Syntaxanalyse (Parsing)
Vorlesung WS 2006/07

Hans Leiß, CIS

16. Januar 2007

(i) Nicht-deterministische Analyse (Backtracking)
   (a) Nicht-deterministische Top-Down-Analyse
   (b) Nicht-deterministische Bottom-Up-Analyse
   (c) Nicht-deterministische Left-Corner-Analyse
   (d) Nicht-deterministische Head-Corner-Analyse

(ii) Tabellengesteuerte Analyse (Chart-Parser)
   (a) Cocke-Younger-Kasami
   (b) Einfache Earley-Analyse
   (c) Einfache Left-Corner-Analyse
   (d) Vorberechnung von Erreichbarkeitsrelationen
   (e) Verallgemeinerte Earley-Analyse
   (f) Verallgemeinerte Left-Corner-Analyse

(iii) Deterministische Analyse (mit k-Vorausschau)

   (a) Top-Down: $LL(k)$-Anlayse
   (b) Bottom-Up: $LR(k)$-Analyse
   (c) Left-Corner: $LC(k)$-Analyse
   (d) (*) Verallgemeinerte $LR(k)$-Analyse (Tomita)

(iv) Kopfgrammatiken und gespaltene Konstituenten

Phasen der Textanalyse

(i) Lexikalische Analyse:

  (a) Reader: Lesen eines Stroms (Datei,Konsole) von Zeichen (Character)
  (b) Tokenizer: Zerlegen des Stroms in Segmente (Token)
  (c) Lemmatizer: Annotierung der Token mit Lexikon-Information
  (d) Tagger: Entfernung irrelevanter Lexikon-Information

(ii) Syntaktische Analyse:

  (a) Parser: Aufbau der syntaktischen Struktur der getaggten Token-Folge

(iii) Semantische Analyse:

  (a) Annotierung der syntaktischen Struktur mit semantischen Werten

(iv) Übersetzung:

  (a) Erzeugung eines Ausgabetexts in der Zielsprache (unter Erhaltung der semantischen Werte)

Lexikalische Analyse

Reader

(i) Falls die Eingabedatei noch nicht zum Lesen geöffnet ist, öffne sie zum Lesen und gib einen Namen für den dabei erzeugten Lesestrom aus.

(ii) Lies vom Eingabestrom bis zum ersten Satzende.

(iii) Gib den gelesenen Satz an den Tokenizer.

(iv) Auf Anforderung lies den nächsten Satz vom Eingabestrom oder schließe den Eingabestrom.

⟨*Strom-Prädikate von SWI-Prolog z.B.*⟩≡

```
current_stream(Datei?,Bearb.modus?,Stromname?)
open(Datei+,Bearbeitungsmodus+,Stromname-)
```

Mit `read_sentence(File+,Sentence-)` erfolgen die ersten beiden Schritte:

⟨*Reader 1.+2.*⟩≡

```
read_sentence(File,Sentence) :-
        (current_stream(File,read,Stream)
        -> true
        ; open(File,read,Stream)
        ),
        read_sentence(Stream,Sentence,[]).
```

Mit `read_sentence(Strom+,Sentence-,Puffer?)` wird das Satzende gesucht.

Satzende-Erkenner

`read_sentence(Strom+,Sentence-,Puffer?)`
`read_sentence(Strom+,Sentence-,Char+,Puffer?)`

(i) Falls der Eingabestrom zu Ende ist (`eof` oder `^C`), melde das (vorzeitige) Dateiende und brich ab.

(ii) Falls der Eingabestrom noch nicht zu Ende ist, lies ein weiteres Zeichen vom Strom.

(iii) Falls dieses Zeichen `eof` oder zusammen mit einem beschränkten Linkskontext (auf dem Puffer!) ein Satzende ist, gib die gelesene Zeichenfolge als `Sentence` zurück.

(iv) Sonst erweitere den Puffer um das gelesene Zeichen und suche weiter nach dem Satzende.

In Prolog sind Strings ASCII-Listen: `"abc" = [97, 98, 99]`, z.B. `Sentence` und `Puffer?`.

⟨*Satzende-Erkenner, 1.+2.*⟩≡

```
read_sentence(Stream,Sentence,Seen) :-
        get0(Stream,Char),
        !,read_sentence(Stream,Sentence,Char,Seen),
        (dateiende(Char)
         -> write('End of file reached.')
        ; true).
```

⟨*Satzende-Erkenner, 3.a*⟩≡

```
read_sentence(_Stream,Sentence,Char,Seen) :-
        dateiende(Char),
        !,reverse(Seen,Sentence).
```

⟨*Sonderfall:* `Char` *ist Leerzeichen nach Punkt im Satz*⟩
⟨*Sonderfall:* `Char` *ist ein Kommentarzeichen, z.B.* `%`⟩

Satzende-Erkenner (Forts.)

Unter den Sonderfällen muß man (Übung!) behandeln:

- Punkte am Ende von Ordinalzahlen (`der 3. Mann`)
- Punkte am Ende von Abkürzungen (`Dr. Sinn`)

Das Satzende `satzende([C,Char])` ist erreicht bei `". "`.

⟨*Satzende-Erkenner, 3.b*⟩≡

```
read_sentence(_Stream,Sentence,Char,[C|Seen]) :-
        satzende([C,Char]),
        !,reverse([C|Seen],Sentence).
```

⟨*Satzende-Erkenner, 4.*⟩≡

```
read_sentence(Stream,Sentence,Char,Seen) :-
        read_sentence(Stream,Sentence,[Char|Seen]).
```

Statt von einer Datei kann man auch vom Strom `user` lesen:

⟨*Satzende-Test an der Konsole*⟩≡

```
satzende :-
        write('Beende den Eingabesatz mit
               <Satzzeichen><Return>'),nl,
        read_sentence(user,Sentence,[]),
        name(Satz,Sentence),
        nl,writeq(Satz).
```

Tokenizer

`stringToAtoms(String+,Atomlist-)`

Der Satz `String` muß an den Leerzeichen in Wörter (Token) zerlegt werden. Dazu

(i) Lies den Satz zeichenweise,

(ii) Stapele gelesene Zeichen als Folge von Token,

(iii) Sammle im obersten Stapelelement (invertiertes *Präfix* eines Tokens!) weitere Character bis zum Tokenende.

(iv) Invertiere den Stapel der Token zu einer Atomliste.

Man unterscheidet nach dem zuletzt gelesenen Zeichen:

(`nt`) Das letzte Zeichen beendete ein Token.

`tokenize(String+,nt,SeenTokensRev?,Atoms-)`

(`at`) Das letzte Zeichen lag in einem angefangenen Token.

`tokenize(String+,at,SeenTokensRev?,Atoms-)`

Diese Fälle rufen sich gegenseitig auf, beginnend mit:

⟨*Aufruf des Tokenizers*⟩≡

```
  stringToAtoms(String,Atomlist) :-
          tokenize(String,nt,[],Atomlist).
```

Tokenizer (Details 1)

(`nt`) Das letzte Zeichen beendete ein Token.

⟨*Ignoriere führende Leerzeichen und Zeilenumbrüche:*⟩≡

```
tokenize([C|String],nt,Rs,Ts) :-
        (leerZeichen(C); zeilenende(C)),
        !,tokenize(String,nt,Rs,Ts).
```

⟨*Mache Sonderzeichen zu neuen Token:*⟩≡

```
tokenize([C|String],nt,Rs,Ts) :-
        sonderzeichen(Atom,C),
        !,tokenize(String,nt,[Atom|Rs],Ts).
```

⟨*Sonst wird das Zeichen* `C` *der Anfang eines Tokens:*⟩≡

```
tokenize([C|String],nt,Rs,Ts) :-
        !,tokenize(String,at,[[C]|Rs],Ts).
```

⟨*Falls der Eingabestring leer ist, gib die Token aus:*⟩≡

```
tokenize([],nt,Rs,Ts) :-
        reverse(Rs,Ts).
```

(`at`) Das letzte Zeichen lag in einem angefangenen Token.

⟨*Ignoriere Leerzeichen nach Worttrennung am Zeilenende:*⟩≡

```
tokenize([N,L|String],at,[[T|Cs]|Rs],Ts) :-
        worttrennung([T,N]),leerZeichen(L),
        !,tokenize([N|String],at,[[T|Cs]|Rs],Ts).
```

⟨*Ignoriere Worttrennung am Zeilenende:*⟩≡

```
tokenize([N|String],at,[[T|Cs]|Rs],Ts) :-
        worttrennung([T,N]),
        !,tokenize(String,at,[Cs|Rs],Ts).
```

Tokenizer (Details 2)

⟨*Am Zeilenende ohne Worttrenner endet ein Token:*⟩≡

```
  tokenize([C|String],at,[R|Rs],Ts) :-
          zeilenende(C),
          !,tokenToAtom(R,Atom),
          tokenize(String,nt,[Atom|Rs],Ts).
```

Die gesammelten Zeichen des Tokens werden in die normale Reihenfolge gebracht und zu einem Atom gemacht:

⟨*Umwandlung der Zeichenfolge des Tokens in ein Atom:*⟩≡

```
  tokenToAtom(RevToken,Atom) :-
          reverse(RevToken,Token),
          name(Atom,Token).
```

Das Atom kommt auf die Liste der Token und man sucht wieder nach einem neuen Token (Fall `nt`).

⟨*Weitere zu behandelnde Fälle*⟩≡
  ⟨*An einem Leerzeichen endet ein Token*⟩
  ⟨*Ziffer, Punkt/Komma, Leerzeichen beenden das Token*⟩
  ⟨*Ziffer, Punkt/Komma, Ziffer setzen das Token fort*⟩
  ⟨*Punkt am Ende einer Abkürzung beendet das Token*⟩
  ⟨*Sonderzeichen sind separate neue Token*⟩
  ⟨*Sonstige Zeichen setzen das Token fort*⟩
  ⟨*Am Ende des Eingabestrings endet das Token*⟩

Manchmal braucht man eine Vorausschau in den String:

⟨*Ziffer, Punkt, Leerzeichen beenden ein Token*⟩≡

```
  tokenize([S,L|String],at,[[C|Cs]|Rs],Ts) :-
          ziffer(C),punktZeichen(S),leerZeichen(L),
          !,tokenToAtom([S,C|Cs],Ordinalzahl),
          tokenize(String,nt,[Ordinalzahl|Rs],Ts).
```

Beispiel zur Tokenisierung

⟨*Aufrufe und Antworten von* `tokenize/4` *zeigen:*⟩≡

```
trace(tokenize,[call,return]).
```

Die ASCII-Nummern des Beispielstrings

⟨*String als ASCII-Liste*⟩≡

```
10 ?- "Der 2. Tag." =
      [68,101,114,32,50,46,32,84,97,103,46].
```

sind im Folgenden (von Hand) durch lesbare Zeichen ersetzt:

⟨*Aufruf des Tokenizers*⟩+≡

```
[debug]
11 ?- stringToAtoms("Der 2. Tag.",Tokens).
  (7) tokenize([D,e,r, ,2,., ,T,a,g,.],nt,[],_G)
  (8) tokenize([e,r, ,2,., ,T,a,g,.],at,[[D]],_G)
  (9) tokenize([r, ,2,., ,T,a,g,.],at,[[e,D]],_G)
  (10) tokenize([ ,2,., ,T,a,g,.],at,[[r,e,D]],_G)
  (11) tokenize([2,., ,T,a,g,.],nt,['Der'],_G)
  (12) tokenize([., ,T,a,g,.],at,[[2],'Der'],_G)
  (13) tokenize([T,a,g,.],nt,['2.','Der'],_G)
  (14) tokenize([a,g,.],at,[[T],'2.','Der'],_G)
  (15) tokenize([g,.],at,[[a,T],'2.','Der'],_G)
  (16) tokenize([.],at,[[g,a,T],'2.','Der'],_G)
  (17) tokenize([],nt,['.','Tag','2.','Der'],_G)

Tokens = ['Der', '2.', 'Tag', '.']

Yes
```

Lemmatizer

`lexAnalyse(Wortliste+)`

Die Lemmatisierung oder lexikalische Analyse im engeren Sinn:

(i) Beseitige das Sublexikon einer früheren Lemmatisierung.

(ii) Extrahiere aus einem (Vollform-) Lexikon zu jedem Wort die vorhandene Information (lexikalische Kategorie).

(iii) Speichere diese Information in einem Sublexikon.

Das Lexikon ist eine Faktensammlung in der PROLOG-Datenbank, das Sublexikon wird in die Datenbank geschrieben.

⟨*Format der Lexikoneinträge*⟩≡

```
word(Wortform?,Kategorie?).
```

⟨*Format der Sublexikoneinträge*⟩≡

```
lexword(Position?,Wortform?,Kategorie?).
```

Die Position ist die Position eines Worts in der Eingabefolge.

⟨*Lemmatisierer, 1.-3.*⟩≡

```
lexAnalyse(Wortliste) :-
     (gescheitert(lexAnalyse)
     -> retract(gescheitert(lexAnalyse)) ; true),
     retractall(lexword(_,_,_)),
     (ein(satzzeichen)  % Option: Satzzeichen erhalten
     -> Ws = Wortliste
     ; entferneSatzzeichen(Wortliste,Ws)),
     lexikalischeAnalyse(Ws,0),
     (ein(filterLexAnalyse) -> (nl,filterLexword); true).
```

Lemmatisierer (Fors.)

Zähle die Positionen mit und mache entsprechende Einträge in die Datenbank:

⟨*Lemmatisierer, 3.*⟩≡

```
lexikalischeAnalyse([W|Ws],N) :-
        eintraege(W,N),
        M is N+1,
        lexikalischeAnalyse(Ws,M).
lexikalischeAnalyse([],_).
```

⟨*Eintrag in* `lexword/3` *machen:*⟩≡

```
eintrag(Wortform,Kategorie,Position) :-
        assertz(lexword(Position,Wortform,Kategorie)).
```

Um die Einträge zu bilden, macht man in `eintraege(W,N)`:

(i) Falls W eine Zahl (Ordinal-, usw.) ist, gib die Zahlart als Kategorie an.

(ii) Falls W eine Abkürzung ist, sind ihre Kategorien einem Abkürzungslexikon zu entnehmen.

(iii) Falls W ein Satzzeichen ist, dient es als seine Kategorie.

(iv) Falls W Einträge im Vollformenlexikon hat, übertrage diese Kategorien.

(v) Falls W mit Großbuchstabe beginnt, keine Einträge im Vollformenlexikon hat und `N` der Satzanfang ist, suche Einträge zum W mit kleinem Anfangsbuchstaben.

(vi) Falls zu W keine Information vorhanden ist, gib eine Warnung `'unbekanntes Wort'` aus und erkläre die lexikalische Analyse als gescheitert. (Parser *nicht* aufrufen!)

⟨*Auszüge aus* `eintraege(W,N)`*:*⟩≡

```
...
eintraege(W,N) :-
        name(W,String),
        abkuerzung(String),
        !,eintraege_abk(String,W,N).
eintraege_abk(String,W,N) :-
        abkuerzung(String,Kategorie),
        eintrag(W,Kategorie,N),
        fail.
eintraege_abk(_String,_W,_N).
...
eintraege(W,N) :-
        word(W,_),
        !,lexKats(W,W,N).
...
lexKats(Orig,W,N) :-
        not(word(Orig,_)),
        write(user,'\nUnbekanntes Wort: '),write(user,W),
        !,eintrag(W,W,N),
        (gescheitert(lexAnalyse) -> true
          ; assertz(gescheitert(lexAnalyse))).
lexKats(Orig,W,N) :-
        word(Orig,Cat),
        eintrag(W,Cat,N),
        fail.
lexKats(_Orig,_W,_N).
```

Tagger

Hier kann man

(i) die nach statistischen Kriterien der üblichsten Trigramme (Tripel lexikalischer Kategorien aufeinanderfolgender Wörter) ausgeschlossenen Lesarten beseitigen,

(ii) die nach syntaktischen Kriterien an bestimmten Positionen (Satzanfang, -Ende, vor und nach Satzzeichen) ausgeschlossenen Lesarten beseitigen,

(iii) die nach Meinung des Benutzers ausgeschlossenen Lesarten beseitigen (nützlich zur Fehlersuche).

Bem. Für die Satzgrammatik ist nur die dritte Möglichkeit implementiert (vgl. die Option `filterLexAnalyse`).

Sprachdefinitionen und Grammatiken

Eingabealphabet: $\Sigma := c_1 \mid c_2 \mid \ldots \mid c_k$.

Eingabefolge: $w \in \Sigma^* := \bigcup_{i=0}^{\infty} \Sigma^n$, oft mit Endmarken: $\$w\$$

Eingabegrammatik: $G = (SynCat, \Sigma, P, S)$ mit

(i) endl. Menge *SynCat* von *syntaktischen Kategorien*,

(ii) endl. Menge Σ von *lexikalischen Kategorien* (*LexCat*)

(iii) endl. Menge P von *grammatischen Regeln*

(iv) ausgezeichnete *Satz-* oder *Hauptkategorie(n)* $S \in SynCat$

Wir betrachten statt einer Eingabefolge w von Zeichen gleich w als Ergebnis der Tokenisierung und Lemmatisierung, grob

$$\Sigma = \text{Alphabet} = \text{Terminalsymbole} = \text{Token} = LexCat.$$

Entsprechend für die durch G rekursiv definierten Kategorien:

$$X = \text{Rekursionsvariable} = \text{Nonterminalsymbole} = SynCat.$$

Eine *Satzform* zu G ist eine Folge $\alpha \in Cat^*$ von syntaktischen und lexikalischen Kategorien, mit

$$Cat := SynCat \cup LexCat.$$

Jede Grammatik G ist eine *endliche* Definition einer i.a. *unendlichen* formalen *Sprache* $L(G) \subseteq \Sigma^*$.

Kontextfreie Grammatiken

P ist eine Menge von Regeln $A \to \alpha$ mit $A \in \mathit{SynCat}$ und $\alpha \in \mathit{Cat}^*$.

Jedes $(A \to \alpha) \in P$ definiert auf Cat^* die Relation der *Ersetzung eines Vorkommens von A in β durch α*:

$$\beta \Rightarrow_{(A\to\alpha)} \gamma :\Longleftrightarrow \exists \beta_1, \beta_2 \in \mathit{Cat}^*[\beta \equiv \beta_1 A \beta_2 \wedge \gamma \equiv \beta_1 \alpha \beta_2]$$

Das wird fortgesetzt zu:

$$\begin{array}{lcl} \Rightarrow_P & := & \bigcup \{ \Rightarrow_{(A\to\alpha)} \mid (A \to \alpha) \in P \} \\ \Rightarrow_P^* & := & \text{reflexive transitive Hülle von } \Rightarrow_P . \end{array}$$

Vereinfachende Schreibweise:

$$\beta \Rightarrow^* \gamma \quad \text{oder} \quad \beta \to^* \gamma \qquad \text{statt} \qquad \beta \Rightarrow_P^* \gamma.$$

Damit erklärt man *die von G definierte Sprache* als

$$L(G) \quad := \quad \{ w \in \Sigma^* \mid S \Rightarrow_P^* w \}.$$

Syntaktische Struktur (allgemein)

Eine *syntaktische Struktur* eines sprachlichen Ausdrucks $w \in \Sigma^*$ bezüglich einer Sprachdefinition $G = (\mathit{SynCat}, \Sigma, P, S)$ ist *ein Beweis dafür, daß $w \in L(G)$.*

Eine grammatische Regel sollte eigentlich eine *Konstruktionsregel* sein, und jede Konstruktion eines Ausdrucks sollte eine *andere* syntaktische Struktur sein.

(„Freie“ Konstruktionen, „Äquivalenzklassen von“ Beweisen!)

Syntaktische Struktur
(kontextfreie Grammatik)

Bei einer kontextfreien Grammatik ist jede Ableitung

$$\gamma_1 \rightarrow_{(A_1 \rightarrow \alpha_1)} \gamma_2 \Rightarrow_{(A_2 \rightarrow \alpha_2)} \gamma_2 \Rightarrow \ldots \Rightarrow_{(A_n \rightarrow \alpha_n)} \gamma_n$$

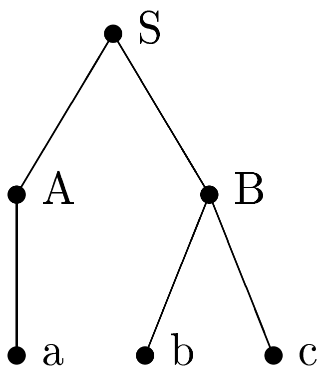
mit $S \equiv \gamma_1$ und $w \equiv \gamma_n$ ein Beweis für $w \in L(G)$.

Gewisse dieser Ableitungen gelten als *äquivalent* (s.u.)

Beisp. 1 Die beiden Ableitungen

$$S \Rightarrow_{(S \rightarrow AB)} AB \Rightarrow_{(A \rightarrow a)} aB \Rightarrow_{(B \rightarrow bc)} abc$$
$$S \Rightarrow_{(S \rightarrow AB)} AB \Rightarrow_{(B \rightarrow bc)} Abc \Rightarrow_{(A \rightarrow a)} abc$$

sind äquivalent, da sie denselben Syntaxbaum haben:



Die beiden Ableitungen

$$S \Rightarrow_{(S \rightarrow AB)} AB \Rightarrow_{(A \rightarrow a)} aB \Rightarrow_{(B \rightarrow bc)} abc$$
$$S \Rightarrow_{(S \rightarrow AC)} AC \Rightarrow_{(C \rightarrow bc)} Abc \Rightarrow_{(A \rightarrow a)} abc$$

sind nicht äquivalent für $B \not\equiv C$.

