Bottom-Up-Chart-Parser

Übersicht

- Grammatik und Lexikon
- Kanten-Repräsentation
- Einfügen von Kanten
  - Terminal-Kanten
  - Inaktive Kanten
  - Aktive Kanten
- Ende des Parsings

Zweck

- Verstehen des Verfahrens und der Prolog-Implementation
  - *nicht:* Auswendiglernen des Prolog-Programms!

Bottom-Up-Chart-Parsing

Nachfolgend wird ein *einfacher* Bottom-Up-Chart-Parser vorgestellt.

- kommt nicht mit komplexen Nichtterminal-Symbolen zurecht
  - entsprechende Mechanismen: → spätere Lektion
- erkannte syntaktische Struktur wird zwar gespeichert, aber nicht ausgegeben
  - der Parser ist also kein Parser, sondern nur ein Akzeptor
  - Ausgabe des Baumes: → Übungsaufgaben

Diese Folien besprechen nur das Grundprinzip

- Programm-Ausdruck zum Verständnis nötig

Grammatik und Lexikon

Wie üblich halten wir die Grammatikregeln und Lexikoneinträge als Prolog-Fakten fest.

```
rule(s, [np, vp]).
rule(np, [det, n]).
rule(vp, [v]).
rule(vp, [v, np]).
```

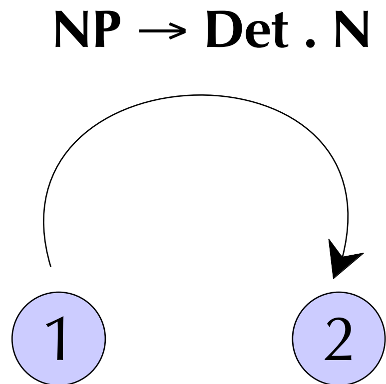
Grammatikregeln

```
word(det, the).
word(det, a).
word(n, dog).
word(n, cat).
word(v, chases).
word(v, sees).
word(v, sings).
```

Lexikon-Einträge

Kanten-Repräsentation

In Prolog geschriebene Chart-Parser verwenden üblicherweise komplexe Terme zum Speichern der Kanten.



Zwischen Knoten Nr. 1 und Knoten Nr. 2 ist eine (unvollständige) NP, von der bereits das Det gefunden wurde, der aber zur Vollständigkeit noch ein N fehlt.

```
edge(1, 2, np, [det], [n], …)
```

Kanten-Repräsentation

In Prolog geschriebene Chart-Parser verwenden üblicherweise eine von zwei Arten zum Speichern aller Kanten:

- Liste der Terme, welche den einzelnen Kanten entprechen
  - Einfügen neuer Kanten mit Listen-Verkettungsoperator |
- Menge von Prolog-Fakten
  - Einfügen neuer Kanten mit `assert`

Beide Varianten sind in etwa gleichwertig.

