, f) \in Unit} (Unit_0)$$

$$\frac{(A \rightarrow BC, f) \in P, \quad (B, v_B) \in Del}{(A \rightarrow C, \lambda v_C. f(v_B, v_C)) \in Unit} (Unit_1)$$

$$\frac{(A \rightarrow BC, f) \in P, \quad (C, v_C) \in Del}{(A \rightarrow C, \lambda v_B. f(v_B, v_C)) \in Unit} (Unit_1)$$

$$\frac{(A \rightarrow B, f) \in Unit, \quad (B \rightarrow C, g) \in Unit}{(A \rightarrow C, f \circ g) \in Unit} (Unit_2)$$

Bem.: Wenn die Grammatik zyklisch ist, kann man für (*close*) eine endliche Teilrelation von *Unit* berechnen (d.h. zu jedem $A \Rightarrow^+ B$ unter den Auswertungsfunktionen eine auswählen), aber welche?

Wie kann man (bei azyklischer Quellgrammatik) diese Hilfsrelationen berechnen, sodaß man aus den Bäumen der Überdeckungsgrammatik die der Quellgrammatik rekonstruieren kann, einschließlich passender Auswertungen?

- ▶ Man kann in die Bäume der Überdeckungsgrammatik die Verwendung der „neuen“ Unit-Regeln einbauen und bei der Extraktion die entsprechenden Bäume der Quellgrammatik einfügen. Das führte aber dazu, daß die Vorberechnungen (oder zumindest ihr Auswertungsteil) bei der Extraktion wiederholt ausgeführt werden, was ein Effizienzverlust wäre.
- ▶ Besser wäre es, die Actions-Struktur um die Unit-Regeln mit Auswertungsanteil zu erweitern und nur „kanonische“ Ableitungen der Quellgrammatik zu rekonstruieren. Geht es mit einer Zwischengrammatik?

Implementierung von Parsergeneratoren

Hauptseminar SS 2010

Programmieraufgaben zum Seminar

Hans Leiß, Georg Klein
Universität München
Centrum für Informations- und Sprachverarbeitung

22. Juli 2010

Programmieraufgaben zu mlglr2nf

1. Lexikonanschluß:

- 1.1 Die Verwendung von ML-Lex durch ein Programm ersetzen, das zu jeder Wortform alle lexikalischen Lesarten aus einem (öffentlich verfügbaren) Lexikon sucht und in Token umwandelt. (Erstmal ohne Semantik; später semantische Werte einem getrennten Lexikon entnehmen.)

Beachte: das Lexikon gibt die Wortarten vor; die Grammatik darf nur dazu passende Terminale deklarieren.

- 1.2 Anschluß an den Parsergenerator: aus der .cm-Datei mit G.lex2nf ein neues CM-Tool aufrufen?

Betreuer: Georg Klein, ggf. auch H.L.

Programmieraufgaben zu mlglr2nf

2. Berechnung der 2NF-Überdeckungsgrammatik:

2.1 Anfang siehe src/mkcover2NF.sm1; Graph ist eliminiert.

2.2 Ersetze Regeln $(A \rightarrow B_1 \cdots B_n)$ mit $n \geq 2$ durch

$$\begin{aligned} & (A \rightarrow X_1 [X_2, \dots, X_n]), \\ & ([X_2, \dots, X_n] \rightarrow X_2 [X_3, \dots, X_n]), \\ & \dots \\ & ([X_n] \rightarrow X_n) \end{aligned}$$

und führende Terminalsymbole X_i in verzweigenden Regeln durch neue Nonterminale X'_i und Regeln $(X'_i \rightarrow X_i)$.

2.3 Umrechnung von 2NF-Bäumen in Bäume der Quellgrammatik.

Betreuung: HL, ggf. Georg Klein

Programmieraufgaben zu mlglr2nf

3. Vorberechnungen:

3.1 Berechnung der Hilfsrelationen

$$\begin{aligned} \mathcal{D} &= \{A \mid A \Rightarrow^+ \epsilon\}, \\ \mathcal{C}_B &= \{A \mid A \Rightarrow^+ B\}, \\ \mathcal{LR} &= \{(C, A) \mid (A \rightarrow C) \in P\} \\ &\cup \{(B, A) \mid (A \rightarrow BC) \in P\} \\ &\cup \{(C, A) \mid (A \rightarrow BC) \in P \wedge B \Rightarrow^+ \epsilon\} \end{aligned}$$

einer 2NF-Grammatik (siehe `src/look.sml`),

3.2 Schreiben der Hilfsrelationen in eine Datei (`G.grm.sml`)

3.3 Behandlung der semantischen Annotationen überlegen

Betreuung: HL, GK

Programmieraufgaben zu mlglr2nf

4. 2NF-CYK: Anpassung des CYK aus bin/chartparser.sml
 - 4.1 zum Parsen nur die Regeln $(A \rightarrow BC)$ verwenden;
 - 4.2 statt der $(A \rightarrow \epsilon)$ und der $(B \rightarrow C)$ vorberechnete Relationen $\mathcal{D} = \{A \mid A \Rightarrow^+ \epsilon\}$ und $\mathcal{C}_B = \{A \mid A \Rightarrow^+ B\}$ verwenden.
 - 4.3 LR-Filter predict anpassen, mit \mathcal{LR} von oben.
 - 4.4 while-Schleife für $(A \rightarrow \epsilon)$, $(A \rightarrow B)$ -Regeln beseitigen; ggf. durch Wegsuche im Graphen $\{(A, B) \mid A \Rightarrow^+ B\}$.

Tip: 4.1 und 4.3 kann man mit CNF-Grammatiken testen, bevor 4.2 und 4.4 implementiert ist.

Betreuung: HL,GK

Status der Dateien zu `m1-g1r2nf` im Verzeichnis von `m1-g1r`:
fertig:

- ▶ `Makefile`, `build2NF`, `src/m1-g1r2NF.cm`: zum Installieren einer neuen Version von `m1-g1r2nf` (ggf. Pfade anpassen)
- ▶ `src/g1r2NF.sml`: zum Einlesen der Quellgrammatik
- ▶ `lib/m1-g1r2NF-lib.cm`: zum Bauen des Parsers (ggf. `chart2NF.sml` laden)

unfertig:

- ▶ `src/mkcover2NF.sml`: zum Erzeugen der 2NF-Überdeckung
- ▶ `src/look.sml`: Berechnen von \mathcal{D} , \mathcal{C} , \mathcal{LR} (ggf. effizienter)
- ▶ `lib/chartparser2NF.sml`: CYK für 2NF-Grammatiken mit Hilfsrelationen \mathcal{D} , \mathcal{C} , \mathcal{LR}
- ▶ `lib/chart.sml`: Anpassen der Transformation von Überdeckungs- zu Quellbäumen (ggf. `chart2NF.sml` nötig)