Syntaktische Struktur = Syntaxbaum

Jeder Ableitung ordnet man einen *Wald* zu, d.h. eine Folge *angeordneter Bäume* mit Knotenmarkierungen in *Cat*.

Für die Folge $A_1 \cdots A_n \equiv \alpha \in (\mathit{Cat} \cup \{\epsilon\})^*$ setze:

(i) der Wald (der Länge $n = |\alpha|$) zur Ableitung $\alpha \Rightarrow^0 \alpha$ ist

$$\langle \bullet A_1, \ldots, \bullet A_n \rangle.$$

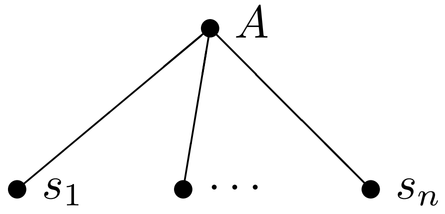
(ii) der Wald zur Ableitung

$$\alpha \equiv \beta A \gamma \Rightarrow_{(A \to A_1 \ldots A_n)} \beta A_1 \cdots A_n \gamma \Rightarrow^k \delta$$

ist

$$\langle r_1, \ldots, r_{|\beta|}, s, t_1, \ldots, t_{|\gamma|} \rangle,$$

wenn s der Baum



und

$$\langle r_1, \ldots, r_{|\beta|}, s_1, \ldots, s_n, t_1, \ldots, t_{|\gamma|} \rangle$$

der Wald zur Restableitung $\beta A_1 \cdots A_n \gamma \Rightarrow^* \delta$ ist.

Zwei Ableitungen heißen *äquivalent*, wenn ihnen hierdurch derselbe Wald zugeordnet wird.

Ein *Syntaxbaum* von G ist ein solcher Wald mit Länge 1, d.h. eine Äquivalenzklasse von Ableitungen $\alpha \Rightarrow^* \beta$ mit $|\alpha| = 1$.

Reguläre Ausdrücke über Σ als Sprachdefinitionen

Die Menge der *reguläre Ausdrücke* mit Konstanten c aus Σ und Variablen x aus X, kurz $reg_\Sigma(X)$, ist definiert durch

$$r, s \quad := \quad x \mid c \mid 0 \mid 1 \mid (r+s) \mid (r;s) \mid r^*$$

Sind Sprachen $A_i \subseteq \Sigma^*$ für die Variablen x_i in $r(x_1, \ldots, x_n)$ gegeben, so erhält man $L(r) := L(r)[A_1, \ldots, A_n]$ induktiv durch

$$\begin{array}{lcl@{\qquad}lcl}
L(0) & := & \emptyset & L(s+t) & := & L(s) \cup L(t) \\
L(1) & := & \{\epsilon\} & L(s\,;t) & := & L(s)L(t) \\
L(c) & := & \{c\} & L(s^*) & := & L(s)^* \\
L(x_i) & := & A_i & & &
\end{array}$$

Jedes $r \in reg_\Sigma$ (ohne Variablen) können wir als eine kontextfreie Grammatik $G_r = (SynCat, \Sigma, P, S)$ verstehen, wobei

(i) *SynCat* sind die Teilausdrücke von r,

(ii) P besteht aus folgenden Regeln, für $s, t \in SynCat$:

$$\begin{array}{lcl@{\qquad}lcl}
(s+t) & \to & s & s^* & \to & 1 \\
(s+t) & \to & t & s^* & \to & s\,s^* \\
(s;t) & \to & s\,t & 1 & \to & \epsilon
\end{array}$$

(iii) S ist r.

Natürlich erhält man $L(r) = L(G_r)$ für geschlossene reguläre Ausdrücke r, also mit G_r auch vernünftige(!) *Syntaxbäume für reguläre Sprachen.*

Interpretation durch Wortmengen

Die obige Interpretation der regulären Ausdrücke und der grammatischen Kategorien im (unendlichen) Modell

$$\mathcal{L} = (|\mathcal{L}|, +^{\mathcal{L}}, 0^{\mathcal{L}}, \cdot^{\mathcal{L}}, 1^{\mathcal{L}}) = (2^{\Sigma^*}, \cup, \emptyset, \cdot\,, \{\epsilon\})$$

interpretiert Kategorien A durch *Wortmengen* $L(A) \subseteq \Sigma^*$, wobei $+^{\mathcal{L}}$ die Vereinigung, $\cdot^{\mathcal{L}}$ die wortweise Verkettung

$$L(A \cdot B) = L(A) \cdot^{\mathcal{L}} L(B) := \{\, uv \mid u \in L(A), v \in L(B) \,\},$$

und $a \in \Sigma$ die Einermenge $L(a) := a^{\mathcal{L}} := \{a\}$ ist.

Interpretation durch Tokenmengen

Zur Analyse eines Satzes w braucht man aber nicht *alle* Ausdrücke einer Kategorie A, sondern nur die in w vorkommenden.

Sei $w = a_1 \cdots a_n$, $I = \{0, 1, \ldots, n\}$ und $v \in \Sigma^*$. Ein *Vorkommen* (Token) *von v in w* ist ein Paar $(i, j) \in I \times I$ mit $v = a_{i+1} \cdots a_j$. Die regulären Ausdrücke und grammatischen Kategorien kann man relativ zu w im endlichen(!) Modell

$$\mathcal{T} = (|\mathcal{T}|, +^{\mathcal{T}}, 0^{\mathcal{T}}, \cdot^{\mathcal{T}}, 1^{\mathcal{T}}) = (2^{I \times I}, \cup, \emptyset, \circ\,, =)$$

durch *Tokenmengen* $T(A) \subseteq I \times I$ interpretieren. Dabei wird $a \in \Sigma$ durch die Menge seiner Vorkommen in w interpretiert,

$$T(a) := a^{\mathcal{T}} := \{\, (i, i+1) \mid i \in I, a_i = a \,\},$$

$+^{\mathcal{T}}$ ist die Vereinigung und $\cdot^{\mathcal{T}}$ das Produkt von Relationen,

$$\begin{aligned} T(A \cdot B) &:= T(A) \cdot^{\mathcal{T}} T(B) \\ &:= \{\, (i, j) \mid \exists k \in I ((i, k) \in T(A) \wedge (k, j) \in T(B)) \,\} \end{aligned}$$

Beisp: In $w = {}_0a_1b_2a_3a_4b_5c_6b_7$ ist $T(ab) = \{(0, 2), (3, 5)\}$.

EBNF-Grammatiken

Eine *EBNF*-Grammatik $G = (SynCat, \Sigma, P, S)$ über Σ ist eine kontextfreie Grammatik mit Regeln

$$(A \to \alpha), \quad \text{wobei } \alpha \in reg_\Sigma(SynCat).$$

Die Menge $\{A \to \alpha_1, \ldots, A \to \alpha_k\}$ aller A-Regeln von G fassen wir zu einer Gleichung $A = \alpha_1 + \cdots + \alpha_k$ zusammen.

Reguläre Gleichungssysteme über Σ

in den Rekursionsvariablen $X = \{x_1, \ldots, x_n\}$ sind Systeme

$$\begin{array}{rcl} x_1 & = & r_1(x_1, \ldots, x_n) \\ \vdots & & \vdots \\ x_n & = & r_n(x_1, \ldots, x_n) \end{array}$$

mit $r_i(x_1, \ldots, x_n) \in reg_\Sigma(X)$. Sie sind eine andere Notation für EBNF-Grammatiken $G = (X, \Sigma, P, x_1)$ mit passendem P.

Definierte Sprache: L_1 für das bzgl. $\subseteq$ kleinste Tupel $L = (L_1, \ldots, L_n)$ von Sprachen über Σ mit

$$\begin{array}{rcl} L_1 & = & r_1(L_1, \ldots, L_n) \\ \vdots & & \vdots \\ L_n & = & r_n(L_1, \ldots, L_n) \end{array}$$

Die nicht-atomaren regulären Teilausdrücke r von r_i bzw. α nennen wir *zusammengesetzte Kategorien.*

Beispiel

Beisp. 2 Die Gleichungssysteme

$$\begin{aligned} x &= a(x+y)b+1 \qquad & x &= axb+ayb+1 \\ y &= b(x+1)+y \qquad & y &= bx+b+y \end{aligned}$$

entsprechen der CF-Grammatik mit $S := x$ und den Regeln

$$x \to axb,\ x \to ayb,\ x \to \epsilon,\ y \to bx,\ y \to b,\ y \to y.$$

Kleinste Lösung eines Gleichungssystems

Sei $\mathcal{M}$ das unendlichen Modell $\mathcal{L}$ aller Wortmengen über Σ oder das endliche Modell $\mathcal{T}$ aller Tokenmengen über $w \in \Sigma^*$.

Die kleinste Lösung $(M_1, \ldots, M_n)$ eines Gleichungssystems

$$x_i = r_i(x_1, \ldots, x_n), \quad (1 \leq i \leq n)$$

in $\mathcal{M}$ erhält man, indem man mit den Mengen (Relationen)

$$M_{i,0} := \emptyset \quad (1 \leq i \leq n)$$

beginnt und dann den Wert der regulären Ausdrücke ergänzt

$$M_{i,k+1} := r_i(M_{1,k}, \ldots, M_{n,k}) \cup M_{i,k}.$$

Die Komponenten der kleinsten Lösung $(M_1, \ldots, M_n)$ sind dann

$$M_i := \bigcup_{k \in \mathbb{N}} M_{i,k}, \quad (1 \leq i \leq n).$$

Beachte: in $\mathcal{M} = \mathcal{T}$ ist die Kette

$$\emptyset = M_{i,0} \subseteq M_{i,1} \subseteq \ldots \subseteq M_i \subseteq I \times I$$

endlich!

NICHT-DETERMINISTISCHE ANALYSE
(universell)

Nicht-det. Top-Down-Analyse

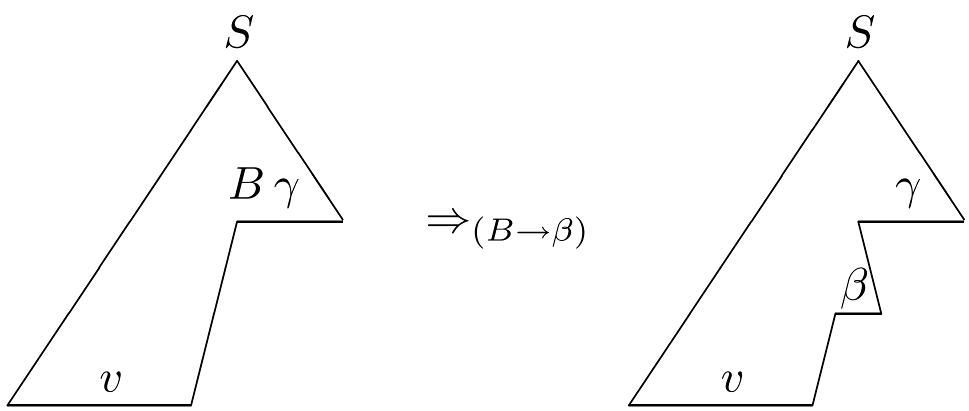
Die Top-Down- oder *hypothesengesteuerte* Analyse geht von der Hypothese, die Eingabe w sei ein Ausdruck einer bestimmten (Start-) Kategorie S, aus und versucht, eine *Linksableitung*

$$S \Rightarrow_{lm} \alpha_1 \Rightarrow_{lm} \alpha_2 \Rightarrow_{lm} \ldots \Rightarrow_{lm} w$$

zu finden. D.h. in jedem Schritt $\alpha_i \Rightarrow_{lm} \alpha_{i+1}$ wird die am weitesten links in α_i stehende syntaktische Kategorie

$$\alpha_i \equiv vB\gamma \Rightarrow_{(B \to \beta)} v\beta\gamma \equiv \alpha_{i+1}$$

für ein $v \in \Sigma^*$, $B \in \mathit{SynCat}$ und eine Regel $(B \to \beta) \in P$ expandiert:



a) Top-Down-Erkenner

nicht-det. Kellerautomat $\mathcal{M}^G$

Konfigurationen:	$(\alpha, w) \in SynCat^* \times \Sigma^*$
Anfangskonfiguration:	$(S, a_1 \cdots a_n)$
Endkonfiguration:	(ϵ, ϵ)
Befehle:	
Expandiere mit $(B \to \beta) \in P$:	$(B\gamma, w) \vdash_{(B\to\beta)} (\beta\gamma, w)$
Vergleiche mit Eingabe $a \in \Sigma$:	$(a\gamma, aw) \vdash_a (\gamma, w)$
Programm von $\mathcal{M}^G$:	$r := (P \cup \Sigma)^*$

Prop. 3 $w \in L(G) \iff S \Rightarrow^*_{lm} w \iff (S, w) \vdash_M (\epsilon, \epsilon)$

Beisp. 4 $\Sigma := \{a, b, c\}, \quad P := \{S \to AC,\ A \to AB,\ A \to a,\ B \to b,\ C \to BC,\ C \to c\}$

Linksableitung:			Erkennung durch $\mathcal{M}^G$		
S	$\Rightarrow_{(S\to AC)}$	AC	(S, abc)	$\vdash_{(S\to AC)}$	(AC, abc)
	$\Rightarrow_{(A\to AB)}$	ABC		$\vdash_{(A\to AB)}$	(ABC, abc)
	$\Rightarrow_{(A\to a)}$	aBC		$\vdash_{(A\to a)}$	(aBC, abc)
	$\Rightarrow_{(B\to b)}$	abC		$\vdash_a$	(BC, bc)
	$\Rightarrow_{(C\to c)}$	abc		$\vdash_{(B\to b)}$	(bC, bc)
				$\vdash_b$	(C, c)
				$\vdash_{(C\to c)}$	(c, c)
				$\vdash_c$	(ϵ, ϵ)

b) Top-Down-Parser
mit Rücksetzungen

Die Grammatikregeln seien numeriert.

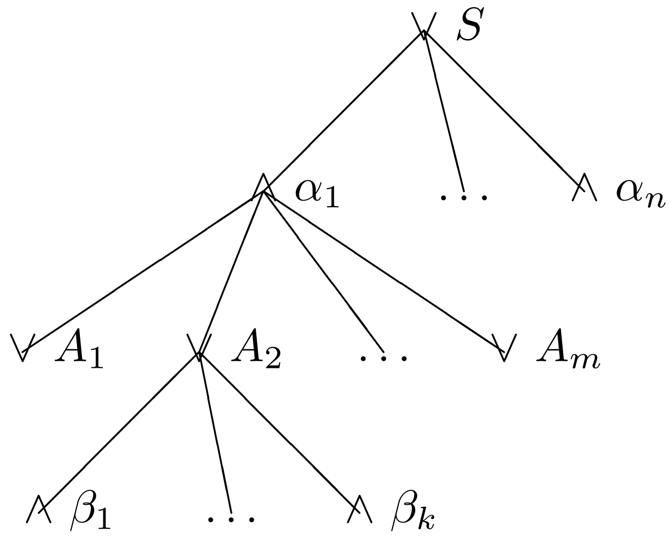
Tiefensuche: Ersetze in α das linkeste $A \in \mathit{SynCat}$ mit der nächsten möglichen Regel $p : (A \to \alpha)$; wenn ein Vergleich mit dem Eingabesymbol scheitert, gehe zurück zur letzten Wahlmöglichkeit und benutze die nächste Regel.

Konfigurationen (α, w, π, p)	$\in \mathit{Cat}^* \times \Sigma^* \times \mathit{Parse} \times P'$
Anfangskonfiguration:	$(S, a_1 \cdots a_n, \epsilon, 0)$
Endkonfiguration:	$(\epsilon, \epsilon, \pi, p)$
Befehle:	
Expandiere mit $p = (A \to \alpha)$:	$(A\beta, w, \pi, p) \vdash_p (\alpha\beta, w, p\pi, 0)$
Suche Regel für $p \neq (A \to \alpha)$:	$(A\beta, w, \pi, p) \vdash (A\beta, w, \pi, p+1)$
bzw. für $p_{\max} = p \neq (A \to \alpha)$:	$(A\beta, w, \pi, p) \vdash (A\beta, w, \pi, \mathit{redo})$
Lesen der Eingabe $a \in \Sigma$:	$(a\beta, aw, \pi, 0) \vdash_a (\beta, w, a\pi, 0)$
bzw. Lesekonflikt $a \neq b$:	$(a\beta, bw, \pi, 0) \vdash (a\beta, bw, \pi, \mathit{redo})$
bzw. Lesekonflikt $\epsilon \neq b$:	$(\epsilon, bw, \pi, 0) \vdash (\epsilon, bw, \pi, \mathit{redo})$
bzw. Lesekonflikt $a\beta \neq \epsilon$:	$(a\beta, \epsilon, \pi, 0) \vdash (a\beta, \epsilon, \pi, \mathit{redo})$
Redo mit $p = (A \to \alpha)$:	$(\alpha\beta, w, p\pi, \mathit{redo}) \vdash (A\beta, w, \pi, p+1)$
bzw. Stop bei $p = p_{\max}$:	$(\alpha\beta, w, p\pi, \mathit{redo}) \vdash (A\beta, w, \pi, \mathit{stop})$
Redo mit $a \in \Sigma$:	$(\alpha, w, a\pi, \mathit{redo}) \vdash (a\alpha, aw, \pi, \mathit{redo})$
Erfolgreiche Analyse:	$(\epsilon, \epsilon, \pi, 0) \vdash (\epsilon, \epsilon, \pi, \mathit{redo})$

Nachteil: Divergiert bei Linksrekursion $A \Rightarrow^+ A\beta$.

Suchbaum der Linksableitungen aus S

Wähle von S ausgehend bei jedem mit $A \in \mathit{SynCat}$ markierten $\vee$-Knoten *einen* Ast (Regel $A \to \beta$) und dann bei dem mit β markierten Nachfolger ($\wedge$-Knoten) für *jeden* Nachfolger B wieder einen Ast, usw.



Eine Analyse wird durch die Folge π der gewählten Regeln und der in der Eingabe gefundenen Blattmarken $a \in \Sigma$ kodiert.

Die Suche nach einer Linksableitung gleicht der Beweissuche in Prolog, wo zuerst die linkesten Teilziele (Konjunktionsglieder) und dann die weiter rechts stehenden bewiesen werden.

⟨*Kontextfreie Grammatikregel*⟩≡

```
np(sg,nom) ---> det(sg,nom) ap(sg,nom) n(sg,nom)
```

⟨*Entsprechende Prolog-Klausel*⟩≡

```
np(sg,nom,I,J) :-
    det(sg,nom,I,K), ap(sg,nom,K,L), n(sg,nom,L,J).
```

mit *Differenzlistendarstellung* $I = [a_{i+1}, \ldots, a_j | J]$ von Teillisten $v = a_{i+1}, \ldots, a_j$ der Eingabe.

Beispiel zum Top-Down-Parser

Der Top-Down-Parser durchsucht bei jedem $\vee$-Knoten zuerst den linkesten Teilbaum.

Beisp. 5 Sei G die Grammatik mit den Regeln

$$0 : (A \to BC), \quad 1 : (B \to ab), \quad 2 : (C \to c), \quad 3 : (C \to AB)$$

Der Parser arbeitet bei Eingabe von abc und Ziel A so:

$$\begin{array}{rlrl}
 & (A, abc, \epsilon, 0) & \vdash & (BC, abc, 0, 0) \\
\vdash & (BC, abc, 0, 1) & \vdash & (abC, abc, 10, 0) \\
\vdash & (bC, bc, a10, 0) & \vdash & (C, c, ba10, 0) \\
\vdash & (C, c, ba10, 1) & \vdash & (C, c, ba10, 2) \\
\vdash & (c, c, 2ba10, 0) & \vdash & (\epsilon, \epsilon, c2ba10, 0) \text{ (1.Analyse!)} \\
\vdash & (\epsilon, \epsilon, c2ba10, redo) & \vdash & (c, c, 2ba10, redo) \\
\vdash & (C, c, ba10, 3) & \vdash & (AB, c, 3ba10, 0) \\
\vdash & (BCB, c, 03ba10, 0) & \vdash & (BCB, c, 03ba10, 1) \\
\vdash & (abCB, c, 103ba10, 0) & \vdash & (abCB, c, 103ba10, redo) \\
\vdash & (BCB, c, 03ba10, 2) & \vdash & (BCB, c, 03ba10, 3) \\
\vdash & (BCB, c, 03ba10, redo) & \vdash & (AB, c, 3ba10, 1) \\
\vdash & (AB, c, 3ba10, 2) & \vdash & (AB, c, 3ba10, 3) \\
\vdash & (AB, c, 3ba10, redo) & \vdash & (C, c, ba10, stop)
\end{array}$$

Implementierung in Prolog

$\langle$*Regelformat*$\rangle\equiv$
```
(SynCat ---> Cat1, ..., [Word], ..., CatN).
```

Beisp. 6

$\langle$*cfg.grm*$\rangle\equiv$
```
% Startsymbole:

start_symbol(s(pl)).
start_symbol(s(sg)).

% Grammatikregeln:

:- op(1100,xfx,'--->').

s(sg) ---> a(sg), b(sg).
s(pl) ---> a(sg), b(pl).
b(pl) ---> b(sg), [c], b(pl).
b(pl) ---> b(sg), a(sg).

% Lexikon:

word(a,a(sg)).
word(b,b(sg)).
% word(b(sg),b(sg)). % (f. Kategorie in der Eingabe)
```

Definierte Sprache:

$$L(cfg.grm) = \{ab\} \cup \{\, a(bc)^n ba \mid n \in \mathbb{N} \,\}.$$

c) Top-Down-Erkenner in Prolog

universell, mit Rücksetzungen

Vor.: G definiert die Prädikate `'--->'/2` und `start_symbol/1` und ist (unbetrachtet) in den Arbeitsspeicher geladen.