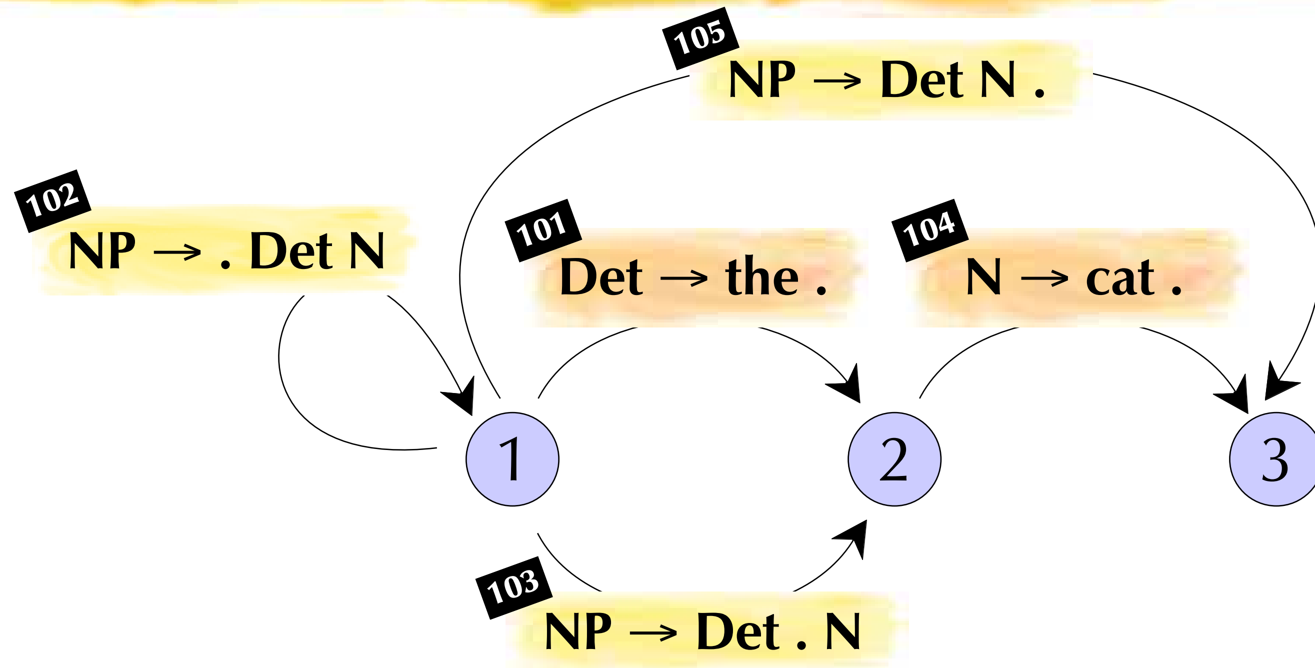
- Im besprochenen Parser wird die zweite Variante benutzt.

Kanten-Repräsentation

Wir speichern folgende Informationen für eine Kante:

- eine Nummer, welche die Kante eindeutig identifiziert
- der Knoten, an dem die Kante beginnt
- der Knoten, an dem die Kante endet
- die Kategorie der Kante
- der »Teil vor dem Punkt«
  - Liste der inaktiven Kanten, welche die bereits gefundenen Konstituenten umfassen
  - Identifikations-Nummern in umgekehrter Reihenfolge
- der »Teil nach dem Punkt«
  - Liste der noch zu findenden Kategorien (in richtiger Reihenfolge)

Kanten-Repräsentation: Beispiel



Kanten:

- `edge(101, 1, 2, det, [lex(the)], [])`
- `edge(102, 1, 1, np,  [], [det, n])`
- `edge(103, 1, 2, np,  [101], [n])`
- `edge(104, 2, 3, n,   [lex(cat)], [])`
- `edge(105, 1, 3, np,  [104, 101], [])`

Terminal-Kanten einfügen

start_chart(+StartVertex, –EndVertex, +Sentence)

- fügt eine Terminalkante ein, die an Knoten StartVertex beginnt
- diese neue Kante endet bei Knoten (StartVertex + 1)
- rekursiver Aufruf füllt Chart, bis das letzte Terminalsymbol aus Sentence verarbeitet wurde — d.h. bis zum Satzende

Abbr.

```
start_chart(V, V, []).
```

Rekursiver Fall

```
start_chart(StartVertex, EndVertex, [Word|Words]) :-
   StartVertex_plus_1 is StartVertex + 1,
   foreach(word(Cat, Word),
       add_edge(StartVertex, StartVertex_plus_1,
                Cat, /* Found */ [lex(Word)],
                /* ToFind */ [])),
   start_chart(StartVertex_plus_1, EndVertex, Words).
```

Kanten einfügen

Ein Chart-Parser fügt niemals eine Kante in die Chart ein, die bereits in der Chart enthalten ist.

- … denn das »Merken« von Zwischenresultaten ist gerade der Zweck der Chart!

Beim Einfügen einer neuen Kante gibt es folgende Möglichkeiten:

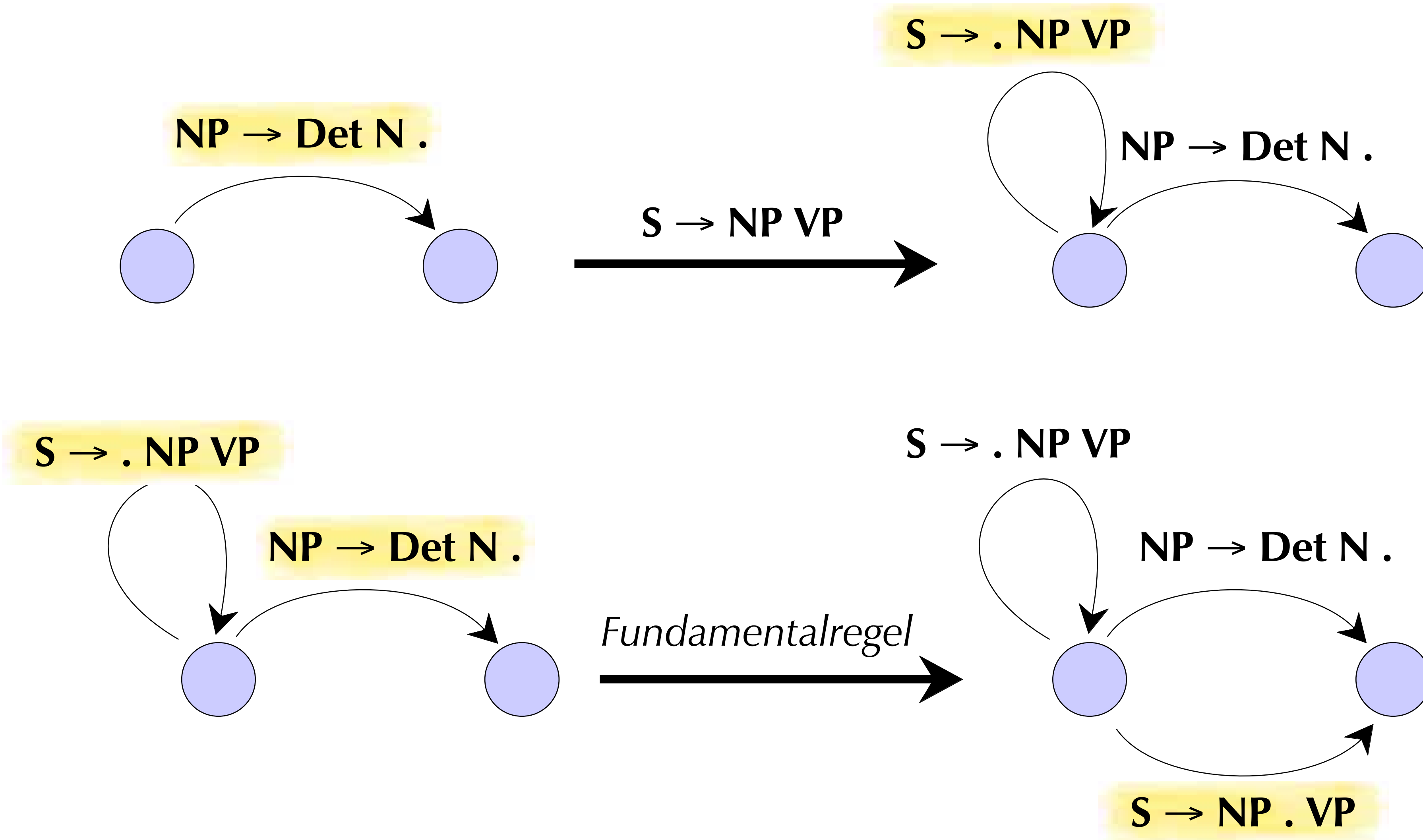
- die einzufügende Kante ist bereits in der Chart
- sonst: die einzufügende Kante ist eine inaktive/passive Kante
- ansonsten ist die einzufügende Kante eine aktive Kante