⟨*Differenzlistendarstellung von* `[`a_{i+1}`,...,`a_j`]` *durch I,J:*⟩≡

```
  I = [a_i+1,...,a_j|J]
```

⟨*top-down-rec.cfg.pl*⟩≡

```
  parse(Atoms) :-
          start_symbol(S),
          parse(S,Atoms,[]).
  parse(A,I,J) :-
          (A ---> B),  % A ist SynCat mit einer Regel
          tupleToList(B,Bs),
          parse_list(Bs,I,J).
  parse(A,I,J) :-      % A ist LexCat
          I = [W|J],
          ( word(W,A)  % eines Worts im Lexikon
          ; A = [W] ). % oder in einer Regel
  ⟨tupleToList⟩
  parse_list([B|Bs],I,J) :-
          parse(B,I,K), parse_list(Bs,K,J).
  parse_list([],I,J) :- I = J.
```

Hilfsfunktion (auch für andere Parser):

⟨*tupleToList*⟩≡

```
  tupleToList((B,C),[B|Cs]) :- !, tupleToList(C,Cs).
  tupleToList([],[]).
  tupleToList([W],[[W]]).
  tupleToList(B,[B]).
```

d) Top-Down-Parser mit explizitem Stapel

Der Erkenner ⟨*top-down-rec.cfg.pl*⟩ benutzt die in Prolog eingebaute Beweissuche zur Verwaltung der Alternativen. Merkt man sich auf einem Stapel die benutzten Regeln und gelesenen Token, so kann man daraus eine Analyse rekonstruieren:

⟨*top-down-stack.cfg.pl*⟩≡
```
:- op(1100,xfx,'--->').

parse(Atoms) :-
        start_symbol(S),
        parse(S,[],Parse,Atoms,[]).

parse([W],Stapel,Parse,I,J) :-    % Terminal in
        !, I = [W|J],              % einer Regel
        Parse = [W|Stapel].
parse((B,Bs),Stapel,Parse,I,J) :- % CFG-Regel
        !, parse(B,Stapel,BStapel,I,K),
        parse(Bs,BStapel,Parse,K,J).

parse(A,Stapel,Parse,I,J) :-
        (A ---> B),
        parse(B,[(A ---> B)|Stapel],Parse,I,J).
parse(A,Stapel,Parse,I,J) :-
        I = [W|J],
        word(W,A),
        Parse = [W|Stapel].
```

Beispiel

Die Analyse ergibt sich aus dem invertierten Stapel

⟨*Beispiel* ⟩≡
```
?- ['cfg.grm', 'top-down-stack.cfg'].

?- parse(s(A),[],Parse,[a,b,a],[]).

A = pl
Parse = [a, b, (b(pl)--->b(sg), a(sg)),
             a, (s(pl)--->a(sg), b(pl))]
```

wenn man die Benutzung des Lexikons berücksichtigt:

$$
\begin{array}{lll}
(s^{pl}, aba) & \vdash_{s^{pl} \to a^{sg} b^{pl}} & (a^{sg} b^{pl}, aba) \\
 & \vdash_{a^{sg} \to a} & (ab^{pl}, aba) \\
 & \vdash_{a} & (b^{pl}, ba) \\
 & \vdash_{b^{pl} \to b^{sg} a^{sg}} & (b^{sg} a^{sg}, ba) \\
 & \vdash_{b^{sg} \to b} & (ba^{sg}, ba) \\
 & \vdash_{b} & (a^{sg}, a) \\
 & \vdash_{a^{sg} \to a} & (a, a) \\
 & \vdash_{a} & (\epsilon, \epsilon)
\end{array}
$$

Einfache Baumanzeige

Wir kodieren Bäume durch Listen in der Form

⟨*Baumkodierung:* ⟩≡

```
[Kategorie der Wurzel | Teilbäume]
```

Eine lesbare Ausgabe der Bäume liefert dann:

⟨*printTree.pl*⟩≡

```
printTree(Tree,Indent) :-
    printTree(Tree,5,Indent).

printTree([A,W],I,Indent) :-
    atom(W),!,
    nl, tab(I), writeq(A), tab(2), writeq(W).
printTree([A|Trees],I,Indent) :-
    !, printTree(A,I,Indent),
    J is I + Indent, printTrees(Trees,J,Indent).
printTree(A,I,_) :-
    nl, tab(I), writeq(A).

printTrees([T|Ts],I,Indent) :-
    printTree(T,I,Indent),
    printTrees(Ts,I,Indent).
printTrees([],_I,_Indent).
```

Hierbei ist `I` die horizontale Einrückung vom linken Zeilenrand und `Indent` die relative Einrückung der Teilbäume.

d) Top-Down-Parser

(universell, für DCGs)

⟨*Einfache und zusammengesetzte Kategorien*⟩≡

```
Cat := Atomic                              % Einfache Kat.
     | (Cat,Cat) | (Cat;Cat) | {Goal} | [W|Ws] | []
```

Konstruiert man für alle `Cat` Baum*listen*, so braucht man für die komplexen Kategorien keine Knoten im Syntaxbaum:

⟨*top-down.pl*⟩≡

```
:- op(1100,xfx,'--->').
parse(Atoms,Tree) :-
    start_symbol(S), parse(S,[Tree],Atoms,[]).

parse((B-C),Trees,I,J) :-   % (Differenz, s.u.)
    !, parse(B,Trees,I,J), not(parse(C,_,I,J)).
parse((B,C),Trees,I,J) :-   % Zusammeng.Kategorien
    !, parse(B,TreesB,I,K), parse(C,TreesC,K,J),
    append(TreesB,TreesC,Trees).
parse((B;C),Trees,I,J) :-
    !, (parse(B,Trees,I,J) ; parse(C,Trees,I,J)).
parse({Goal},Trees,I,J) :-
    !, call(Goal), Trees = [], I = J.
parse([],Trees,I,J) :-
    !, Trees = [], I = J.
parse([W|Ws],Trees,I,J) :-
    !, append([W|Ws],J,I), Trees = [W|Ws].

parse(A,Trees,I,J) :-       % Einfache Kategorien
    (A ---> Body),
    parse(Body,SubTrees,I,J), Trees = [[A|SubTrees]].
parse(A,Trees,I,J) :-
    I = [W|J], word(W,A), Trees = [[A,W]].
```

Beispiel

⟨*cfg3.grm*⟩≡

```
start_symbol('A').

:- op(1100,xfx,'--->').

'A' ---> ((('B',[c]),'D') ; ('D',('B';'A'))).
'B' ---> [a,b].
'C' ---> 'A','B'.
'D' ---> [d].

word(c,'C').
```

Anwendung

⟨*Beispiel* ⟩+≡

```
?- ['top-down','cfg3.grm',printTree].
?- parse([d,a,b,c,d],Baum), printTree(Baum,4).

    'A'
       'D'
          d
       'A'
          'B'
             a
             b
          c
          'D'
             d

Baum = ['A', ['D', d], ['A', ['B', a, b], c, ['D', d]]]
```

Beispiel: Definite Clause Grammar

Beisp. 7 Grammatik *cfg.grm* kompakter als DCG formuliert:

$\langle$*dcg.grm*$\rangle\equiv$

```
% Startsymbole:

start_symbol(s(pl)).
start_symbol(s(sg)).

% Grammatikregeln:

:- op(1100,xfx,'--->').

s(Num) ---> a(sg),
        ( b(sg), {Num = sg}
        ; b(pl), {Num = pl}
        ).
b(pl) ---> b(sg), ([c], b(pl) ; a(sg)).

% Lexikon:

word(a,a(sg)).
word(b,b(sg)).
```

Definierte Sprache:

$$L(dcg.grm) = L(cfg.grm) = \{ab\} \cup \{\, a(bc)^n ba \mid n \in \mathbb{N}\,\}.$$

Beachte, wie mit `{Goal}` das Merkmal `Num` abhängig vom benutzten Disjunktionsglied belegt und vererbt wird.

$\langle$*Beispielanalyse*$\rangle\equiv$

```
time(parse([a,b,c,b,a],Baum)), printTree(Baum,4).

    s(pl)
       a(sg)
          a
       b(pl)
          b(sg)
             b
          c
          b(pl)
             b(sg)
                b
             a(sg)
                a

% 150 inferences, 0.00 CPU in 0.00 seconds
  (0% CPU, Infinite Lips)

Baum = [s(pl), [a(sg), a],
        [b(pl), [b(sg), b], c, [b(pl), [...|...]|...]]] ;
```

Beispiel für Differenzkategorien

Ein Top-Down-Parser kann mit `(C-D)` *ausschließen*, daß eine Konstituente `C` eine bestimmte Form `D` hat: z.B. darf ein Genitivattribut keine pronominalisierte Nominalphrase sein.

⟨*cfg4.grm*⟩≡

```
start_symbol('NP').
:- op(1200,xfx,'--->').

'NP'    ---> 'proNP' ; 'Det', 'N', ([] ; 'NPgen').
'NPgen' ---> 'NP' - 'proNP'.  % Differenzkategorie!
'Det'   ---> 'Art' ; poss.
'Art'   ---> [die] ; [der] ; [das] ; [des].
'N'     ---> [tür] ; [haus] ; [hauses].
'proNP' ---> [er].
'poss'  ---> [seine] ; [seines].
```

⟨*Nur nicht-pronominalisierte NPs sind Genitivattribute*⟩≡

```
?- parse('NP',Baum,[er],[]).
Baum = [['NP', [proNP, er]]]

?-  parse('NP',Baum,[seines,hauses],[]).
Baum = [['NP', ['Det', [poss, seines]], ['N', hauses]]]

?-  parse('NP',Baum,[die,tür,seines,hauses],[]).
Baum = [['NP', ['Det', ['Art', die]], ['N', tür],
               ['NPgen', ['NP', [...|...]|...]]]]

?-  parse('NP',Baum,[die,tür,er],[]).
No
```

Schwieriger: Tests an die Struktur erkannnter Konstituenten!

Nicht-det. Bottom-Up-Analyse

Die Bottom-Up- oder *Shift-Reduce*- oder *datengesteuerte* Analyse versucht, eine *Rechtsableitung*

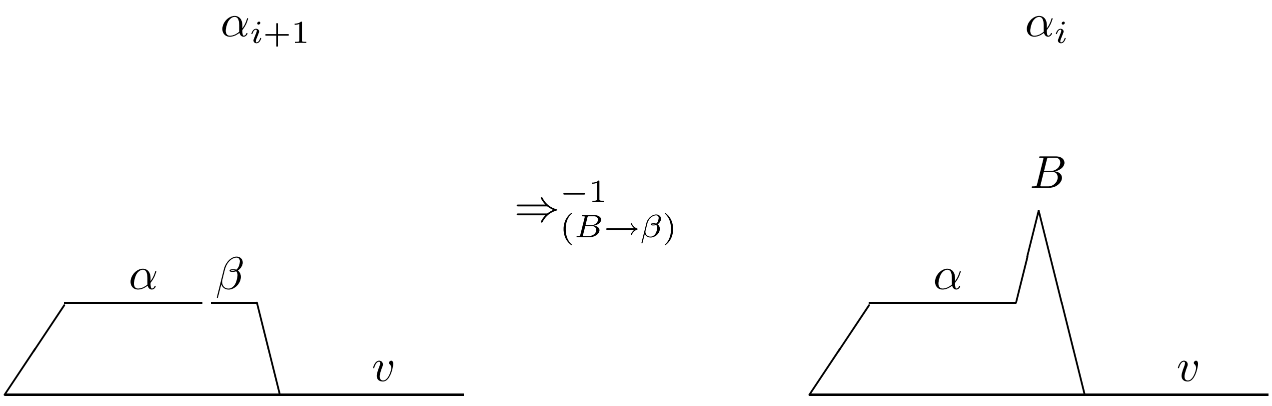
$$S \Rightarrow_{rm} \alpha_1 \Rightarrow_{rm} \alpha_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} w$$

zu finden. D.h. in jedem Schritt $\alpha_i \Rightarrow_{rm} \alpha_{i+1}$ benutzt man die am weitesten rechts in α_i stehende syntaktische Kategorie

$$\alpha_i \equiv \alpha B v \quad \Rightarrow_{(B \to \beta)} \quad \alpha\beta v \equiv \alpha_{i+1}$$

für ein $v \in \Sigma^*$, $B \in \mathit{SynCat}$ und eine Regel $(B \to \beta) \in P$.

Die Ableitungsschritte werden in umgekehrter Reihenfolge durch Reduktionen von α_{i+1} zu α_i rekonstruiert:



In $\alpha_{i+1} \equiv \alpha\beta v$ muß man den *Henkel* β finden und reduzieren.

a) Bottom-Up-Erkenner

(nicht-det. Kellermaschine $\mathcal{M}_G$)

Konfigurationen:	$(\alpha, w) \in Cat^* \times \Sigma^*$
Anfangskonfiguration:	$(\epsilon, a_1 \cdots a_n)$
Endkonfiguration:	(S, ϵ)
Befehle:	
Reduziere mit $(B \to \beta) \in P$:	$(\alpha\beta, w) \vdash_{(B\to\beta)} (\alpha B, w)$
Lese $a \in \Sigma$ von der Eingabe:	$(\alpha, aw) \vdash_a (\alpha\, a, w)$
Programm von $\mathcal{M}_G$:	$r := (P \cup \Sigma)^*$

Prop. 8 $w \in L(G) \iff S \Rightarrow_{rm}^* w \iff (\epsilon, w) \vdash_{M_G} (S, \epsilon)$

Beisp. 9 $\Sigma := \{a, b, c\}, \quad P := \{S \to AC,\ A \to AB,\ A \to a,$
$B \to b,\ C \to BC, C \to c\}$

Rechtsableitung:			Erkennung durch $\mathcal{M}_G$		
S	$\Rightarrow_{(S\to AC)}$	AC	(ϵ, abc)	$\vdash_a$	(a, bc)
	$\Rightarrow_{(C\to c)}$	Ac		$\vdash_{(A\to a)}$	(A, bc)
	$\Rightarrow_{(A\to AB)}$	ABc		$\vdash_b$	(Ab, c)
	$\Rightarrow_{(B\to b)}$	Abc		$\vdash_{(B\to b)}$	(AB, c)
	$\Rightarrow_{(A\to a)}$	abc		$\vdash_{(A\to AB)}$	(A, c)
				$\vdash_c$	(Ac, ϵ)
				$\vdash_{(C\to c)}$	(AC, ϵ)
				$\vdash_{(S\to AC)}$	(S, ϵ)

b) Bottom-Up-Parser

(deterministisch, mit Rücksetzungen)

Prinzip: Falls möglich, reduziere den Stapel mit der gegenwärtigen (numerierten) Grammatikregel. Nach jeder Reduktion versuche wieder zu reduzieren. Falls keine Reduktion möglich ist, schiebe das nächste Eingabesymbol auf den Stapel, und versuche wieder zu reduzieren. Falls kein Eingabesymbol mehr vorhanden ist, setze bis zur letzten Alternative zurück und versuche es mit der nächsten Regel.

Abstrakte Maschine:

Konfigurationen (α, w, π, p)	$\in Cat^* \times \Sigma^* \times Parse \times P'$
Anfangskonfiguration:	$(\epsilon, a_1 \cdots a_n, \epsilon, 0)$
Befehle: $(p = (B \to \beta))$	
Reduziere $\gamma \equiv \alpha\beta$:	$(\gamma, w, \pi, p) \vdash_p (\alpha B, w, p\pi, 0)$
Nächste Regel, $\gamma \not\equiv \alpha\beta$:	$(\gamma, w, \pi, p) \vdash (\gamma, w, \pi, p+1)$
bzw. Shift:	$(\gamma, aw, \pi, p_{\max}) \vdash_a (\gamma a, w, \pi, 0)$
bzw. Lesekonflikt:	$(\gamma, \epsilon, \pi, p_{\max}) \vdash (\gamma, \epsilon, \pi, redo)$
Redo mit $p \neq p_{\max}$:	$(\alpha B, w, p\pi, redo) \vdash (\alpha\beta, w, \pi, p+1)$
Redo mit $p = p_{\max}$:	$(\alpha B, w, p\pi, redo) \vdash (\alpha\beta, w, \pi, redo)$
Redo mit $a \in \Sigma$:	$(\alpha a, w, \pi, redo) \vdash (\alpha, aw, \pi, redo)$
Erfolgreiche Analyse:	$(S, \epsilon, \pi, 0) \vdash (S, \epsilon, \pi, redo)$
Stop:	$(\epsilon, w, \epsilon, redo) \vdash (\epsilon, w, \epsilon, stop)$

Nachteil: Divergiert bei $A \Rightarrow^+ A$ und $B \to \epsilon$

Exponentieller Aufwand in $|w|$

Beispiel zum Bottom-Up-Parser

Beisp. 10 Sei G die Grammatik mit den Regeln

$$0 : (A \to BC), \quad 1 : (B \to ab), \quad 2 : (C \to c), \quad 3 : (C \to AB)$$

Der Parser arbeitet bei Eingabe von abc so:

$$\begin{array}{llll}
 & (\epsilon, abc, \epsilon, 0) & \vdash & (\epsilon, abc, \epsilon, 1) \\
\vdash & (\epsilon, abc, \epsilon, 2) & \vdash & (\epsilon, abc, \epsilon, 3) \\
\vdash & (a, bc, \epsilon, 0) & \vdash & (a, bc, \epsilon, 1) \\
\vdash & (a, bc, \epsilon, 2) & \vdash & (a, bc, \epsilon, 3) \\
\vdash & (ab, c, \epsilon, 0) & \vdash & (ab, c, \epsilon, 1) \\
\vdash & (B, c, 1, 0) & \vdash & (B, c, 1, 1) \\
\vdash & (B, c, 1, 2) & \vdash & (B, c, 1, 3) \\
\vdash & (Bc, \epsilon, 1, 0) & \vdash & (Bc, \epsilon, 1, 1) \\
\vdash & (Bc, \epsilon, 1, 2) & \vdash & (Bc, \epsilon, 1, 2) \\
\vdash & (BC, \epsilon, 21, 0) & \vdash & (A, \epsilon, 021, 0) \quad \text{(1.Analyse)} \\
\vdash & (A, \epsilon, 021, 1) & \vdash & (A, \epsilon, 021, 2) \\
\vdash & (A, \epsilon, 021, 3) & \vdash & (A, \epsilon, 021, \mathit{redo}) \\
\vdash & (BC, \epsilon, 21, 1) & \vdash & (BC, \epsilon, 21, 2)
\end{array}$$

Beispiel zum Bottom-Up-Parser
. . . und weitersuchen

$$\begin{array}{llll}
\vdash & (BC, \epsilon, 21, 3) & \vdash & (BC, \epsilon, 21, \mathit{redo}) \\
\vdash & (Bc, \epsilon, 1, 3) & \vdash & (Bc, \epsilon, 1, \mathit{redo}) \\
\vdash & (abc, \epsilon, \epsilon, 2) & \vdash & (abC, \epsilon, 2, 0) \\
\vdash & (abC, \epsilon, 2, 1) & \vdash & (abC, \epsilon, 2, 2) \\
\vdash & (abC, \epsilon, 2, 3) & \vdash & (abC, \epsilon, 2, \mathit{redo}) \\
\vdash & (abc, \epsilon, \epsilon, 3) & \vdash & (abc, \epsilon, \epsilon, \mathit{redo}) \\
\vdash & (ab, c, \epsilon, \mathit{redo}) & \vdash & (a, bc, \epsilon, \mathit{redo}) \\
\vdash & (\epsilon, abc, \epsilon, \mathit{redo}) & \vdash & (\epsilon, abc, \epsilon, \mathit{stop})
\end{array}$$

Nachteile

Divergenz bei Zyklen $A \Rightarrow^+ A$
Divergenz bei Löschregeln $A \Rightarrow \epsilon$

Der Parser divergiert z.B., wenn es eine Löschregel $p : (A \to \epsilon)$, aber keine andere Löschregel und keine Regel $B \to \beta A$ gibt:

$$\begin{array}{llll}
 & (\gamma, w, \pi, p) & \vdash & (\gamma A, w, p\pi, 0) \\
\vdash & (\gamma A, w, p\pi, 1) & \vdash & (\gamma A, w, p\pi, 2) \\
\vdash & \ldots & \vdash & (\gamma A, w, p\pi, p) \\
\vdash & (\gamma AA, w, pp\pi, 0) & \vdash & \ldots
\end{array}$$

c) Bottom-Up-Parser in Prolog
(mit Rücksetzungen, für DCG-Regeln)

Ein Bottom-Up-Parser für DCGs spezialisiert beim Reduzieren die rechte Seite (zusammengesetzte Kategorie) der Regel zu einer Folge einfacher Kategorien und entfernt diese vom Stapel:

⟨*bottom-up.pl*⟩≡

```
parse(Ws,Tree) :-
    parse(Ws,[],[Tree]),
    Tree = [S|_], start_symbol(S).

parse([W|Ws],Stack,Reduced) :-
    ( reduce(Stack,RedStack),
        parse([W|Ws],RedStack,Reduced)
    ; shift(W,Stack,WStack),
        parse(Ws,WStack,Reduced) ).
parse([],Stack,Reduced) :-
    ( reduce(Stack,RedStack),
        parse([],RedStack,Reduced)
    ; Reduced = Stack ).

shift(W,Stack,[Tree|Stack]) :-
    ( word(W,A), Tree = [A,W]
    ; A = [W], Tree = [A] ).
reduce(Stack,Reduced) :-
    (B ---> Beta),
    roots(Beta,Roots),
    split(Stack,Roots,BetaForest,AlphaForest),
    reverse(BetaForest,BetaTrees),
    Tree = [B|BetaTrees],
    Reduced = [Tree|AlphaForest].
```

⟨*roots.pl*⟩

Die Spezialisierung der rechten Regelseite erfolgt durch eine Verallgemeinerung von `tupleToList/2`:

⟨*roots.pl*⟩≡

```
roots((A,B),Cats) :-
    !, roots(A,CatsA), roots(B,CatsB),
    append(CatsA,CatsB,Cats).
roots((A;B),Cats) :-
    !, (roots(A,Cats) ; roots(B,Cats)).
roots([W|Ws],Cats) :-
    !, roots(Ws,CatsWs), Cats = [[W]|CatsWs].
roots([],Cats) :-
    !, Cats = [].
roots({Code},Cats) :-
    !, call(Code), Cats = [].
roots(Cat,[Cat]).
```

Beim Aufspalten des `Forest` belegt `append` die unbekannten Teilbäume in `BetaForest`:

⟨*bottom-up.pl*⟩+≡

```
split(Forest,Beta,BetaForest,AlphaForest) :-
    reverse(Beta,BetaRev),
    newTrees(BetaRev,BetaForest),
    append(BetaForest,AlphaForest,Forest).

newTrees([A|As],[[A|_]|Ts]) :-
    newTrees(As,Ts).
newTrees([],[]).
```

⟨*Bem. unvollständige Behandlung von Differenzkategorien*⟩≡

```
roots((A-B),Cats) :-
    !, roots(A,Cats), not(roots(B,Cats)).
```

Das verbietet leider nicht, daß die Eingabe eine `A`-Phrase ist, aber auch eine `B`-Phrase mit *anderen* direkten Konstituenten.

Beispiel: Bottom-Up-Parsen

⟨*cfg8.grm*⟩≡

```
start_symbol('A').
:- op(1100,xfx,'--->').
'A' ---> 'B', 'C'.
'B' ---> [a], 'B'.
'C'---> 'A', 'B'.
word(b,'B').
word(c,'C').
```

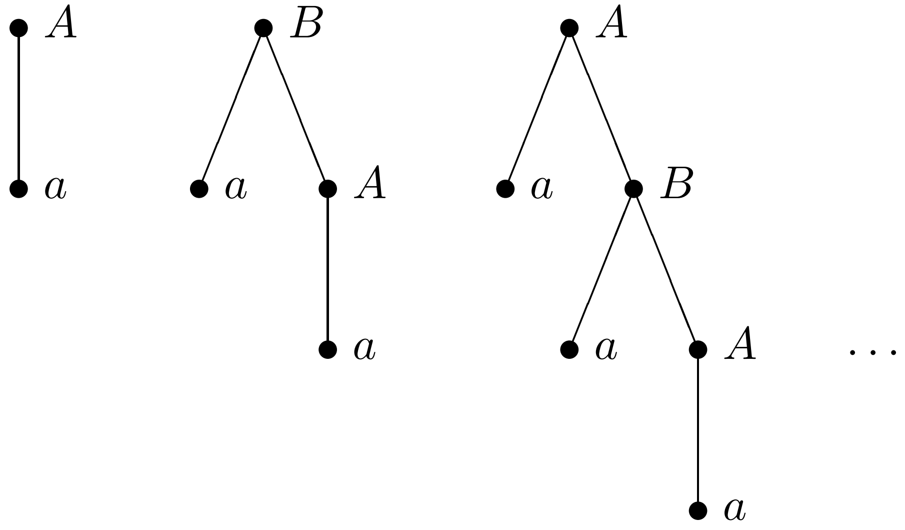
⟨*Analyse:* ⟩≡

```
?- ['cfg8.grm', 'sr.nd.pl'],
   trace([shift,reduce],[exit]).

?- parse([a,b,c],Tree), printTree(Tree,8).
Exit (10) shift(a, [], [[[a]]])
Exit (11) shift(b, [[[a]]], [['B', b], [[a]]])
Exit (12) reduce([['B', b], [[a]]],
                  [['B', [[a]], ['B', b]]])
Exit (13) shift(c, [['B', [[a]], ['B', b]]],
                 [['C', c], ['B', [[a]], ['B', b]]])
Exit (14) reduce([['C', c], ['B', [[a]], ['B', b]]],
                  [['A', ['B', [[a]], ['B', b]],
                         ['C', c]]])
        'A'
           'B'
              a
              'B'
                 b
           'C'
              c

Tree = ['A', ['B', [[a]], ['B', b]], ['C', c]]
```

Aufwand der Bottom-Up-Analyse mit Rücksetzungen

Es gibt kontextfreie Grammatiken G, bei denen jedes $w \in L(G)$ nur eine Analyse hat, der Aufwand zu ihrer Berechnung aber $O(2^{|w|})$ Schritte erfordert.

Beisp. 11 G habe die Regeln `0:(A ---> a) , 1:(A ---> aB), 2:(B ---> aA)` . Für $n = 1, 2, 3, \ldots$ hat a^n die Analyse



Braucht die deterministische Bottom-Up-Analyse $f(2n+1)$ Schritte zur Analyse von a^{2n+1}, so braucht die von a^{2n+2} mindestens $3 \cdot f(2n+1)$ Schritte:

$$\begin{array}{llll}
(\epsilon, a^{2n+1}a) & \vdash^{f(2n+1)} & (A, a) & \\
& \vdash^{\geq f(2n+1)} & (a^{2n+1}, a) & \text{(redo)} \\
& \vdash & (aa^{2n+1}, \epsilon) & \\
& \vdash^{\geq f(2n+1)} & (aA, \epsilon) & \\
& \vdash & (B, \epsilon). &
\end{array}$$

Daher ist $f(2n+2) \simeq 3 \cdot f(2n+1)$ und analog $f(2n+1) \simeq 3 \cdot f(2n)$, also $f(n) \simeq 3^n$.

c') Bottom-Up-Parser in Prolog
(mit Speichern der Wälder)

In `forests(I,Roots,Forest)` werden die Baumfolgen `Forest` mit Wurzelmarken `Roots` gesammelt, die den Teil $a_1 \cdots a_i$ der Eingabe $a_1 \cdots a_n$ abdecken.

Wälder und ihre Wurzelfolgen werden in invertierter Reihenfolge angegeben, damit man mit Listenoperationen auf die Enden zugreifen kann.

⟨*bottom-up.forests.pl*⟩≡
```
% Voraussetzung: CFG-Regeln, Lexikon abgetrennt.
:- dynamic forests/3.

parse(Ws, Tree) :-
        retractall(forests(_,_,_)),
        add(forests(0,[],[])),
        reduce(0,Ws),
        start_symbol(S),
        length(Ws,N),
        forests(N,[S],[Tree]).

shift(I,[W|Ws]) :-
        word(W,A),
        forests(I,Roots,Forest),
        J is I + 1,
        add(forests(J,[A|Roots],[[A,W]|Forest])),
        fail.
shift(I,[W|Ws]) :-
        J is I + 1, reduce(J,Ws).
shift(I,[]).
```

Bottom-Up-Parser (Forts.)
(mit Speichern der Wälder)

⟨*bottom-up.forests.pl*⟩+≡
```
reduce(I,_Ws) :-
    forests(I,Roots,Forest),
    reduce(I,Roots,Forest),
    fail.
reduce(I,Ws) :-
    shift(I,Ws).

reduce(I,Roots,Forest) :-
    (B ---> Beta),
    roots(Beta,Bs),
    reverse(Bs,BetaRev),
    append(BetaRev,AlphaRev,Roots),
    length(BetaRev,N),
    length(BetaForest,N), % N unbekannte Baeume
    append(BetaForest,AlphaForest,Forest),
    reverse(BetaForest,BetaTrees),
    NewForest = [[B|BetaTrees]|AlphaForest],
    NewRoots = [B|AlphaRev],
    add(forests(I,NewRoots,NewForest)),
    reduce(I,NewRoots,NewForest),
    fail.
reduce(I,_Roots,_Forest).

add(T) :- (known_instance(T) -> true ; assertz(T)).
known_instance(Term) :-
    \+ \+ (numbervars(Term,const,0,_), Term).
```

⟨*roots.pl*⟩

Vergleich des Aufwands am Beispiel

⟨*cfg9.grm*⟩≡

```
start_symbol('A').
:- op(1100,xfx,'--->').
'A' ---> a, 'B'.
'B' ---> a, 'A'.
'A' ---> a.
word(a,a).
```

Das Speichern der Wälder erspart es zwar, die Bäume abzubauen, aber mit Abspeichern und Suchen dauert es länger:

⟨*Aufwand mit* `sr.pl`*, ohne* `explain`*-Teile*⟩≡

```
?- time(parse([a,a,a,a,a,a,a,a,a],Baum)).
% 277,648 inferences, 2.56 CPU in 2.56 seconds
  (100% CPU, 108456 Lips)
```

⟨*Aufwand mit sr.nd.pl*⟩≡

```
?- time(parse([a,a,a,a,a,a,a,a,a],Baum)).
% 1,702,425 inferences, 2.01 CPU in 2.01 seconds
  (100% CPU, 846978 Lips)
```

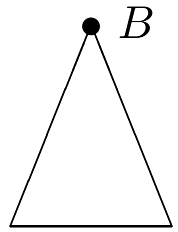
Zwar hat jede Eingabe nur eine Analyse, aber es gibt viele unvollständige Analysen, die auch gespeichert werden.

Bem. Man kann Bäume kompakter darstellen, indem man Bäume mit gleichen Blättern und gleicher Wurzel an den Blättern und der Wurzel identifiziert. Analog für Wälder.

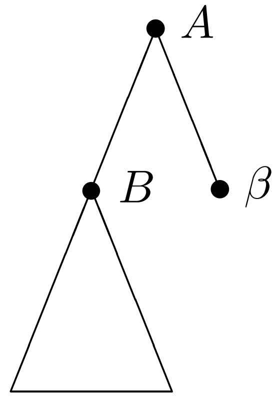
Nichtdeterministische Left-Corner-Analyse

Die (bottom-up) Left-Corner-Analyse mit der (top-down) Erwartung S erweitert partielle Analysen auf zwei unterschiedliche Weisen:

(i) Falls ein Teilwort $a_{i+1} \cdots a_j$ der Eingabe durch

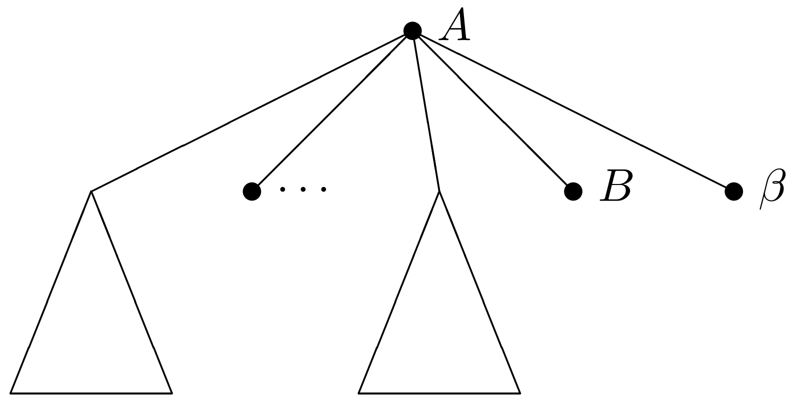


als Ausdruck der Kategorie B analysiert und B die *linke Ecke* einer Grammatikregel $(A \to B\beta) \in P$ ist, so wird die Teilanalyse mit dieser Regel fortgesetzt zu

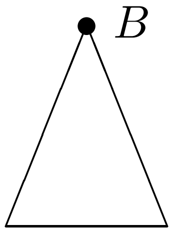


und die Kategorien von β bilden neue Erwartungen für die Resteingabe ab a_{j+1}.

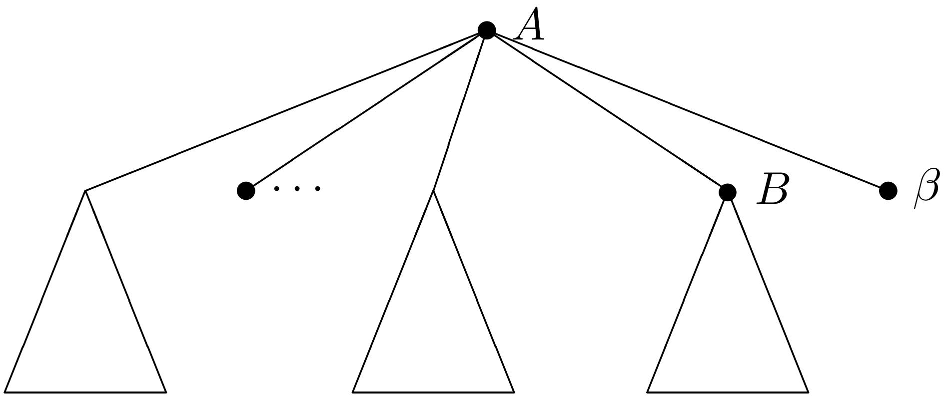
(ii) Falls B die nächste Erwartung einer partiellen Analyse



von $a_{i+1} \cdots a_k$ ist, wird sie um eine vollständige Analyse



von $a_{k+1} \cdots a_j$ der Kategorie A zu einer partiellen Analyse von $a_{i+1} \cdots a_j$ fortgesetzt:



(iii) Durch Lesen des nächsten Eingabesymbols a erhält man eine vollständige Analyse, $\bullet\ a$, eines Teilworts.

a) Left-Corner-Erkenner
(nicht-det. Kellermaschine)

Eine *gepunktete Regel* ist Grammatikregel mit einem Trennpunkt auf der rechten Regelseite. Die Menge

$$\dot{P} := \{\, (A \to \alpha \cdot \beta) \mid (A \to \alpha\beta) \in P \,\}$$

enthält die gepunkteten Regeln zu den Regeln von P.

Prinzip: (i) Lies ein Eingabesymbol a, (ii) reduziere ϵ, (iii) erweitere die vollständige Analyse eines B-Ausdrucks um eine Regel $(A \to B\beta)$ mit linker Ecke B, oder (iv) erweitere eine partielle Analyse mit Erwartung B für eine Position um die vollständige Analyse eines B-Ausdrucks an dieser Position.

Konfigurationen:	$(\gamma, w) \in (Cat \cup \dot{P})^* \times \Sigma^*$
Anfangskonfiguration:	$(\epsilon, a_1 \cdots a_n)$
Endkonfiguration:	(S, ϵ)
Befehle:	
Eingabe lesen:	$(\gamma, aw) \vdash_a (\gamma a, w)$
ϵ reduzieren:	$(\gamma, w) \vdash_{(A\to\epsilon)} (\gamma A, w)$
Partiell reduzieren:	$(\gamma B, w) \vdash_{(A\to B\beta)} (\gamma(A \to B \cdot \beta), w)$
Vervollständigen:	$(\gamma(A \to \alpha \cdot B\beta)B, w)$
	$\vdash (\gamma(A \to \alpha B \cdot \beta), w)$
Vervollständigen:	$(\gamma(A \to \alpha \cdot), w) \vdash (\gamma A, w)$

$(A \to \alpha \cdot \beta)$ oben auf dem Stapel bedeutet: beim Versuch, einen A-Ausdruck zu erkennen, wurden Ausdrücke der Kategorien α erkannt und Ausdrücke der Kategorien β werden noch gesucht.

Beispiel zur Left-Corner-Erkennung

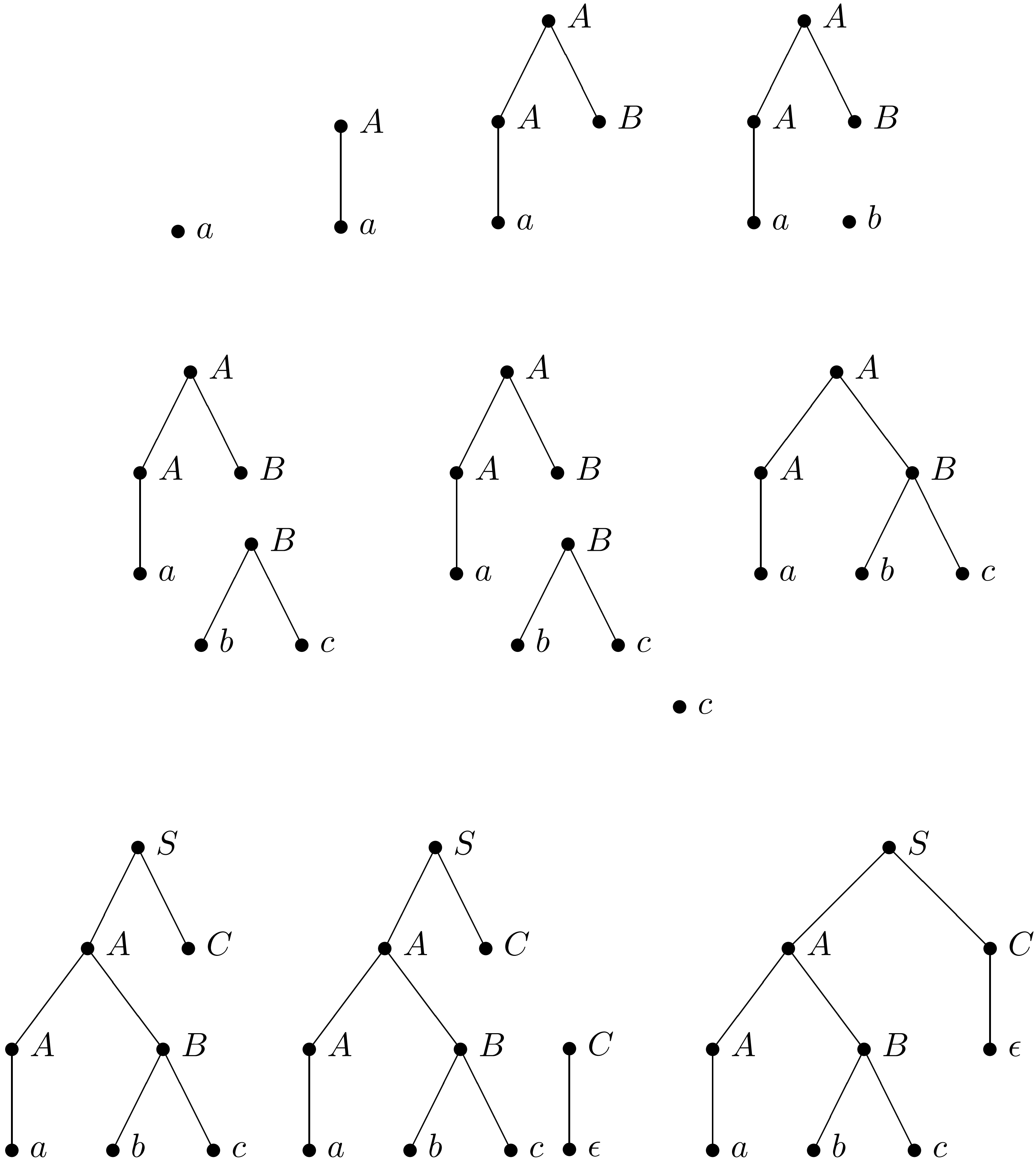
Für die Grammatik mit den Regeln

$$(S \to AC), \quad (A \to AB), \quad (A \to a), \quad (B \to bc), \quad (C \to \epsilon)$$

und die Eingabe *abc* ist dies eine akzeptierende Berechnung:

$$\begin{array}{lll}
(\epsilon, abc) & \vdash_a & (a, bc) \\
& \vdash_{(A \to a)} & ((A \to a\cdot), bc) \\
& \vdash & (A, bc) \\
& \vdash_{(A \to AB)} & ((A \to A \cdot B), bc) \\
& \vdash_b & ((A \to A \cdot B)b, c) \\
& \vdash_{(B \to bc)} & ((A \to A \cdot B)(B \to b \cdot c), c) \\
& \vdash & ((A \to A \cdot B)(B \to b \cdot c)c, \epsilon) \\
& \vdash & ((A \to A \cdot B)(B \to bc\cdot), \epsilon) \\
& \vdash & ((A \to A \cdot B)B, \epsilon) \\
& \vdash & ((A \to AB\cdot), \epsilon) \\
& \vdash & (A, \epsilon) \\
& \vdash_{(S \to AC)} & ((S \to A \cdot C), \epsilon) \\
& \vdash_{(C \to \epsilon)} & ((S \to A \cdot C)C, \epsilon) \\
& \vdash & ((S \to AC\cdot), \epsilon) \\
& \vdash & (S, \epsilon)
\end{array}$$

Aufbau des Syntaxbaums (Left-Corner-Analyse)



c) Left-Corner-Parser (mit Rücksetzungen)

Aufg. 1 Definiere aus dem nicht-deterministischen Left-Corner-Erkenner einen deterministischen Left-Corner-Parser (mit Rücksetzungen).

Frage 1 Terminiert der Left-Corner-Parser (mit Rücksetzungen) bei jeder kontextfreien Grammatik?

c) Leftcorner-Parser in Prolog
(für DCGs, mit Rücksetzungen)

Der Parser benutzt wie der Erkenner einen Stapel mit gepunkteten Regeln ($A \to \alpha \cdot \beta$), aber merkt sich statt der erkannten Kategorien α die (invertierte) Liste entsprechender Bäume.