Inaktive Kante einfügen

Wenn eine inaktive Kante *I* der Kategorie *Cat* einzufügen ist:

- ▸ füge die Kante zur Chart hinzu
- ◆ für jede Grammatikregel der Form *LHS* → *Cat* β:
  - ▸ füge eine aktive Kante der Form *LHS* → . *Cat* β zur Chart hinzu
- ◆ für jede aktive Kante *A*,
  - ◆ die bei jenem Knoten endet, an dem *I* beginnt,
  - ◆ und die als nächstes ein Symbol der Kategorie *Cat* benötigt,
  - ▸ wende die Fundamentalregel an und füge die Kombination aus *A* und *I* zur Chart hinzu.

Inaktive Kante einfügen: Beispiel

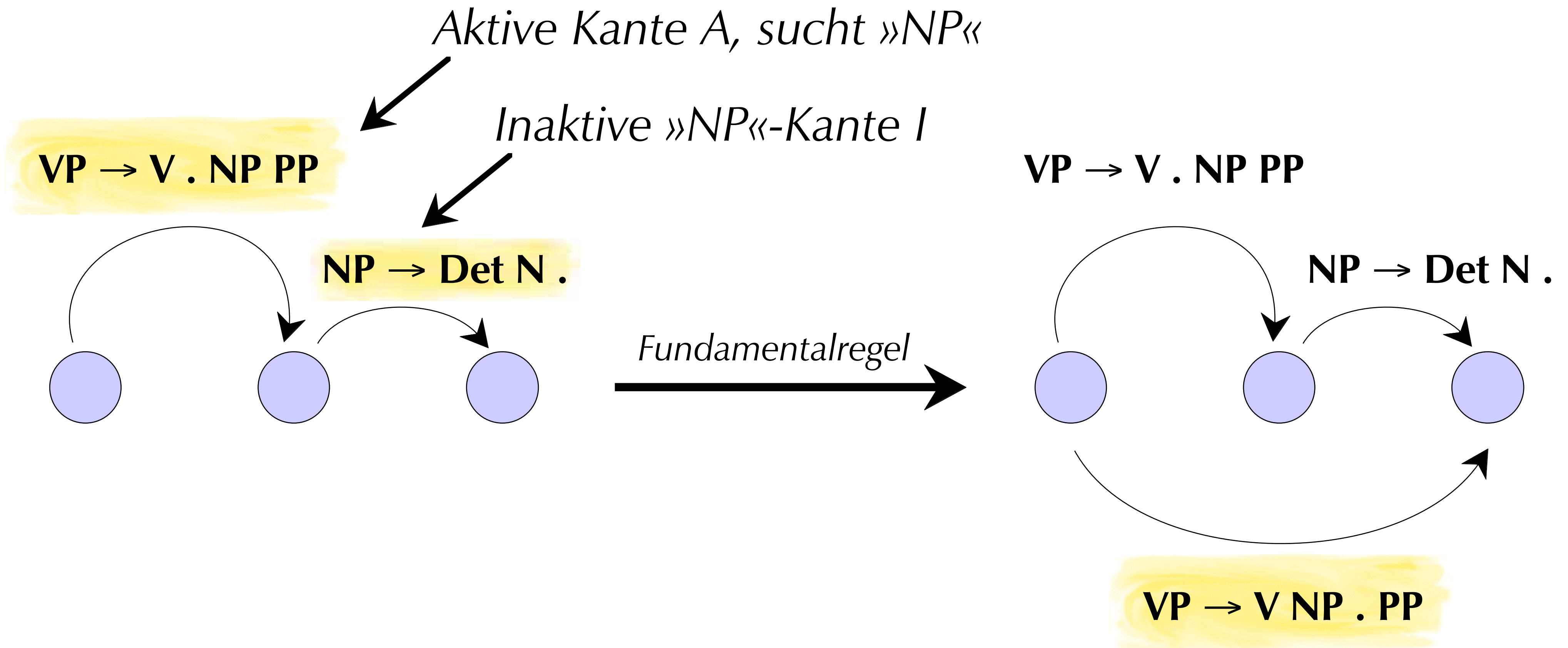


Aktive Kante einfügen

Wenn eine aktive Kante *A* einzufügen ist:

- füge die Kante zur Chart hinzu
- für jede inaktive Kante *I*,
  - die bei jenem Knoten beginnt, an dem *A* endet,
  - und die von der Kategorie ist, welche *A* als nächste benötigt,
  - wende die Fundamentalregel an und füge die Kombination aus *A* und *I* zur Chart hinzu.

Aktive Kante einfügen: Beispiel



Ende des Parsings

Wenn ein Chart-Parser keine Kanten mehr einfügen kann, ist die syntaktische Analyse beendet.

Eine Analyse für die Eingabe-Kette wurde gefunden, wenn

- eine Kante vom ersten bis zum letzten Knoten reicht,
- und diese Kante ist inaktiv,
- und die Kategorie dieser Kante ist das Startsymbol der Grammatik.

botUpChartParse.pl

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999
Online unter http://www.coli.uni-sb.de/~brawer/prolog/botupchart/

```
% ============================================================
% botUpChartParse.pl
%
% A simple bottom-up chart parser
%
% Example Query:
% ?- parse(s, [the,cat,sings]).
%
% The program was taken from [Gazdar/Mellish, 1989], but it
% has been modified to a large extent.
% ============================================================

% Representation of edges
% -----------------------
% All edges (both inactive and active) are asserted as Prolog
% facts edge/6. They have the following form:
%
% edge(ID, StartVertex, EndVertex, Category, Found, ToFind)
% ID:          a unique identification number for the edge
% StartVertex: the vertex number where the edge starts
% EndVertex:   the vertex number where the edge ends
% Category:    the category of the edge
% Found:       a list of those inactive edges that lead to the
%              part "before the dot" -- in reversed order.
%              either an edge ID or a term lex/1
% ToFind:      a list of categories which have yet to be found
%              in order to make the edge complete (i.e. the
%              part "after the dot")
%
% Examples
% --------
% edge(101, 1, 2, det, [lex(the)], []).  % <1,2> det --> the .
% edge(102, 1, 1, np,  [], [det, n]).    % <1,1> np  --> . det n
% edge(103, 1, 2, np,  [101], [n]).      % <1,2> np --> det . n
% edge(104, 2, 3, n,   [lex(cat)], []).  % <2,3> n  --> cat .
% edge(105, 1, 3, np,  [104, 101], []).  % <1,3> np --> det n .
% edge(106, 1, 1, s,   [], [np, vp]).    % <1,1> s --> . np vp

% Tell Prolog that edge/6 will be modified dynamically, i.e.
% with assert and retract.
:- dynamic(edge/6).


test :-
   parse(s, [the,cat,sings]).       % for debugging
```

```
% ------------------------------------------------------------
% parse(+Symbol, +String)
% ------------------------------------------------------------
% Tries to parse String as Symbol, according to a phrase-
% structure grammar given at the end of the listing.
% Initializes the chart and performs bottom-up parsing
% (with start_chart/3). Then, all edges are retrieved
% - that span the entire chart
% - and that are inactive (i.e. don't need any more things in
%   order to be complete)
% - and whose category is Symbol.
% ==> write the ID for these edges on the screen.

parse(Symbol, String) :-
   V0 is 1,
   start_chart(V0, Vn, String),
   foreach(edge(ID, V0, Vn, Symbol, /* Found */ _, /* ToFind */ []),
     (write('Found parse: Edge #'), write(ID), nl)),
   retractall(edge(_,_,_,_,_,_)).


% ------------------------------------------------------------
% start_chart(+StartVertex, -EndVertex, +SentenceList)
% ------------------------------------------------------------
% Initializes the chart (starting at StartVertex) with edges
% for the terminal symbols in SentenceList. Performs a
% bottom-up chart parsing (see add_edge below). Returns the
% last vertex in EndVertex.

start_chart(V, V, []).                    % termination

start_chart(Vfirst, Vlast, [Word | Words]) :-
   Vfirst_plus_1 is Vfirst + 1,
   foreach(word(Cat, Word),
     add_edge(Vfirst, Vfirst_plus_1, Cat,
      /* Found */ [lex(Word)],
      /* ToFind */ [])),
   start_chart(Vfirst_plus_1, Vlast, Words).


% ------------------------------------------------------------
% add_edge -- case 1: add an existing edge
% ------------------------------------------------------------
% Edges are only added to the chart if no such edge is in
% the chart yet.

add_edge(Vfrom, Vto, Cat, Found, ToFind) :-
   % The next line succeeds only if the to-be-added edge is
   % already in the edge. Otherwise, backtracking occurs
   % and another clause of add_edge/5 is called.
   edge(ID, Vfrom, Vto, Cat, Found, ToFind),
   !,
   write('Ignoring: '),
   write_edge(edge(ID, Vfrom, Vto, Cat, Found, ToFind)).