$\langle$*leftcorner.pl*$\rangle\equiv$

```
parse(Words,Tree) :-
        parse(Words,[],[Tree]),
        Tree = [S|_], start_symbol(S).

parse([W|Words],Stack,Reduced) :-
        ( ( complete(Stack,RedStack)
          ; predict(Stack,RedStack)
          ), parse([W|Words],RedStack,Reduced)
        ; shift(W,Stack,RedStack),
            parse(Words,RedStack,Reduced)
        ).
parse([],Stack,Reduced) :-
        ( ( complete(Stack,RedStack)
          ; predict(Stack,RedStack)
          ), parse([],RedStack,Reduced)
        ; Reduced = Stack ).

shift(W,Stack,[Tree|Stack]) :-
        ( word(W,A), Tree = [A,W]
        ; A = [W], Tree = [A] ).
```

Left-Corner-Parser in Prolog (Forts.)

Eine partielle Analyse mit Erwartungskategorie kann durch einen passenden Ausdruck vervollständigt werden:

⟨*leftcorner.pl*⟩+≡
```
complete([[B|Trees],(C ---> RevTrees,[B|Beta])|Stack],
         [(C ---> [[B|Trees]|RevTrees],Beta)|Stack]).
```

Eine partielle Analyse ohne Erwartungskategorie kann in eine vollständige Analyse umgewandelt werden:

⟨*leftcorner.pl*⟩+≡
```
complete([(C ---> RevTrees,[])|Stack],
         [[C|Trees]|Stack]) :-
        reverse(RevTrees,Trees).
```

Das leere Wort am Anfang der Resteingabe kann zu einem Ausdruck vervollständigt werden:

⟨*leftcorner.pl*⟩+≡
```
complete(Stack,[[C]|Stack]) :-
        (C ---> []).
```

Ein gefundener Ausdruck kann als linke Ecke einer partiellen Analyse der Resteingabe dienen:

⟨*leftcorner.pl*⟩+≡
```
predict([[B|Trees]|Stack],Reduced) :-
        (A ---> Alpha),
        roots(Alpha,[B|Beta]),
        Reduced = [(A ---> [[B|Trees]],Beta)|Stack].
```

⟨*roots.pl*⟩

DCG-Regeln ersetzt `roots/2` quasi in mehrere CFG-Regeln.

Beispiel zum Left-Corner-Parser

⟨*cfg.lc.grm*⟩≡
```
start_symbol(s).

% Grammatikregeln:
:- op(1100,xfx,'--->').
s ---> ap,cp.
ap ---> ap,bp.

% Lexikon:
word(a,ap).  word(b,bp).  word(c,cp).
```

⟨*Trace*⟩≡
```
?- trace(parse(_,_,_),[all]).
?- parse([a,b,c],Tree).
 (8) parse([a,b,c], [], [Tree])
 (9) parse([b,c], [[ap,a]], [Tree])
 (10) parse([b,c], [(s--->[[ap,a]],[cp])], [Tree])
 ... Fail: (10) parse([b,c], [(s---> ..)], [Tree])
 (10) parse([b,c],[(ap--->[[ap,a]],[bp])], [Tree])
 (11) parse([c],[[bp,b], (ap--->[..],[bp])],[Tree])
 (12) parse([c],[(ap--->[[bp,b],[ap,a]],[])],[Tree])
 (13) parse([c],[[ap,[ap,a],[bp,b]]], [Tree])
 (14) parse([c],[(s--->[[ap,..]],[cp])],[Tree])
 (15) parse([],[[cp,c],(s--->[[ap,..]],[cp])],[Tree])
 (16) parse([],[(s--->[[cp,c],[ap,..]],[])],[Tree])
 (17) parse([],[[s,[ap,[ap,a],[bp,b]],[cp,c]]],[Tree])
 (17) parse([],[[s,[ap,[ap,a],[bp,b]],[cp,c]]],
                [[s,[ap,[ap,a],[bp,b]],[cp,c]]])
 (16) .. (8) parse([a,b,c],[],
                    [[s,[ap,[ap,a],[bp,b]],[cp,c]]])

Tree = [s, [ap, [ap,a], [bp,b]], [cp,c]] ;
```

d) Leftcorner-Erkenner in Prolog (mit Rücksetzungen)

Manchmal schreibt man den Parser mit Prädikaten `parse/3` und `leftcorner/4` so, daß bei der Eingabe $a_1 \cdots a_n$ gilt:

$$\begin{aligned} \mathit{parse}(A,i,j) \quad &\Longleftrightarrow \quad A \Rightarrow^* a_{i+1} \cdots a_j \\ \mathit{leftcorner}(B,A,i,j) \quad &\Longleftrightarrow \quad A \Rightarrow^* B a_{i+1} \cdots a_j \end{aligned}$$

⟨*leftcorner.shieber.pl*⟩≡

```
parse(Words) :-
        start_symbol(S), parse(S,Words,[]).
parse(A,I,J) :-
        scan(B,I,K),
        leftcorner(B,A,K,J).

scan(B,I,K) :-
        ( (B ---> []), I=K
        ; (word(W,B) ; B=[W]), I = [W|K] ).
leftcorner(B,A,I,J) :-
        ( B=A, I=J
        ; (C ---> Gamma),
            roots(Gamma,[B|Beta])
            parse_list(Beta,I,K),
            leftcorner(C,A,K,J)
        ).
parse_list([B|Bs],I,J) :-
        parse(B,I,K), parse_list(Bs,K,J).
parse_list([],I,J) :- I = J.
```

⟨*roots.pl*⟩

Nichtdeterministische Head-Corner-Analyse

In sog. „$\bar{X}$-Regeln" `(XP ---> ... X ...)` soll die Hauptkonstituente `X` die Kategorien (ggf.: und Anordnungen) der Nebenkonstituenten ... bestimmen, z.B. das Verb seine Objekte.

In jeder Grammatikregel sei eine *Hauptkonstituente* bestimmt.

⟨*Regelformat*⟩≡

```
(SynCat ---> Cat1,...,Cati,...,Catn).
```

Wir definieren die Hauptkonstituente `B` durch ein Prädikat:

⟨*Hauptrelation*⟩≡

```
head((SynCat ---> Alpha,C,Beta),Alpha,B,Beta).
```

Beim Head-Corner-Parser soll eine erkannte Phrase nicht als linke Ecke, sondern als Kopfkonstituente verwendet werden.

Vorteil

(i) Linguistisch attraktiv: Kopf-Komplement-Schema

(ii) Während die Kategorie der linken Ecke von der Anordnung abhängt, ist die Position des Kopfs (meist) fest, die Kopfrelation also unabhängig von Argumentstellungen berechenbar. (Daher auch für Metaregeln geeignet.)

Nachteil

(i) Kompliziert zu implementieren: Parserichtung rechts-links im Linkskontext, links-rechts im Rechtskontext des Kopfs

(ii) Praktische Effizienz?

a) Head-Corner-Erkenner in Prolog

Man definiert induktiv die Relation

$$\begin{aligned} &\mathit{headcorner}(B, A, i, i', j', j) \ :\Longleftrightarrow \\ &\quad A \Rightarrow^* a_{i+1} \cdots a_{i'} B a_{j'+1} \cdots a_j. \end{aligned}$$

⟨*headcorner-rec.pl*⟩≡

```
parse(A,I,J) :-
    (head(B,A) ; B=A),
    leaf(B,I,I2,J2,J),
    headcorner(B,A,I,I2,J2,J).

headcorner(B,A,I,I2,J2,J) :-            %      A
    ( I=I2, J2=J, B=A                   %    /   \
    ; (C ---> Gamma),                   %   /     \
      head((C ---> Gamma),Alpha,B,Beta),% i--- C ---j
      reverse(Alpha,AlphaRev),          %     / \
      parse_list_rev(AlphaRev,I,I3,I2), %    /   \
      parse_list(Beta,J2,J3),           %   /     \
      headcorner(C,A,I,I3,J3,J)        % i3-i2 B j2-j3
    ).
```

Mit `leaf/5` sucht man im Teilwort $a_{i+1} \cdots a_j$ nach möglichen lexikalischen Köpfen $a_{i'+1} = a_{j'}$:

⟨*headcorner-rec.pl*⟩+≡

```
leaf(A,I,I2,J2,J) :-
     ( (A ---> []),
         I=I2, I2=J2, J2=J
     ; append(_,I2,I), I2 = [W|J2], append(_,J,J2),
        (word(W,A) ; A = [W])
     ).
```

Die Grammatik bestimmt die Hauptkonstituente(n) einer Regel, hier grob durch `head/2` allein aus der Phrasenkategorie:

⟨*headcorner-rec.pl*⟩+≡
```
head((C ---> Gamma),Alpha,B,Beta) :-
      roots(Gamma,GammaList),
      head(B,C),
      append(Alpha,[B|Beta],GammaList).
```

In `parse_list_rev(AlphaRev,+I,-K,+J)` sucht man `K` zwischen `I` und `J` mit `parse(Alpha,K,J)`:

⟨*headcorner-rec.pl*⟩+≡
```
parse_list_rev([],_,J,J).
parse_list_rev([A|As],I,K,J) :-
        parse_rev(A,I,L,J),
        parse_list_rev(As,I,K,L).

parse_rev(A,I,K,J) :-
        append(_,K,I), append(_,J,K),
        parse(A,K,J).

parse_list([B|Bs],I,J) :-
        parse(B,I,K), parse_list(Bs,K,J).
parse_list([],I,J) :- I = J.
```

⟨*roots.pl*⟩

Aufg. 2 Erweitere diesen Erkenner zu einem Parser, d.h. ergänze die Ausgabe eines Syntaxbaums.

Beispiel zur Head-Corner-Erkennung

⟨*cfg.hc.grm*⟩≡

```
start_symbol(s).
start_symbol(nom).

% Grammatikregeln:
:- op(1100,xfx,'--->').
s ---> nom, v, akk, rs.
s ---> nom, v, akk.
rs ---> rnom, akk, v.
rs ---> rakk, nom, v.

nom ---> det(nom), n(nom)).
akk ---> det(akk), n(akk)).

% Hauptrelation:
head(v,s).              head(v,rs).
head(n(nom),nom).       head(n(akk),akk).

% Lexikon:
word(er,nom).           word(ihn,akk).
word(der,det(nom)).     word(den,det(akk)).
word('Mann',n(nom)).    word('Mann',n(akk)).
word(der,rnom).         word(den,rakk).
word(kennt,v).          word(grüßt,v).
```

Bem. Zur Beschleunigung sollte man die transitive Hülle von `{(B,C)| head((C ---> _),_,B,_)}` vorberechnen und in `parse/3` ausnutzen.

Beispiel zur Head-Corner-Erkennung (Forts.)

⟨*Tracer einschalten*⟩≡

```
100 ?- trace([headcorner,parse],[+call,+exit]).
```

⟨*Eingabesatz*⟩≡

```
101 ?- parse(s,[der,'Mann',kennt,
                den,'Mann',der,ihn,grüßt],[]).
```

⟨*Parse-Aufruf*⟩≡

```
 C (7) parse(s, [der,'Mann',...], [])
 C (8) headcorner(v, s,
           [der,'Mann',...], [kennt,..],[..], [])
 E (9) s--->nom, v, akk, rs
```

⟨*Linkskontext* `nom` *von* `v` *aus (9)* ⟩≡

```
 C (11) parse(nom, [der,'Mann',kennt,...],
                                  [kennt,...])
 C (12) headcorner(n(nom),nom,
            [der,'Mann',kennt,...],
            ['Mann',..],[..], [kennt,...])
 E (13) ... E (13) nom--->det(nom), n(nom)
 C ... E (15) parse(det(nom), [der,'Mann',...],
                                      ['Mann',...])
 C E (13) headcorner(nom, nom,
              [der,'Mann',kennt,...],
              [der,'Mann',..],[..], [kennt, ...])
 E (12) headcorner(n(nom),nom,
            [der,'Mann',kennt,...],
            ['Mann',..],[..], [kennt,...])
 E (11) parse(nom, [der,'Mann',kennt,...],
                                  [kennt,...])
```

Beispiel zur Head-Corner-Erkennung (Forts.)

⟨*Teil* `akk` *des Rechtskontexts* `akk,rs` *von* `v` *aus (9)* ⟩≡

```
C (10) parse(akk, [den,'Mann',der,ihn,grüßt],_G933)
... (analog)
E (10) parse(akk, [den,'Mann',der,ihn,grüßt],
                                    [der,ihn,grüßt])
```

⟨*Teil* `rs` *des Rechtskontexts* `akk,rs` *von* `v` *aus (9)* ⟩≡

```
C (11) parse(rs, [der, ihn, grüßt], _G972)
C (12) headcorner(v, rs,
           [der,ihn,grüßt], [grüßt],[], [])
E (13) ...
E (13) rs--->rnom, akk, v
```

⟨*Teil* `akk` *des Linkskontexts* `rnom,akk` *von* `v` *aus (13)* ⟩≡

```
C (15) parse(akk, [der,ihn,grüßt], [grüßt])
F ...
C (15) parse(akk, [ihn,grüßt], [grüßt])
E (15)
```

⟨*Teil* `rnom` *des Linkskontexts* `rnom,akk` *von* `v` *aus (13)* ⟩≡

```
C (16) parse(rnom, [der,ihn,grüßt], [ihn,grüßt])
C (17) headcorner(rnom, rnom,
          [der,ihn,grüßt],[der,..],[..],[ihn,grüßt])
E (17)
E (16) parse(rnom, [der,ihn,grüßt], [ihn,grüßt])
```

⟨*Teil* `rs` *des Rechtskontexts* `akk,rs` *von* `v` *aus (9)* ⟩+≡

```
C (13) headcorner(rs, rs,
        [der,ihn,grüßt], [der,ihn,grüßt],[], [])
E (13)
```

Beispiel zur Head-Corner-Erkennung (Ende)

⟨*Rückgabe der Ergebnisse*⟩≡

```
   E (12) headcorner(v, rs,
                     [der,ihn,grüßt], [grüßt],[], [])
   E (11) parse(rs, [der,ihn,grüßt], [])

   C (9) E (9) headcorner(s, s,
          [der,'Mann',...], [der,'Mann',...],[], [])
   E (8) headcorner(v, s,
          [der,'Mann',kennt,...], [kennt,...],[...], [])
   E (7) parse(s,
         [der,'Mann',kennt,den,'Mann',der,ihn,grüßt],
         [])
  Yes
```

b) Head-Corner-Parser in Prolog
(für DCGs, mit Rücksetzen)

In den Erkenner bauen wir eine Ausgabe der Syntaxbäume ein. `leaf/5` gibt im ersten Argument einen Baum aus:

⟨*headcorner.pl*⟩≡

```
parse(Ws,Tree) :-
    start_symbol(S), parse(S,Ws,[],Tree).
parse(A,I,J,Tree) :-
    (head(B,A) ; B=A),
    leaf([B|Trees],I,I2,J2,J),
    headcorner([B|Trees],A,I,I2,J2,J,Tree).
leaf([A|Trees],I,I2,J2,J) :-
    ( (A ---> []),
        I=I2, I2=J2, J2=J, Trees=[]
    ; append(_,I2,I), I2=[W|J2], append(_,J,J2),
        (word(W,A), Trees=[W] ; A=[W], Trees=[])
    ).
```

`headcorner/7` setzt eine Kopfkonstituente nach oben fort:

⟨*headcorner.pl*⟩+≡

```
headcorner([B|BTs],A,I,I2,J2,J,Tree) :-
    ( I=I2, J2=J, B=A, Tree=[B|BTs]       %       A
    ; (C ---> Gamma),                     %     /   \
      head((C ---> Gamma),Alpha,B,Beta),  %    /  ?  \
      reverse(Alpha,AlphaRev),            %  i-?- C -?-j
      parse_list_rev(AlphaRev,I,I3,I2,AlphaRs),%/ | \
      reverse(AlphaRs,AlphaTs),           % Alpha B Beta
      parse_list(Beta,J2,J3,BetaTs),      % /  | / \ |  \
      append(AlphaTs,[[B|BTs]|BetaTs],Ts),%i3-i2 BTs j2-j3
      headcorner([C|Ts],A,I,I3,J3,J,Tree)
    ).
```

Die Grammatik bestimme die Hauptkonstituente(n) einer Regel wieder grob durch `head/2` allein aus der Phrasenkategorie:

⟨*headcorner.pl*⟩+≡

```
head((C ---> Gamma),Alpha,B,Beta) :-
    roots(Gamma,GammaList),
    head(B,C),
    append(Alpha,[B|Beta],GammaList).
```

Bei `parse_list` und `parse_list_rev` gibt man jetzt eine Liste von Bäumen aus:

⟨*headcorner.pl*⟩+≡

```
parse_list_rev([],_,J,J,[]).
parse_list_rev([A|As],I,K,J,[Tree|Trees]) :-
    parse_rev(A,I,L,J,Tree),
    parse_list_rev(As,I,K,L,Trees).

parse_rev(A,I,K,J,Tree) :-
    append(_,K,I), append(_,J,K),
    parse(A,K,J,Tree).

parse_list([B|Bs],I,J,[Tree|Trees]) :-
    parse(B,I,K,Tree), parse_list(Bs,K,J,Trees).
parse_list([],I,J,[]) :- I = J.
```

⟨*roots.pl*⟩

Man könnte mit einem besseren `roots/2` noch die Grammatikregeln nach gemeinsamen Kopfkonstituenten faktorisieren.

Beispiel zur Head-Corner-Analyse

Die Reihenfolge der Verwendung lexikalischer Elemente ist:

⟨*Beispiel*⟩≡

```
?- ['cfg.hc.grm', headcorner], trace(leaf,[exit]).
?- parse(s,[der,'Mann',kennt,den,'Mann',der,ihn,grüßt],
         [],Tree), printTree(Tree,2).
E (9) leaf([v,kennt],[der,..],[kennt,..],[den,..],[])
E (13) leaf([n(nom),'Mann'],[der,..],['Mann'..],
                                [kennt,..],[kennt,..])
E (17) leaf([det(nom),der],[der,..],[der,..],
                              ['Mann',..],['Mann',kennt,..])
E (12) leaf([n(akk),'Mann'],[den,..],['Mann',der,..],
                                [der,ihn,..],[der,..])
E (16) leaf([det(akk),den],[den,..],[den,..],
                                ['Mann',der,..],['Mann',..])
E (13) leaf([v,grüßt],[der,ihn,grüßt],[grüßt],[],[])
E (17) leaf([akk,ihn],[der,ihn,..],[ihn,..],[g..],[g..])
E (17) leaf([akk,ihn],[ihn,g..],[ihn,g..],[g..],[g..])
E (18) leaf([rnom,der],[der,ihn,..],[der,..],
                                [ihn,..],[ihn,..])
     s
       nom
         det(nom)  der      3.
         n(nom)  'Mann'     2.
       v  kennt             1.
       akk
         det(akk)  den      5.
         n(akk)  'Mann'     4.
       rs
         rnom  der          8.
         akk  ihn           7.
         v  grüßt           6.
```

TABELLENVERFAHREN (CHART PARSER)

Bei den Analyseverfahren mit Rücksetzungen wird ein Teilausdruck bei einer Rücksetzung i.a. nochmals analysiert. Das vermeidet man bei den *Tabellenverfahren*, indem man die Analysen von Teilausdrücken in einer Tabelle speichert und bei Bedarf dort nachschlägt.

Braucht man nur ein Erkennungsverfahren, so speichert man in der Tabelle T zu jedem Teilausdruck v der Eingabe w alle Kategorien A, die v hat, d.h. $T(v) = \{ A \mid A \Rightarrow^* v \}$.

1. Der Erkenner von Cocke, Younger, Kasami (CYK)

Vor.: Kontextfreie Grammatik $G = (N, \Sigma, S, P)$ in *Chomsky-Normalform*: alle Regeln haben die Form $(A \to a)$ oder $(A \to BC)$ mit $A, B, C \in N, a \in \Sigma$.

Eingabe: $w = a_1 \cdots a_n \in \Sigma^+$.

Methode: Konstruiere für die Positionen $0 \leq i < j \leq n$ alle *Kanten* $i \xrightarrow{A} j$ nach den Regeln:

$$\frac{a = a_{i+1}, \quad (A \to a) \in P}{i \xrightarrow{A} i+1} \quad (Scan)$$

$$\frac{i \xrightarrow{B} k, \quad k \xrightarrow{C} j, \quad (A \to BC) \in P}{i \xrightarrow{A} j} \quad (Complete)$$

Ausgabe: $0 \xrightarrow{S} n$ wird konstruiert $\iff w \in L(G)$.

Das folgt aus: $i \xrightarrow{A} j$ wird konstruiert $\iff A \Rightarrow^* a_{i+1} \cdots a_j$.

Da es nur endlich viele Kanten $i \xrightarrow{A} j$ zu w gibt, terminiert das Verfahren (bei geeigneter Verwaltung).

Das CYK-Verfahren ist ein „bottom-up"-Verfahren: die Konstruktion der Kanten beginnt bei den $i \xrightarrow{a_{i+1}} i+1$ und nutzt S nicht aus.

Programmschleife zur Konstruktion der Kanten

Die Kanten stellt man als Eintrag einer $n \times n$-Tabelle T dar:

$$T(i,j) = \{\, A \mid i \xrightarrow{A} j \,\} = \{\, A \mid A \Rightarrow^* a_{i+1} \cdots a_j \,\}.$$

Man kann die Kanten in verschiedener Reihenfolge bilden, z.B. zuerst alle Kanten $i \xrightarrow{A} k$ mit $i < k < j$, dann alle $j-1 \xrightarrow{A} j$, dann alle $j-2 \xrightarrow{A} j$, bis man alle mit $i < k \leq j$ hat, usw.:

for $j = 1$ to n do
  $T(j-1, j) := \{\, A \in N \mid (A \to a_j) \in P \,\}$;
  if $j > 1$ then
    for $i = j-2$ to 0 do
      $T(i,j) := \bigcup_{i<k<j} \{\, A \in N \mid B \in T(i,k), C \in T(k,j), (A \to BC) \in P \,\}$
      od
  od

Aufgabe: Konstruiere die Kanten in der Reihenfolge: zuerst alle Kanten der Länge 1, die $i \xrightarrow{A} i+1$, dann die der Länge 2, die $i \xrightarrow{A} i+2$, usw.

Komplexität der Erkennung mit CYK

- Für $i = 0, \ldots, n-1$, initialisiere Feld $T(i, i+1)$. Dazu muß für jede Regel geprüft werden, ob sie von der Form $A \to a_i$ ist. Das erfordert jeweils $O(|P|)$ Schritte.

  Insgesamt also $O(|w| * |P|) \leq O(|w| * |G|)$ Schritte.

- Angenommen, $T(i, j)$ ist für alle Teilwörter $a_{i+1} \ldots a_j$ der Länge $j - i < k$ (endgültig) bestimmt.

  Zum Auffüllen der $T(i', j')$ mit $|j' - i'| = k$ ist für alle Regeln $A \to BC$ und alle k' mit $i' < k' < j'$ zu prüfen, ob $B \in T(i', k')$ und $C \in T(k', j')$.

  Um $T(i', j')$ zu füllen, braucht man

  $$O(|P| \times (j' - i') \times |N|)$$

  viele Schritte (auf einer RAM), wobei höchstens $|N|$ Schritte für die Suche durch das Feld $T(i, j)$ nötig sind.

- Um alle Felder von T zu füllen, braucht man also insgesamt höchstens

$$\begin{aligned}
&O(|w| * |G|) + \sum_{i<j\leq|w|} O(|P| \times (j-i) \times |N|) \\
&\quad = \quad O(|w| * |G|) + O(|P| \times |N| \times \sum_{j\leq|w|} \frac{j(j+1)}{2}) \\
&\quad \leq \quad O(|P| \times |N| \times \frac{|w|^3}{3}) = O(|w|^3 \times |G|^2)
\end{aligned}$$

Für Regeln der Form $(A \to B_1 \cdots B_k)$ ist der Aufwand größer!

CYK-Erkenner in Prolog

Eine Kante $i \xrightarrow{A} j$ stellen wir als Prolog-Fakt `kante(I,A,J)` dar. Wir füllen die Tabelle unter der Annahme, daß alle Kanten mit Indizes $< j$ eingetragen sind, indem der Reihe nach die Felder $T(j-1,j),\ldots,T(0,j)$ gefüllt werden.

⟨*cyk-rec.cfg.pl*⟩≡

```
% Cocke/Younger/Kasami-Erkenner

parse([W|Ws]) :-
    retractall(kante(_,_,_)), % alte Tabelle loeschen
    parse(0,[W|Ws]).

parse(I,[W|Ws]) :-  % on-line Vorgehen
    scan(I,W),
    J is I+1,
    complete(I,J),  %
    parse(J,Ws).
parse(J,[]) :-
    kante(0,A,J),
    start_symbol(A).

scan(I,W) :-
    ((A ---> [W]) ; word(W,A)), % Lexikoneintrag
    J is I+1,
    add(kante(I,A,J)),
    fail.       % weitere Lexikoneintraege zu W ?
scan(_,_).
```

Eine Kante wird nur ergänzt, falls sie keine Instanz einer vorhandenen Kante ist. Dazu ersetzt man in der Kante die Variablen durch `c(0),c(1),...` und versucht, diese variablenfreie

Variante zu beweisen; geht das, ist die Kante die Instanz einer vorhandenen. Dann macht man die Belegung von Variablen rückgängig; sonst speichert man die Kante:

$\langle$*cyk-rec.cfg.pl*$\rangle{+}{\equiv}$
```
  add(Fakt) :-
      (\+ \+ (numbervars(Fakt,c,0,_), Fakt)
      -> true ; % nichts tun, falls Fakt[c/X] beweisbar
          assert(Fakt)
      ).
```

Nachdem Feld `(J-1,J)` gefüllt ist, werden die Felder `(K,J)` mit `K < J-1` gefüllt:

$\langle$*cyk-rec.cfg.pl*$\rangle{+}{\equiv}$
```
  complete(K,J) :-
      (K = 0 -> true
      ; I is K-1, complete(I,K,J), complete(I,J)
      ).

  complete(I,K,J) :- % I < K < J
      kante(I,B,K),
      kante(K,C,J),
      (A ---> B,C),
      add(kante(I,A,J)),
      fail.
  complete(I,K,J) :-
      (I = 0 -> true
      ; L is I-1, complete(L,K,J)
      ).
```

Beispiel zur Erkennung mit CYK

⟨*Beispiel mit cfg.grm*⟩≡

```
?- ['cfg.grm', 'cyk-rec.cfg.pl'].
?- trace([complete/2,complete/3],[call,exit]).
?- parse([a,b,a]), listing(kante).
 T Call: (9) complete(0, 1)
 T Exit: (9) complete(0, 1)
 T Call: (10) complete(1, 2)
 T Call: (11) complete(0, 1, 2)
 T Exit: (11) complete(0, 1, 2)
 T Call: (11) complete(0, 2)
 T Exit: (11) complete(0, 2)
 T Exit: (10) complete(1, 2)
 T Call: (11) complete(2, 3)
 T Call: (12) complete(1, 2, 3)
 T Call: (13) complete(0, 2, 3)
 T Exit: (13) complete(0, 2, 3)
 T Exit: (12) complete(1, 2, 3)
 T Call: (12) complete(1, 3)
 T Call: (13) complete(0, 1, 3)
 T Exit: (13) complete(0, 1, 3)
 T Call: (13) complete(0, 3)
 T Exit: (13) complete(0, 3)
 T Exit: (12) complete(1, 3)
 T Exit: (11) complete(2, 3)
:- dynamic kante/3.
kante(0, a(sg), 1).
kante(1, b(sg), 2).
kante(0, s(sg), 2).
kante(2, a(sg), 3).
kante(1, b(pl), 3).
kante(0, s(pl), 3).
```

2. Der Erkenner von Earley

Das Verfahren von Earley kann man als eine Verbesserung des CYK-Verfahrens ansehen: die Grammatik braucht nicht in Chomsky-Normalform zu sein, und die Analyse benutzt die Kategorie S als „Top-Down-Filter“, d.h. für die Eingabe werden nur ihre Analysen als S-Ausdruck konstruiert.

Vor.: Beliebige kontextfreie Grammatik $G = (N, \Sigma, S, P)$

Eingabe: $w = a_1 \cdots a_n \in \Sigma^*$

Methode: Konstruiere für die Positionen $0 \leq i \leq j \leq n$ alle Kanten $i \stackrel{A\to\alpha\cdot\beta}{\longrightarrow} j$, wo $(A \to \alpha\beta) \in P$, nach den Regeln:

$$\frac{(S \to \alpha) \in P}{0 \stackrel{S\to\cdot\alpha}{\longrightarrow} 0} \quad (\textit{Predict})$$

$$\frac{i \stackrel{A\to\alpha\cdot C\beta}{\longrightarrow} j, \quad (C \to \gamma) \in P}{j \stackrel{C\to\cdot\gamma}{\longrightarrow} j} \quad (\textit{Predict})$$

$$\frac{i \stackrel{A\to\alpha\cdot a\beta}{\longrightarrow} j, \quad a = a_{j+1}}{i \stackrel{A\to\alpha a\cdot\beta}{\longrightarrow} j+1} \quad (\textit{Scan})$$

$$\frac{i \stackrel{A\to\alpha\cdot C\beta}{\longrightarrow} k, \qquad k \stackrel{C\to\gamma\cdot}{\longrightarrow} j}{i \stackrel{A\to\alpha C\cdot\beta}{\longrightarrow} j} \quad (\textit{Complete})$$

Ausgabe: Eine Kante der Form $0 \stackrel{S\to\alpha\cdot}{\longrightarrow} n$ wird erzeugt

$$\iff \quad w \in L(G).$$

Zum Beweis zeigt man:

Beh.: Die Kante $i \overset{A\to\alpha\cdot\beta}{\longrightarrow} j$ wird genau dann konstruiert, wenn

(i) $S \Rightarrow^* a_1 \cdots a_i A\gamma$ für ein $\gamma \in (N \cup \Sigma)^*$, und

(ii) $\alpha \Rightarrow^* a_{i+1} \cdots a_j$.

Bew. $\Longrightarrow$ (Korrektheit der Regeln): Zeige (i), (ii) durch Induktion über die Anzahl der Schritte zur Erzeugung der Kante.

$\Longleftarrow$ (Vollständigkeit der Regeln): Für $(A \to \alpha\beta) \in P$ gelte (i) und (ii). Dann gibt es Linksableitungen

$$S \Rightarrow^{m_1} a_1 \cdots a_i A\gamma \quad \text{und} \quad \alpha \Rightarrow^{m_2} a_{i+1} \cdots a_j$$

bestimmter Längen m_1, m_2. Durch Induktion über $m_1 + m_2$ zeigt man, daß $i \overset{A\to\alpha\cdot\beta}{\longrightarrow} j$ konstruiert wird.[1]

Die Analyse von Earley ist eine Mischung aus Top-Down- und Bottom-Up-Vorgehen:

- (Predict) beginnt zu Erwartungskategorien C „top-down" neue partielle Erkennungsversuche.
- (Complete) erweitert „bottom-up" eine partielle erkannte A-Phrase durch eine vollständig erkannte C-Phrase.

Der Earley-Erkenner terminiert, da es zu jedem $w = a_1 \cdots a_n$ nur endlich viele Kanten $i \overset{A\to\alpha\cdot\beta}{\longrightarrow} j$ mit $0 \leq i \leq j \leq n$ gibt.

[1] Man braucht die Hilfsbehauptung, daß für $m_1 > 0$ ein $S \Rightarrow^{m_1 - 1} a_1 \cdots a_k A'\gamma'$ entsteht und alle $k \overset{A'\to\cdot\alpha'}{\longrightarrow} k$ erzeugt werden.

Earley-Erkenner in Prolog

(für CFGs, mit Speichern einer Tabelle `kante/5`)

Das Lexikon kann mit `word(W,Kat)` abgetrennt sein.

⟨*earley-rec.pl*⟩≡

```
:- dynamic kante/5. % kante(I,J,A,Alpha,Beta)

erkenne(Wortfolge) :-
    retractall(kante(_I,_J,_A,_Alpha,_Beta)),
    predict(0),
    erkenne(0,Wortfolge).

erkenne(I,[Wort|Rest]) :-
    J is I + 1,
    scan(I,J,Wort),
    erkenne(J,Rest).
erkenne(I,[]) :-        % Eingabe erkannt?
    start_symbol(S),
    kante(0,I,S,_Alpha,[]).

scan(I,J,Wort) :-         % i -Wort-> j=i+1
    ( word(Wort,A),
        closure(I,J,A,[Wort],[])
    ; ( kante(K,I,A,Alpha,([Wort],Beta))
      ; kante(K,I,A,Alpha,[Wort]),Beta = []
      ), append(Alpha,[[Wort]],AlphaW),
        closure(K,J,A,AlphaW,Beta)
    ), fail.
scan(_,_,_).
```

Durch `closure(I,J,A,Alpha,Beta)` werden auf $i \stackrel{A\to\alpha\cdot\beta}{\longrightarrow} j$ rekursiv die Regeln (*Complete*) und (*Predict*) anwendet, bis keine weiteren Kanten hinzukommen:

Earley-Erkenner in Prolog (Forts.)

Entsteht eine neue Kante $i \stackrel{B\to\beta\cdot}{\longrightarrow} j$, so werden „rückwärts“ alle „wartenden“ Kanten $k \stackrel{A\to\alpha\cdot B\beta}{\longrightarrow} i$ damit verbunden (*Complete*):

⟨*earley-rec.pl*⟩+≡
```
closure(I,J,B,Beta,[]) :-
    (add(kante(I,J,B,Beta,[]))
    -> complete(I,J,B) ; true).
```

Entsteht eine neue Kante $i \stackrel{A\to\alpha\cdot B\beta}{\longrightarrow} j$, so werden „vorwärts“ Kanten $j \stackrel{B\to\cdot\gamma}{\longrightarrow} j$ in die Tabelle eingetragen und bei $\gamma = \epsilon$ mit vorhandenen verkettet (*Predict*):

⟨*earley-rec.pl*⟩+≡
```
closure(I,J,A,Alpha,(B,Bs)) :-
    !, add(kante(I,J,A,Alpha,(B,Bs))),
    predict(J,B).
closure(I,J,A,Alpha,B) :-
    add(kante(I,J,A,Alpha,B)),
    predict(J,B).
```

Nach der Verkettung zweier Kanten muß die Tabelle mit der neuen Kante weiter unter Verkettung abgeschlossen werden:

⟨*earley-rec.pl*⟩+≡
```
complete(I,J,B) :-
    ( kante(K,I,A,Alpha,(B,Beta))
    ; kante(K,I,A,Alpha,B), Beta = []
    ),
    append(Alpha,[B],AlphaB),
    closure(K,J,A,AlphaB,Beta),
    fail.
complete(_,_,_).
```

Earley-Erkenner in Prolog (Forts.)

Die Voraussage (*Predict*) sucht in der Grammatik nach den zur Erwartung `S` bzw. `A` passenden Regeln, trägt sie in die Tabelle ein und schließt diese unter Verkettung ab:

⟨*earley-rec.pl*⟩+≡
```
predict(0) :-
    start_symbol(S), predict(0,S), fail.
predict(0).

predict(J,A) :-
    (A ---> Alpha), closure(J,J,A,[],Alpha),
    fail.
predict(J,A) :-
    (A ---> []),        % mit Loeschkanten verketten,
    complete(J,J,A),    % auch wenn diese schon
    fail.               % in der Tabelle sind!
predict(_,_).
```

Nur *neue* Kanten, die keine Instanz einer vorhandenen Kante sind, sollen in die Tabelle eingetragen werden:

⟨*earley-rec.pl*⟩+≡
```
add(Fact) :-
    ((var(Fact) ; known(Fact))
    -> fail ; assertz(Fact)).
known(Term) :-
    \+ \+ (numbervars(Term,c,0,_), Term).
```

Zu `numbervars(Term,c,0,N)` siehe den CYK-Erkenner.

Earley-Parser in Prolog

(für CFGs, mit Speichern einer Tabelle `kante/5`)

Im erkannten Teil `Alpha` speichern wir jetzt eine Baumliste.

⟨*earley.pl*⟩≡

```
:- dynamic kante/5. % kante(I,J,A,Alpha,Beta)

parse(Wortfolge,Baum) :-
    retractall(kante(_I,_J,_A,_Alpha,_Beta)),
    predict(0),
    parse(0,Wortfolge,Baum).

parse(I,[Wort|Rest],Baum) :-
    J is I + 1,
    scan(I,J,Wort),
    parse(J,Rest,Baum).
parse(I,[],[S|Alpha]) :-  % Eingabe erkannt?
    start_symbol(S),
    kante(0,I,S,Alpha,[]).

scan(I,J,Wort) :-         % i -Wort-> j=i+1
    ( word(Wort,A),
        closure(I,J,A,[Wort],[])
    ; ( kante(K,I,A,Alpha,([Wort],Beta))
      ; kante(K,I,A,Alpha,[Wort]),Beta = []
      ), append(Alpha,[[Wort]],AlphaW),
        closure(K,J,A,AlphaW,Beta)
    ), fail.
scan(_,_,_).
```

Durch `closure(I,J,A,Alpha,Beta)` werden auf $i \stackrel{A\to\alpha\cdot\beta}{\longrightarrow} j$ rekursiv die Regeln (*Complete*) und (*Predict*) anwendet, bis keine weiteren Kanten hinzukommen:

Earley-Parser in Prolog (Forts.)

Entsteht eine neue Kante $i \overset{B \to \beta\cdot}{\longrightarrow} j$, so werden alle „wartenden“ Kanten $k \overset{A \to \alpha \cdot B\beta}{\longrightarrow} i$ damit verbunden (Complete):

$\langle$*earley.pl*$\rangle$+≡

```
closure(I,J,B,Beta,[]) :-
     (add(kante(I,J,B,Beta,[]))
     -> complete(I,J,B) ; true).
```

Entsteht eine neue Kante $i \overset{A \to \alpha \cdot B\beta}{\longrightarrow} j$, so muß mit (Predict) bei j jede B-Regel vorgeschlagen und ggf. B gelöscht werden:

$\langle$*earley.pl*$\rangle$+≡

```
closure(I,J,A,Alpha,(B,Bs)) :-
     !, add(kante(I,J,A,Alpha,(B,Bs))),
     predict(J,B).
closure(I,J,A,Alpha,B) :-
     add(kante(I,J,A,Alpha,B)),
     predict(J,B).
```

Werden Kanten verkettet, muß weiter abgeschlossen werden:

$\langle$*earley.pl*$\rangle$+≡

```
complete(I,J,B) :-
     ( kante(K,I,A,Alpha,(B,Beta))
     ; kante(K,I,A,Alpha,B), Beta = []
     ),
     kante(I,J,B,Trees,[]),
     append(Alpha,[[B|Trees]],AlphaB),
     closure(K,J,A,AlphaB,Beta),
     fail.
complete(_,_,_).
```

Earley-Parser in Prolog (Forts.)

Mit `predict(j,A)` werden bei j alle A-Regeln eingetragen und –für löschbare A– mit vorhandenen Kanten verkettet:

$\langle$*earley.pl*$\rangle$+≡

```
predict(0) :-
    start_symbol(S),predict(0,S),fail.
predict(0).

predict(J,A) :-
    (A ---> Alpha),
    closure(J,J,A,[],Alpha),
    fail.
predict(J,A) :-
    (A ---> []),        % mit Loeschkanten verketten,
    complete(J,J,A),    % auch wenn diese schon
    fail.               % in der Tabelle sind!
predict(_,_).
```

Nur *neue* Kanten, die keine Instanz einer vorhandenen Kante sind, sollen in die Tabelle eingetragen werden:

$\langle$*earley.pl*$\rangle$+≡

```
add(Fact) :-
    ((var(Fact);known(Fact)) -> fail ; assertz(Fact)).
known(Term) :-
    \+ \+ (numbervars(Term,c,0,_), Term).
```

Der Earley-Parser kann *divergieren*, weil (*Complete*) evtl. (für $i = k$ und $\alpha \Rightarrow^* \epsilon$, $\beta \Rightarrow^* \epsilon$) einen Baum nach oben erweitert, ohne mehr von der Eingabe abzudecken.

Beispiel

⟨*cfg2.grm*⟩≡
```
start_symbol(s).
:- op(1100,xfx,'--->').
s ---> ap, bp.
ap ---> [].
bp ---> s.
word(a,ap).
word(b,bp).
```

⟨*Der Earley-Erkenner terminiert:*⟩≡
```
?- ['cfg2.grm',earley-rec],
   erkenne([b]).
Yes
```

⟨*Der Earley-Parser divergiert:*⟩≡
```
?- ['cfg2.grm',earley], trace(add,[exit]),
   parse([b],S).

 E (12) add(kante(0, 0, s, [], (ap, bp)))
 E (14) add(kante(0, 0, ap, [], []))
 E (16) add(kante(0, 0, s, [[ap]], bp))
 E (18) add(kante(0, 0, bp, [], s))
 E (12) add(kante(0, 1, bp, [b], []))
 E (14) add(kante(0, 1, s, [[ap],
                            [bp,b]], []))
 E (16) add(kante(0, 1, bp, [[s,[ap],[bp,b]]], []))
 E (18) add(kante(0, 1, s, [[ap],
                            [bp,[s,[ap],[bp,b]]]], []))
...
```

(Bei einer solchen Eingabe divergiert natürlich jeder Parser.)

GRAMMATIKÜBERSETZUNG

Um die Analyse zu beschleunigen, soll das Suchen nach einer Regel $(A \to \alpha) \in P$ so geändert werden, daß man nicht immer alle Grammatikregeln durchläuft, sondern

(i) die Kategorien des nächsten Eingabesymbols und

(ii) die für die Resteingabe „erwarteten“ Kategorien

benutzt, um Teilmengen $P' \subseteq P$ zu bestimmen, die alle in der jeweiligen Analysesituation noch nötigen Regeln umfassen.

Hilfsrelationen zwischen Kategorien

Aus den DCG-Regeln der Grammatik berechnen wir

(i) die *löschbaren Kategorien* $\{A \mid A \Rightarrow^+ \epsilon\}$,

(ii) die *Kettenrelation* $\{\, (A, B) \mid \ A \Rightarrow^+ B \,\}$,

(iii) die *Linke-Ecken-Relation*

$$<_{lc} \quad := \quad \{(B, A) \mid (A \to \alpha B \gamma) \in P,\ \alpha \Rightarrow^* \epsilon\}$$

und deren reflexive, transitive Hülle

$$<^*_{lc} \quad := \quad \{(B, A) \mid \exists \gamma \in Cat^*\ A \Rightarrow^* B\gamma\}$$

(iv) ggf. die *Hauptkategorie-Relation*

$$<_{hc} \quad := \quad \{(B, A) \mid (A \to \alpha \underline{B} \gamma) \in P\}$$

und ihre reflexive, transitive Hülle $<^*_{hc}$.

Wie man diese Regeln benutzt, zeigen wir an der Left-Corner-Analyse mit Rücksetzungen und an den Tabellenverfahren von Earley und dem für die Left-Corner-Analyse.