```

```
% ------------------------------------------------------------
% add_edge -- case 2: add an inactive edge
% ------------------------------------------------------------
% When an inactive edge I of category Cat is to be added:
% 1. add I to the chart
% 2. for each grammar rule
%    - where the leftmost symbol in the right-hand side = Cat
%      (i.e. the rule is of the form LHS --> Cat Cs)
%    ==> add an active edge that needs I to the chart
% 3. for each active edge A
%    - that immediately precedes I
%      (i.e. end vertex of A = start vertex of I)
%    - and that can combine with I
%      (i.e. A needs an edge of I's category)
%    ==> add the combination of A with I to the chart

add_edge(V1, V2, Cat, Found, []) :-
   new_id(ID),
   assert_edge(edge(ID, V1, V2, Cat, Found, [])),
   foreach(rule(LHS, [Cat | Cs]),
     add_edge(V1, V1, LHS,
      /* Found */  [],
      /* ToFind */ [Cat | Cs])),
   foreach(edge(_, V0, V1, LeftCat, LeftFound, [Cat | LeftToFind]),
      add_edge(V0, V2, LeftCat,
      /* Found */  [ID | LeftFound],
      /* ToFind */ LeftToFind)).


% ------------------------------------------------------------
% add_edge -- case 3: add an active edge
% ------------------------------------------------------------
% When an active edge A is to be added:
% 1. add A to the chart
% 2. for each inactive edge I
%    - that immediately follows A
%      (i.e. end vertex of A = start vertex of I)
%    - and that can combine with A
%      (i.e. A needs an edge of I's category)
%    ==> add the combination of A with I to the chart

% Example
% -------
%   <2,3> VP --> V . NP PP   <V0,V1> Cat --> Found . [M1|MRest]
% + <3,5> NP --> Det N .     <V1,V2> M1  --> _ .
% ------------------------   ----------------------------------
% = <2,5> VP --> V NP . PP   <V0,V2> Cat --> Found+M1 . MRest

add_edge(V0, V1, Cat, Found, [M1|MRest]) :-
   new_id(ID),
   assert_edge(edge(ID, V0, V1, Cat, Found, [M1|MRest])),
   foreach(edge(InactiveEdgeID, V1, V2, M1, _, []),
     add_edge(V0, V2, Cat,
      /* Found */  [InactiveEdgeID | Found],
      /* ToFind */ MRest)).
```

```
% ------------------------------------------------------------
% assert_edge(+Edge)
% ------------------------------------------------------------
% Adds an edge into the chart with assert. For debugging, the
% added edge is written to the screen as well.

assert_edge(Edge) :-
   asserta(Edge),
   write_edge(Edge).


% ------------------------------------------------------------
% write_edge(+Edge)
% ------------------------------------------------------------
% Writes an edge to the screen.

write_edge(edge(ID, V1, V2, Category, Found, ToFind)) :-
   write(ID), write(':'),
   write('<'), write(V1), write(','),
   write(V2), write('>  '),
   write(Category), write(' --> '), write_list(Found),
   write(' . '), write_list(ToFind),
   nl.

write_list([]).
write_list([First|Rest]) :-
   write(First),
   write(' '),
   write_list(Rest).


% ------------------------------------------------------------
% foreach(+X, +Y)
% ------------------------------------------------------------
% Applies a failure-driven loop to find all possible solutions
% for X. For each one, Y is called once.

foreach(X, Y) :-
   call(X),
   once(Y),
   fail.

foreach(_, _) :-
   true.


% ------------------------------------------------------------
% once(+Goal)
% ------------------------------------------------------------
% Calls Goal once; will fail upon backtracking.

once(Goal) :-
   call(Goal),
   !.
```

```
% -------------------------------------------------------------
% new_id(-ID)
% -------------------------------------------------------------
% Returns a new, unique identification number. Upon each call,
% a different number will be returned.

:- dynamic(last_id/1).
last_id(100).
new_id(Result) :-
   last_id(LastID),
   Result is LastID + 1,
   retract(last_id(LastID)),
   asserta(last_id(Result)).


% -------------------------------------------------------------
% Phrase Structure Rules
% -------------------------------------------------------------

rule(s,  [np, vp]).             % S --> NP VP
rule(np, [det, n]).             % NP --> Det N
rule(vp, [v]).                  % VP --> V
rule(vp, [v, np]).              % VP --> V NP


% -------------------------------------------------------------
% Lexicon
% -------------------------------------------------------------

word(det, the).
word(det, a).
word(n, dog).
word(n, cat).
word(v, chases).
word(v, sees).
word(v, sings).
```