Nichtdeterministische Left-Corner-Analyse für übersetzte Grammatiken

Ist E die für ein Präfix der Resteingabe „erwartete“ Kategorie, D die Kategorie eines nicht-leeren, schon erkannten Anfangsstücks der Resteingabe und b das auf dieses Anfangsstück folgende Eingabesymbol (bzw. seine Kategorie), so sei

$$\begin{aligned}
&\mathit{predict}_{lc}(E, D, c) := \\
&\qquad \{A \to \alpha \cdot B\beta_1 \cdot C\beta_2) \in \ddot{P} \mid \\
&\qquad\qquad A <^*_{lc} E,\ \alpha\beta_1 \Rightarrow^* \epsilon,\ B \Rightarrow^* D,\ c <^*_{lc} C\}
\end{aligned}$$

Diese Teilmenge von $\ddot{P}$ kann für jedes (E, D, b) unabhängig von der Eingabe vorberechnet werden.

Der frühere Left-Corner-Erkenner kann damit beschleunigt werden, indem man das Probieren einer passenden Regel auf die jeweils als brauchbar vorausgesagten beschränkt:

Konfigurationen:	$(\eta, \gamma, w) \in \mathit{Cat}^* \times (\mathit{Cat} \cup \ddot{P})^* \times \Sigma^*$
Anfang bzw. Ende:	$(S, \epsilon, a_1 \cdots a_n)$ bzw. (ϵ, S, ϵ)
Eingabe lesen:	$(\eta E, \gamma, aw) \vdash_a (\eta E, \gamma a, w)$, falls $a <^*_{lc} E$.
Partiell reduzieren:	$(\eta E, \gamma D, cw)$
	$\vdash (\eta EC, \gamma(A \to \alpha B\beta_1 \cdot C\beta_2), cw)$,
	falls $(A \to \alpha \cdot B\beta_1 \cdot C\beta_2) \in \mathit{predict}_{lc}(E, D, c)$
Vervollständigen 1:	$(\eta B, \gamma(A \to \alpha \cdot B\beta_1 C\beta_2)D, cw)$
	$\vdash (\eta C, \gamma(A \to \alpha B\beta_1 \cdot C\beta_2), cw)$
	falls $B \Rightarrow^* D$, $\beta_1 \Rightarrow^* \epsilon$, und $c <^*_{lc} C$,
Vervollständigen 2:	$(\eta B, \gamma(A \to \alpha \cdot B\beta)D, w) \vdash (\eta, \gamma A, w)$,
	falls $B \Rightarrow^* D$ und $\beta \Rightarrow^* \epsilon$

Tabellenparser für übersetzte Grammatiken

Aus der Grammatik seien die Hilfsrelationen

(i) die *löschbaren Kategorien* $\{A \mid A \Rightarrow^+ \epsilon\} \subseteq Cat$,

(ii) die *Kettenrelation* $\{\, (A, B) \mid \; A \Rightarrow^+ B \,\} \subseteq Cat^2$,

(iii) die *Linke-Ecken-Relation*

$$<_{lc} \quad := \quad \{(B, A) \mid (A \to \alpha B\gamma) \in P,\; \alpha \Rightarrow^* \epsilon\}$$

und deren reflexive, transitive Hülle

$$<^*_{lc} \quad := \quad \{(B, A) \mid \exists\gamma \in Cat^*\; A \Rightarrow^* B\gamma\}$$

(iv) und folgende Teilmengen von $\dot{P}$, für $E, D \in Cat, c \in \Sigma$:

$$\begin{aligned}
&\mathit{predict}_{ep}(E, c) := \\
&\qquad \{(A \to \alpha \cdot C\beta) \in \dot{P} \mid A <^*_{lc} E,\; \alpha \Rightarrow^* \epsilon,\; c <^*_{lc} C\} \\
&\mathit{predict}_{lc}(E, D, c) := \\
&\qquad \{(A \to \alpha B\beta \cdot C\gamma) \in \dot{P} \mid \\
&\qquad\qquad A <^*_{lc} E,\; \alpha\beta \Rightarrow^* \epsilon,\; B \Rightarrow^* D,\; c <^*_{lc} C\}
\end{aligned}$$

vorberechnet. Wir benutzen sie nun, um

(i) die Suche nach passenden Grammatikregeln während der der Leftcorner- bzw. Earley-*Erkennung* einzuschränken,

(ii) dafür zu sorgen, daß nur kanonische Lösch- und Kettenableitungen gesucht werden.

Aus (ii) erhalten wir ein für jede Eingabe terminierendes *Analyse*verfahren, wobei in der Tabelle die Kanten $i \stackrel{A\to\alpha\cdot\beta}{\longrightarrow} j$ durch α-Bäume zu partiellen Analysen erweitert werden.

Verallgemeinerter Earley-Erkenner

Speichern von *Erwartungen* $j \cdot B$, „bei Position j beginne eine B-Phrase“, ersparen das Suchen nach Kanten $i \overset{A\to\alpha\cdot B\beta}{\longrightarrow} j$.

Vor.: Kontextfreie Grammatik $G = (N, \Sigma, S, P)$,
Hilfsrelationen $Cat \Rightarrow^* \epsilon$, $Cat \Rightarrow^* Cat$, $<_{lc}^*$, $predict_{ep}$

Eingabe: $w = a_1 \cdots a_n \in \Sigma^*$,

Methode: Konstruiere für die Positionen $0 \leq i \leq j \leq n$ alle Kanten $i \overset{A\to\alpha\cdot\beta}{\longrightarrow} j$, wo $(A \to \alpha\beta) \in P$, nach den Regeln:

$$\frac{start(S)}{0 \cdot S} \quad (Start) \qquad \frac{i \overset{A\to\alpha\cdot B\beta}{\longrightarrow} j}{j \cdot B} \quad (Expect)$$

$$\frac{j \cdot C, \quad (A \to \alpha \cdot B\beta) \in predict_{ep}(C, a_{j+1})}{j \overset{A\to\alpha\cdot B\beta}{\longrightarrow} j} \quad (Predict)$$

$$\frac{j = i+1}{i \overset{a_j\to a_j\cdot}{\longrightarrow} j} \quad (Scan)$$

$$\frac{i \overset{A\to\alpha\cdot B\beta}{\longrightarrow} k, \quad B \Rightarrow^* D, \quad k \overset{D\to\delta\cdot}{\longrightarrow} j, \quad \beta \Rightarrow^* \epsilon}{i \overset{A\to\alpha B\beta\cdot}{\longrightarrow} j} \quad (Complete_1)$$

$$\frac{\begin{array}{c} i \overset{A\to\alpha\cdot B\beta C\gamma}{\longrightarrow} k, \quad B \Rightarrow^* D, \quad k \overset{D\to\delta\cdot}{\longrightarrow} j, \\ \beta \Rightarrow^* \epsilon, \quad j < n, \quad a_{j+1} <_{lc}^* C \end{array}}{i \overset{A\to\alpha B\beta\cdot C\gamma}{\longrightarrow} j} \quad (Complete_2)$$

Ausgabe: Eine Kante $0 \overset{S\to\alpha\cdot}{\longrightarrow} n$ wird erzeugt $\iff$ $w \in L(G)$.

Verallgemeinerter Leftcorner-Erkenner

Der Aufbau der Kanten zu $w = a_1 \cdots a_n$ erfolgt durch

(i) Festlegen von Erwartungskategorien für die Resteingabe:

$$\frac{start(S)}{0 \cdot S} \quad (Start) \qquad \frac{i \xrightarrow{A \to \alpha \cdot B\beta} j}{j \cdot B} \quad (Expect)$$

(ii) Lesen der Eingabewörter:

$$\frac{j = i + 1}{i \xrightarrow{a_j \to a_j \cdot} j} \quad (Scan)$$

(iii) Voraussagen brauchbarer Regeln:

$$\frac{\begin{array}{c} i \cdot E, \qquad i \xrightarrow{D \to \delta \cdot} j, \\ (A \to \alpha B\beta \cdot C\gamma) \in predict_{lc}(E, D, a_{j+1}) \end{array}}{i \xrightarrow{A \to \alpha B\beta \cdot C\gamma} j} \quad (Predict)$$

(iv) Abschluß unter Verkettung von Kanten:

$$\frac{i \xrightarrow{A \to \alpha \cdot B\beta} k, \quad B \Rightarrow^* D, \quad k \xrightarrow{D \to \delta \cdot} j, \quad \beta \Rightarrow^* \epsilon}{i \xrightarrow{A \to \alpha B\beta \cdot} j} \quad (Complete_1)$$

$$\frac{\begin{array}{c} i \xrightarrow{A \to \alpha \cdot B\beta C\gamma} k, \quad B \Rightarrow^* D, \quad k \xrightarrow{D \to \delta \cdot} j, \\ \beta \Rightarrow^* \epsilon, \quad j < n, \quad a_{j+1} <^*_{lc} C \end{array}}{i \xrightarrow{A \to \alpha B\beta \cdot C\gamma} j} \quad (Complete_2)$$

Neben den Erwartungskategorien $j \cdot B$ wird das nächste Eingabewort und $c <^*_{lc} C$ zum Filtern der Regeln benutzt wird.

Beobachtungen

(i) Die Menge P der Grammatikregeln wird zur Parsezeit nicht mehr durchsucht; stattdessen nur die Teilmengen $predict_{ep}$ bzw. $predict_{lc}$ und die Tabelle der Kanten.

(ii) Die verallgemeinerten Earley- und Leftcorner-Erkenner unterscheiden sich *nur* in (*Predict*).

Der Leftcorner-Erkenner erzeugt nur $i \xrightarrow{A\to\alpha\cdot\beta} j$ mit $i < j$, d.h. wo α einen nicht-leeren Teil der Eingabe erkennt.

(iii) In (*Scan*) werden *ad-hoc*-Regeln $(a \to a)$ verwendet, die in $(D \to \delta)$ in (*Complete*) eingesetzt werden.

(iv) Zum *Erkennen* kann man Kanten $i \xrightarrow{A\to\alpha\cdot\beta} j$ zu $i \xrightarrow{A\to\cdot\beta} j$ vereinfachen, da der α-Teil nicht benutzt wird.

(v) Eine Kante $k \xrightarrow{D\to\delta\cdot} j$ wird nur für $k < j$ erzeugt.

Erweiterung der verallgemeinerten Erkenner zu Earley- und Leftcorner-Tabellenparsern

(i) Die Kanten $i \overset{A\to\alpha\cdot\beta}{\longrightarrow} j$ stehen für partielle Analysen von $a_{i+1}\cdots a_j$, d.h. sind Adressen für Baumfolgen mit Wurzeln α und Blättern $a_{i+1}\cdots a_j$.

(ii) In (*Complete*)-Schritten werden diese Baumfolgen um einen Baum zu $B \Rightarrow^* D \to \delta \Rightarrow^* a_{k+1}\cdots a_j$ mit $k < j$ erweitert.

(iii) In ($Complete_1$) wird für $i < k$ ein echt verzweigender Knoten A erzeugt, für $i = k$ muß man eine kanonische Ableitung $A \Rightarrow^* D$ aufbauen, also $B \Rightarrow^* D$ ignorieren.

(iv) ($Complete_2$) erzeugt nur echt verzweigende Knoten A, mit $k < j < l$ in $k \overset{B}{\longrightarrow} j$ und $j \overset{C}{\longrightarrow} l$ für ein l.

(v) Jede vollständige Analyse $A \Rightarrow^* a_i \dots a_j \neq \epsilon$ beginnt daher mit einer Kettenableitung $A \Rightarrow^* D$ zu einem Blatt D (und $i = j$) oder einem Verzweigungsknoten D.

Daraus folgt, daß nur kanonische Bäume aufgebaut werden, und daß der Parser terminiert.

Bem. Kanten $i \overset{D\to\delta\cdot}{\longrightarrow} j$ kann man zu $i \overset{D}{\longrightarrow} j$ vereinfachen, d.h. man braucht die *vollständigen* Analysen nur nach der Wurzel zu unterscheiden.

Implementierung

Auf die Implementierung der Erreichbarkeitsrelationen und die Parser für übersetzte Grammatiken gehen wir nicht ein. (Vielleicht kommt ein Nachtrag für den CFG-Fall; für den DCG-Fall siehe die Doku zum Seminar Grammatikimplementierung.)

DETERMINISTISCHE ANALYSE

Nicht-deterministische Parser „probieren aus“, ob die Erweiterung einer partiellen Analyse mit einer Grammatikregel oder einem Eingabewort zur Erkennung führt; wenn nicht, wird die Erweiterung rückgängig gemacht und eine Alternative gesucht.

Deterministische Parser sind solche, bei denen in jeder Situation *höchstens eine* Operation (Regelanwendung oder Lesen) zur Erkennung führt. Sie existieren nur für Grammatiken (von Programmier- oder logischen Sprachen), für die jeder Satz nur eine Analyse hat.

Für jede erreichbare Konfiguration (γ, v) ist die nächste Operation eindeutig durch γ und die k-Vorausschau $_k(v)$ bestimmt.

Nondet.	Transitionen (der Erkenner)	Det.
Top-Down	$(S, w) \vdash^* (\epsilon, \epsilon)$	LL(k)
(read)	$(a\gamma, av) \vdash_a (\gamma, v)$	
(expand)	$(B\gamma, v) \vdash_{(B\to\beta)} (\beta\gamma, v)$	
Leftcorner	$(\epsilon, w) \vdash^* (S, \epsilon)$	LC(k)
(shift)	$(\gamma, av) \vdash_a (\gamma a, v)$	
(predict)	$(\gamma B, v) \vdash_{(A\to B\beta)} (\gamma(A \to B \cdot \beta), v)$	
(compl-1)	$(\gamma(A \to \alpha \cdot B\beta)B, v) \vdash_1 (\gamma(A \to \alpha B \cdot \beta), v)$	
(compl-2)	$(\gamma(A \to \alpha\cdot), v) \vdash_2 (\gamma A, v)$	
Bottom-Up	$(\epsilon, w) \vdash^* (S, \epsilon)$	LR(k)
(shift)	$(\gamma, av) \vdash_a (\gamma a, v)$	
(reduce)	$(\alpha\beta, v) \vdash_{(B\to\beta)} (\alpha B, v)$	

Vergleich der erkannten Sprachklassen

Sei ein festes Alphabet gegeben und $LL(k)$ die Menge der Sprachen über diesem Alphabet, die eine Grammtik hat, für die der Top-Down-Parser mit k-Vorausschau determinisitisch ist. Entsprechend für Leftcorner und $LC(k)$ bzw. und Bottom-Up-Parser und $LR(k)$. Man kann zeigen:

(i) $LL(k) \subset LL(k+1)$,

(ii) $LC(k) \subset LC(k+1)$,

(iii) $LR(k) \subset LR(k+1)$,

(iv) $LL(k) \subseteq LR(k)$,

Für Programmiersprachen gibt es Parsergeneratoren, die Parser nach den LL- und LR-Verfahren erzeugen:

(i) YACC (oder Gnu/Bison) erzeugt LR(1)-Parser

(ii) ANTLR erzeugt LL(1)-Parser

Die praktisch wichtigste Klasse ist $LR(1)$.

LR(k)-Analyse

Für alle $(\epsilon, w) \vdash^* (\gamma, uv)$ mit $u = {}_k(uv)$ soll es höchstens eine direkte Nachfolgekonfiguration $(\gamma, u) \vdash (\tilde{\gamma}, \tilde{u})$ geben, die zur Erkennung irgendeiner Fortsetzung führt:

$$(\gamma, u\tilde{w}) \vdash (\tilde{\gamma}, \tilde{u}\tilde{w}) \vdash^* (S, \epsilon)$$

Voraussetzung: $G = (N, \Sigma, P, S)$ ist eine reduzierte kontextfreie Grammatik, also jedes Nonterminal A vom Startsymbol erreichbar und $\emptyset \neq L(A)$.

LR(0)-Analyse

Im Fall $k = 0$ ist die nächste Operation *allein durch den Kellerinhalt* bestimmt, also unabhängig von der Resteingabe.

Für jede erreichbare Konfiguration (γ, v) muß es also mindestens ein $\tilde{w} \in \Sigma^*$ geben, das zur Erkennung führt,

$$(\gamma, \tilde{w}) \vdash_? (\gamma', \tilde{w}') \vdash^* (S, \epsilon)$$

und die Operation ? bei $\vdash_?$ hängt nur von γ ab.

Wieviel müssen wir vom Keller $\gamma = \alpha\beta$ wissen, um sicher zu sein, daß höchstens eine Reduktion mit der Regel $(B \to \beta) \in P$ in Frage kommt? Es kann dann nur *eine* Rechtsableitung

$$S \Rightarrow^*_{rm} \alpha B \tilde{w} \Rightarrow_{rm} \alpha\beta\tilde{w} = \gamma\tilde{w}$$

geben, weshalb man die *reduzierten Linkskontexte von B bzgl. S*,

$$rlc(B) := \{\, \alpha \in Cat^* \mid S \Rightarrow^*_{rm} \alpha B \tilde{w} \text{ für ein } \tilde{w} \in \Sigma^* \,\},$$

kennen muß. Nun überlegt man sich leicht:

Prop. 12 Für jede reduzierte kontextfreie Grammatik gilt:

(i) ist $(S \to \alpha B \beta) \in P$, so ist $\alpha \in rlc(B)$.

(ii) ist $(A \to \alpha B \beta) \in P$, so ist $rlc(A) \cdot \alpha \subseteq rlc(B)$,

(iii) die Mengen $rlc(B)$ sind die kleinsten solchen Mengen.

Folgerung: die $rlc(B)$ sind reguläre Mengen über Cat, also kann ein endlicher Automaten überprüfen, ob $\alpha \in rlc(B)$.

Der LR(0)-Automat zu G

Seien $S', \triangleright, \triangleleft$ neue Symbole, $P' := P \cup \{S' \to \triangleright S \triangleleft\}$ und

$$\dot{P} := \{\, (A \to \alpha \cdot \beta) \mid (A \to \alpha\beta) \in P' \,\}.$$

Zu $q \subseteq \dot{P}$ sei $\overline{q}$ das kleinste $p \subseteq \dot{P}$ mit

$$q \cup \{\, (B \to \cdot\beta) \mid (A \to \alpha \cdot B\gamma) \in p, (B \to \beta) \in P \,\} \subseteq p.$$

Definiere den det. endl. Automaten $\mathcal{A}_G = (Q, \Sigma', \delta, I, F)$ durch

- $I := \overline{\{S' \to \triangleright \cdot S \triangleleft\}}$,
- $F := \delta(I, S)$, ($=$ das $q \in Q$ mit $(S' \to \triangleright S \cdot \triangleleft) \in q$)
- $\delta(q, X) := \{\, (A \to \alpha X \cdot \beta) \mid (A \to \alpha \cdot X\beta) \in \bar{q} \,\}$,
- $Q \subseteq 2^{\dot{P}}$ ist die kleinste Teilmenge von $2^{\dot{P}}$, die I enthält und mit einem q auch alle nicht-leeren $\delta(q, X)$.

Übliche Namen: *closure*(q) für $\overline{q}$, *goto*(q, X) für $\delta(q, X)$.

Hat man in $\mathcal{A}_G$ einen Reihe von Übergängen

$$q_0 \xrightarrow{X_1} q_1 \xrightarrow{X_2} q_2 \;\ldots\; q_{m-1} \xrightarrow{X_m} q_m \qquad \text{d.h. } q_i = \delta(q_{i-1}, X_i)$$

so gilt: $(B \to X_1 \cdots X_m \cdot) \in \overline{q}_m \Longrightarrow (B \to \cdot X_1 \cdots X_m) \in \overline{q}_0$.

Das ist klar für $m = 0$, und für $m \geq k > 0$ ist

$$(B \to X_1 \cdots X_k \cdot X_{k+1} \cdots X_m)$$

genau dann in $\overline{q}_k$, wenn es in q_k ist.

Ist $\gamma = \alpha\beta = \alpha X_1 \cdots X_m$ der Keller einer Konfiguration (γ, v), so ergibt eine Reduktion mit $(B \to X_1 \cdots X_m)$

$$(\gamma, \tilde{w}) = (\alpha\beta, \tilde{w}) \vdash_{(B\to\beta)} (\alpha B, \tilde{w}).$$

Die Kellerinhalte werden durch Lesen mit $\mathcal{A}_G$ in Zustände von Q überführt. Ist nun $q_0 = \delta(I, \alpha)$,

$$I \xrightarrow{\alpha} q_0 \xrightarrow{X_1} q_1 \xrightarrow{X_2} q_2 \ \ldots\ q_{m-1} \xrightarrow{X_m} q_m$$

so kommt $\mathcal{A}_G$ nach einer Reduktion mit $(B \to \beta)$ durch Lesen des neuen Kellerinhalts in den Zustand

$$I \xrightarrow{\alpha} q_0 \xrightarrow{B} \delta(q_0, B).$$

Man definiert daher einen *Kellerautomaten* $\mathcal{M}_G$, der Zustände von $\mathcal{A}_G$ auf seinem Keller stapelt, und folgende Übergänge hat:

(i) für $p := \delta(q, a)$, $a \in \Sigma$, $q \in Q$:

$$q \vdash_a^{\mathcal{M}} q\ p$$

(ii) für $(B \to \beta) \in P$, $\beta = X_1 \cdots X_m$, $q_0, \ldots, q_m \in Q$ mit $(B \to \cdot\beta) \in q_0$ und $(B \to X_1 \cdots X_k \cdot X_{k+1} \cdots X_m) \in q_k$ für $0 < k \leq m$, und $p = \delta(q_0, B)$:

$$q_0 q_1 \cdots q_m \vdash_{(B\to\beta)}^{\mathcal{M}} q_0\ p.$$

Falls G eine $LR(0)$-Grammatik ist, ist $\mathcal{M}_G$ deterministisch: das oberste Kellersymbol q legt fest, was man tun muß:

(i) wenn es ein $(B \to X_1 \cdots X_k \cdot aX_{k+2} \cdot X_m) \in q$ gibt und a das nächste Eingabesymbol ist, (i) anwenden,

(ii) wenn es $(B \to X_1 \cdots X_m \cdot) \in q = q_m$ gibt, (ii) anwenden.

Beh. Für $\alpha \in Cat^*$ gilt (mit $(B \to \cdot\epsilon) = (B \to \epsilon\cdot)$)

$\alpha \in rlc(B) \iff$ es gibt $(B \to \beta) \in P$ mit $(B \to \cdot\beta) \in \overline{\delta(I, \alpha)}$

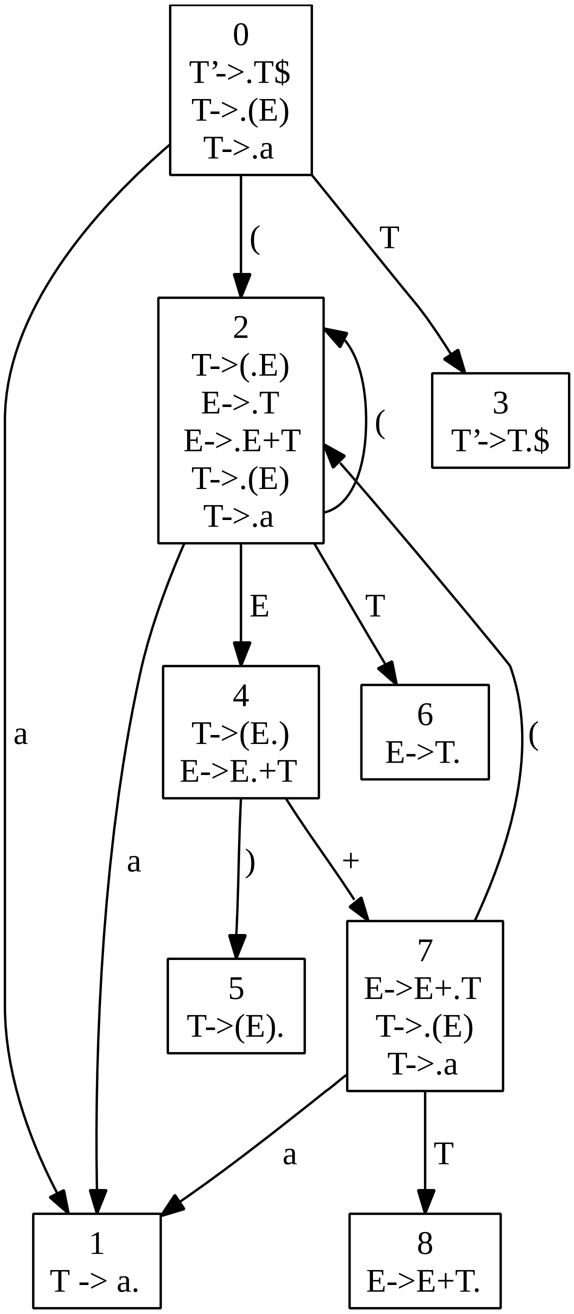
Beispiel

(Appel, Modern Compiler Implementation, 1997, korrig.)

Grammatik G: 1 $T \to (\, E\,)$ 3 $E \to T$

2 $T \to a$ 4 $E \to E\ +\ T$

Automat $\mathcal{A}_G$:



Kellerautomat $\mathcal{M}_G$:

	action					goto	
	(	)	a	+	$	T	E
0	s2		s1			4	
1	r2	r2	r2	r2	r2		
2	s2		s1			6	4
3					ok		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			8	
8	r4	r4	r4	r4	r4		

Eintrag bei Feld (q, X) je nach $q \xrightarrow{X} p$:

- sp *shift* p auf den Keller
- ri *reduce* mit Regel i $= (B \to \beta\cdot) \in q$
- ok *accept*

Reduzieren mit $(B \to \beta\cdot)$:=
pop$^{|\beta|}$; *push*(*goto*(*top*, B))

Erkennnungsbeispiel:

$$0, (a + a)\$ \quad \vdash 02, a + a)\$ \vdash 021, +a\$$$
$$\vdash 026, +a\$ \vdash 024, +a\$$$
$$\vdash 0247, a\$ \vdash \ldots$$

Implementierung des LR(0)-Automaten

Unterscheide Zustände $I \subseteq \dot{P}$ als (hoffentlich kleine) Teilmengen ihres Abschlusses $\overline{I}$ unter der Regel

$$\frac{(A \to \alpha \cdot B\gamma) \in \overline{I} \qquad (B \to \beta) \in P}{(B \to \cdot\beta) \in \overline{I}} \quad (pr).$$

Die Zustände werden numeriert und durch Fakten `state(N,R)` dargestellt; die Abschlüsse durch Fakten `closure(N,R)`.

$\langle$*lr0-automat.pl*$\rangle\equiv$

```
:- dynamic changed/1, goto/3, done/1, action/3.

compute_lr0_automaton :-
    remove_old_automata,
    compute_initial_state,     % endl.Automat  A_G
    compute_successor_states, % endl.Automat  A_G
    compute_actions.           % Kellerautomat M_G

remove_old_automata :-        % aufraeumen
    retractall(action(_,_,_)),
    retractall(maxstate(_)),
    retractall(state(_)),
    retractall(state(_,_)),
    retractall(closure(_,_)),
    retractall(goto(_,_,_)),
    retractall(changed(_)),
    retractall(done(_)).
```

Den Anfangszustand 0 legen wir mit neuen Symbolen 'start' und 'end' fest. (Da der Automat unabhängig von Eingaben berechnet wird, muß man mit 'end' das Eingabeende simulieren.)

Anfangszustand und Abschluß

⟨*lr0-automat.pl*⟩+≡

```
compute_initial_state :-
    assert(state(0)),
    assert(maxstate(0)),
    ( start_symbol(S),
        add(state(0,(start --> [],[S,end]))),
        fail
    ; true ),
    close_state(0).
```

Abschluß $\overline{I}$: ergänze zu den Regeln $(A \rightarrow \alpha \cdot C\beta)$ von I so lange die Regeln $(C \rightarrow \cdot\gamma)$, bis keine neuen mehr hinzukommen.

⟨*lr0-automat.pl*⟩+≡

```
close_state(P) :-
    ( state(P,Rule),
        add(closure(P,Rule)),
        add(changed(P)),
        fail
    ; (changed(P) -> close_state2(P) ; true)
    ).
close_state2(P) :-
    retract(changed(P)),
    ( closure(P,(A --> Alpha,[C|Beta])),
        (C --> GammaTuple),
        tupleToList(GammaTuple,Gamma),
        \+ known_instance(closure(P,(C --> [],Gamma))),
        add(closure(P,(C --> [],Gamma))),
        add(changed(P)),
        fail
    ; (changed(P) -> close_state2(P) ; true) ).
```

$\langle$*lr0-automat.pl*$\rangle$+$\equiv$

```
% ---- Hilfsdefinitionen ----
add(A) :-
    known_instance(A), !.
add(A) :-
    assertz(A).
known_instance(A) :- % ist A[Xi/consti] beweisbar?
        \+ \+ (numbervars(A, const, 0, B), A).

tupleToList((A, B), [A|C]) :-
    !, tupleToList(B, C).
tupleToList(A, [A]).
% ----------------------------
```

Nachfolgerzustände

Nachdem ein Zustand `Q` berechnet ist, werden seine direkten Nachfolger bestimmt und `Q` wird mit `done(Q)` als erledigt markiert. Ist ein Nachfolger ein neuer Zustand, so werden auch dessen direkte Nachfolger berechnet, bis keine neuen Zustände mehr entstehen:

$\langle$*lr0-automat.pl*$\rangle$+$\equiv$

```
compute_successor_states :-
    state(P), \+ done(P),
    !, compute_successors(P), add(done(P)),
    compute_successor_states.
compute_successor_states.
```

Um die Nachfolger p eines Zustands q zu bestimmen, brauchen wir alle B, für die es eine Regel $(A \to \alpha{\cdot}B\beta) \in q$ gibt. Für jedes solche B wird ein B-Nachfolger von p gebildet und ggf. mit einem schon vorhandenen Zustand identifiziert:

⟨*lr0-automat.pl*⟩+≡

```
compute_successors(P) :-
    ( setof(B,A^Alpha^Beta^closure(P,
             (A --> Alpha,[B|Beta])),Bs)
    -> compute_successors(P,Bs)
    ; true ).

compute_successors(P,[B|Bs]) :-
    mk_succ(P,B),
    compute_successors(P,Bs).
compute_successors(P,[]).

mk_succ(P,end) :- !.
mk_succ(P,B) :-
    ( closure(P,(A --> Alpha,[B|Beta])),
        append(Alpha,[B],AlphaB),
        add(state(newstate,(A --> AlphaB,Beta))),
        fail             % omit: state(newstate)!
    ; close_state(newstate) ),
    number(newstate,Q),
    ( retract(state(newstate,Rule)),
        add(state(Q,Rule)),
        fail
    ; retract(closure(newstate,Rule)),
        add(closure(Q,Rule)),
        fail
    ; true ),
    add(goto(P,B,Q)).
```

Vergleich mit vorhandenen Zuständen

Ob der mögliche neue Zustand aus den geteilten Regeln `R` mit `closure(newstate,R)` ein schon bekannter Zustand ist, wird durch Vergleichen mit den bisher berechneten Zuständen ermittelt. Wenn ja, wird für die Übergänge die Nummer des bekannten Zustands benutzt, sonst die um 1 erhöhte bisherige maximale Zustandsnummer.

⟨*lr0-automat.pl*⟩+≡

```
number(newstate,N) :-
     setof(Q,state(Q),Qs),
     number(newstate,Qs,N).

number(newstate,[Q|Qs],N) :-
     ( same_states(newstate,Q)
     -> N=Q ; number(newstate,Qs,N) ).
number(newstate,[],N) :-
     maxstate(M),
     retract(maxstate(M)),
     N is M+1,
     assert(maxstate(N)),
     add(state(N)).
```

Da `set_of/3` die Ergebnisliste sortiert, lassen sich Zustände leicht vergleichen:

⟨*lr0-automat.pl*⟩+≡

```
same_states(Q,P) :-
     setof(Rule,closure(Q,Rule),ClosureQ),
     setof(Rule,closure(P,Rule),ClosureP),
     ClosureQ = ClosureP.
```

Berechnung der Aktionstabelle

Schließlich muß noch die Aktionstabelle des Kellerautomaten konstruiert werden.

⟨*lr0-automat.pl*⟩+≡

```
compute_actions :-
    state(Q),
    terminal(B),
    ( closure(Q,(A --> Alpha,[])),
        add(action(Q,B,reduce(A --> Alpha)))
    ; closure(Q,(_A --> _Alpha,[B|_])),
        goto(Q,B,P),
        add(action(Q,B,shift(P)))
    ; closure(Q,(A --> Alpha,[end])),
        add(action(Q,end,accept))
    ),
    fail.
compute_actions :-
    state(Q),
    terminal(B),
    (action(Q,B,_) -> true ; add(action(Q,B,error))),
    fail.
compute_actions.

terminal(B) :-
    (_ --> GammaTuple),
    tupleToList(GammaTuple,Gamma),
    member(B,Gamma),
    \+ (B --> _).
```

Der LR(0)-Parser (Kellerautomat)

Um eine Eingabe mit dem Kellerautomaten zu erkennen, brauchen wir noch die Stapelveränderungen:

⟨*lr0-rec.pl*⟩≡

```
parse(Ws) :- parse([0],Ws).

parse([Q|Qs],[W|Ws]) :-
    action(Q,W,Action),
    ( Action = reduce(A --> Alpha),
        reduce([Q|Qs],(A --> Alpha),RedStack),
        parse(RedStack,[W|Ws])
    ; Action = shift(P), parse([P,Q|Qs],Ws)
    ; Action = error, write('\n reject')
    ).
parse([Q|Qs],[]) :-
    action(Q,end,Action),
    ( Action = reduce(A --> Alpha),
        reduce([Q|Qs],(A --> Alpha),RedStack),
        parse(RedStack,[])
    ; Action = error, write('\n reject')
    ; Action = accept, write('\n accept')
    ).
reduce(Stack,(A --> Alpha),RedStack) :-
    length(Alpha,N),
    length(Remove,N),
    append(Remove,[R|Rest],Stack),
    goto(R,A,P),
    RedStack = [P,R|Rest].
```

Der Parser braucht nur `action/3` und von `goto/3` die Übergänge mit Nonterminalen — nicht die Interna der Zustände.

Beispiel 1

Die übliche DCG-Regelexpansion wird mit `term_expansion/2` ausgeschaltet.

⟨*Grammatiken/lr0.grm*⟩≡

```
term_expansion((A --> B), (A --> B)).

start_symbol(t).
e --> t.
e --> e, +, t.
t --> '(', e, ')'.
t --> a.
```

Die Berechnung des LR(0)-Automaten ergibt:

⟨*Berechnung des Automaten*⟩≡

```
?- compute_lr0_automaton, listing(state/2),
   listing(closure), listing(goto), listing(action).

state(0, (start-->[], [t, end])).
state(1, (t-->['('], [e, ')'])).
state(2, (t-->[a], [])).
state(3, (start-->[t], [end])).
state(4, (t-->['(', e], [')'])).
state(4, (e-->[e], [+, t])).
state(5, (e-->[t], [])).
state(6, (t-->['(', e, ')'], [])).
state(7, (e-->[e, +], [t])).
state(8, (e-->[e, +, t], [])).

closure(0, (start-->[], [t, end])).
closure(0, (t-->[], ['(', e, ')'])).
closure(0, (t-->[], [a])).
```

```
closure(1, (t-->['('], [e, ')'])).
closure(1, (e-->[], [t])).
closure(1, (e-->[], [e, +, t])).
closure(1, (t-->[], ['(', e, ')'])).
closure(1, (t-->[], [a])).
closure(2, (t-->[a], [])).
closure(3, (start-->[t], [end])).
closure(4, (t-->['(', e], [')'])).
closure(4, (e-->[e], [+, t])).
closure(5, (e-->[t], [])).
closure(6, (t-->['(', e, ')'], [])).
closure(7, (e-->[e, +], [t])).
closure(7, (t-->[], ['(', e, ')'])).
closure(7, (t-->[], [a])).
closure(8, (e-->[e, +, t], [])).

goto(0, '(', 1).
goto(0, a, 2).
goto(0, t, 3).
goto(1, '(', 1).
goto(1, a, 2).
goto(1, e, 4).
goto(1, t, 5).
goto(4, ')', 6).
goto(4, +, 7).
goto(7, '(', 1).
goto(7, a, 2).
goto(7, t, 8).

action(0, '(', shift(1)).
action(0, a, shift(2)).
action(1, '(', shift(1)).
action(1, a, shift(2)).
```

```
action(2, +, reduce((t-->[a]))).
action(2, '(', reduce((t-->[a]))).
action(2, ')', reduce((t-->[a]))).
action(2, a, reduce((t-->[a]))).
action(2, end, reduce((t-->[a]))).
action(3, end, accept).
action(4, +, shift(7)).
action(4, ')', shift(6)).
action(5, +, reduce((e-->[t]))).
action(5, '(', reduce((e-->[t]))).
action(5, ')', reduce((e-->[t]))).
action(5, a, reduce((e-->[t]))).
action(5, end, reduce((e-->[t]))).
action(6, +, reduce((t-->['(', e, ')']))).
action(6, '(', reduce((t-->['(', e, ')']))).
action(6, ')', reduce((t-->['(', e, ')']))).
action(6, a, reduce((t-->['(', e, ')']))).
action(6, end, reduce((t-->['(', e, ')']))).
action(7, '(', shift(1)).
action(7, a, shift(2)).
action(8, +, reduce((e-->[e, +, t]))).
action(8, '(', reduce((e-->[e, +, t]))).
action(8, ')', reduce((e-->[e, +, t]))).
action(8, a, reduce((e-->[e, +, t]))).
action(8, end, reduce((e-->[e, +, t]))).
action(0, +, error).
action(0, ')', error).
action(0, end, error).
action(1, +, error).
action(1, ')', error).
action(1, end, error).
action(3, +, error).
action(3, '(', error).
```

```
action(3, ')', error).
action(3, a, error).
action(4, '(', error).
action(4, a, error).
action(4, end, error).
action(7, +, error).
action(7, ')', error).
action(7, end, error).
```

Die Ergebnisse stimmen mit dem oben gezeigten Beispiel nach A.Appel überein, außer daß die Zustandsnummern gemäß $1 \leftrightarrow 2,\ 5 \leftrightarrow 6$ umzurechnen sind.

Erkennung mit dem LR(0)-Parser $\mathcal{M}_G$

Der Parser erkennt passende Eingaben:

⟨*Erkennung mit dem Kellerautomaten*⟩≡

```
?- trace(parse,[call,exit]), trace(action,[exit]).
Yes
[debug]  ?- parse(['(',a,+,a,')']).
 T Call: (8) parse(['(', a, +, a, ')'])
 T Call: (9) parse([0], ['(', a, +, a, ')'])
 T Exit: (10) action(0, '(', shift(1))
 T Call: (10) parse([1, 0], [a, +, a, ')'])
 T Exit: (11) action(1, a, shift(2))
 T Call: (11) parse([2, 1, 0], [+, a, ')'])
 T Exit: (12) action(2, +, reduce((t-->[a])))
 T Call: (12) parse([5, 1, 0], [+, a, ')'])
 T Exit: (13) action(5, +, reduce((e-->[t])))
 T Call: (13) parse([4, 1, 0], [+, a, ')'])
 T Exit: (14) action(4, +, shift(7))
 T Call: (14) parse([7, 4, 1, 0], [a, ')'])
```

```
 T Exit: (15) action(7, a, shift(2))
 T Call: (15) parse([2, 7, 4, 1, 0], [')'])
 T Exit: (16) action(2, ')', reduce((t-->[a])))
 T Call: (16) parse([8, 7, 4, 1, 0], [')'])
 T Exit: (17) action(8, ')', reduce((e-->[e, +, t])))
 T Call: (17) parse([4, 1, 0], [')'])
 T Exit: (18) action(4, ')', shift(6))
 T Call: (18) parse([6, 4, 1, 0], [])
 T Exit: (19) action(6, end, reduce((t-->['(', e, ')'])))
 T Call: (19) parse([3, 0], [])
 T Exit: (20) action(3, end, accept)

 accept
 T Exit: (19) parse([3, 0], [])
 T Exit: (18) parse([6, 4, 1, 0], [])
 T Exit: (17) parse([4, 1, 0], [')'])
 T Exit: (16) parse([8, 7, 4, 1, 0], [')'])
 T Exit: (15) parse([2, 7, 4, 1, 0], [')'])
 T Exit: (14) parse([7, 4, 1, 0], [a, ')'])
 T Exit: (13) parse([4, 1, 0], [+, a, ')'])
 T Exit: (12) parse([5, 1, 0], [+, a, ')'])
 T Exit: (11) parse([2, 1, 0], [+, a, ')'])
 T Exit: (10) parse([1, 0], [a, +, a, ')'])
 T Exit: (9) parse([0], ['(', a, +, a, ')'])
 T Exit: (8) parse(['(', a, +, a, ')'])
Yes
```

Beispiel 2

Die folgende Grammatik ist keine LR(0)-Grammatik.

⟨*Grammatiken/lr1.grm*⟩≡
```
term_expansion((A --> B), (A --> B)).

start_symbol(e).
e --> t, +, e.
e --> t.
t --> a.
```

Die Berechnung des LR(0)-Automaten $\mathcal{A}_G$ ergibt einen Zustand, der eine `shift` *und* eine `reduce`-Aktion empfiehlt:

⟨*Berechnung des LR(0)-Automaten*⟩≡
```
?- ['Grammatiken/lr1.grm'].
?- compute_lr0_automaton,
   listing(closure), listing(goto), listing(action).

closure(0, (start-->[], [e, end])).
closure(0, (e-->[], [t, +, e])).
closure(0, (e-->[], [t])).
closure(0, (t-->[], [a])).
closure(1, (t-->[a], [])).
closure(2, (start-->[e], [end])).
closure(3, (e-->[t], [+, e])).
closure(3, (e-->[t], [])).
closure(4, (e-->[t, +], [e])).
closure(4, (e-->[], [t, +, e])).
closure(4, (e-->[], [t])).
closure(4, (t-->[], [a])).
closure(5, (e-->[t, +, e], [])).
```

```
goto(0, a, 1).
goto(0, e, 2).
goto(0, t, 3).
goto(3, +, 4).
goto(4, a, 1).
goto(4, e, 5).
goto(4, t, 3).

action(0, a, shift(1)).
action(1, +, reduce((t-->[a]))).
action(1, a, reduce((t-->[a]))).
action(1, end, reduce((t-->[a]))).
action(2, end, accept).
action(3, +, reduce((e-->[t]))). % Konflikt:
action(3, +, shift(4)).  % reduce oder shift ??
action(3, a, reduce((e-->[t]))).
action(3, end, reduce((e-->[t]))).
action(4, a, shift(1)).
action(5, +, reduce((e-->[t, +, e]))).
action(5, a, reduce((e-->[t, +, e]))).
action(5, end, reduce((e-->[t, +, e]))).
action(0, +, error).
action(0, end, error).
action(2, +, error).
action(2, a, error).
action(4, +, error).
action(4, end, error).
```

Wegen des Konflikts im Zustand 3 ist die nächste Aktion nicht eindeutig bestimmt, die Grammatik also keine LR(0)-Grammatik.