Aufgaben: Chart-Parsing

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999

1. Bottom-Up-Chart-Parser

Tippe das ausgeteilte Programm ab, oder lade den Parser vom World-Wide Web (Adresse: http://www.coli.uni-sb.de/~brawer/botupchart/).

a) Mache Dir klar, wie dieser Parser funktioniert.

b) Verbinde den Parser mit dem Tokenizer aus der ersten Lektion.

c) Das fünfte Argument der edge-Fakten entspricht der eigentlichen Konstituentenstruktur, allerdings in umgekehrter Reihenfolge. Warum ist die Reihenfolge verdreht?

d) In der ausgeteilten Version wird das Ergebnis des Parsings nicht wie sonst üblich als komplexer Prolog-Term zurückgegeben. Schreibe daher ein Prädikat, welches eine Kanten-Nummer entgegennimmt und den entsprechenden Baum zurückliefert.

   Beispiel für die Anwendung:

```
?- as_tree(111, Tree).
Tree = s(np(det(the), n(cat)), vp(v(sings)))
```

e) Erweitere das parse-Prädikat so, dass ein Ableitungs-Baum (als Prolog-Term wie in Teilaufgabe d) zurückgegeben wird, wenn vom angegebenen Nichtterminalsymbol die angegebene Kette aus Terminalsymbolen ableitbar ist. Wenn die Kette nicht ableitbar ist, soll die Anfrage fehlschlagen.

   Beispiele:

```
?- parse(s, [the, cat, sings], Tree).
Tree = s(np(det(the), n(cat)), vp(v(sings)))

?- parse(s, [cat, the, sings], Tree).
no
```

2. Probleme für Bottom-Up-Chart-Parser

Schlage nach, welche Arten von Regeln typischerweise Probleme beim Parsing aufgeben können. Wie kommt der vorgestellte Bottom-Up-Chart-Parser mit den entsprechenden Grammatiken zurecht?

Überlege Dir gegebenenfalls, wie geeignete Korrekturmassnahmen aussehen könnten.

3. Links-nach-Rechts? ?skinL-hcan-sthceR

Parst der in der Vorlesung besprochene Bottom-Up-Chart-Parser von links nach rechts oder von rechts nach links? Was müsste geändert werden, um die andere Richtung zu benutzen?