Earley-Parsing

Übersicht

- Eigenschaften von Earleys Verfahren
- Einzelne Komponenten
  - Initialisierung
  - Predictor
  - Scanner
  - Completer
- Ende des Parsings
- Implementation in Prolog

Ziel

- gutes Verständnis dieses *sehr* wichtigen Verfahrens

Earley-Algorithmus

Von Jay Earley wurde 1970 ein Chart-Parsing-Verfahren vorgestellt.

- parst Sätze mit n Wörtern im schlechtesten Fall in einer Zeit proportional zu n^3 — Verfahren gehört zur Komplexitätsklasse $O(n^3)$
- kommt mit den häufig problematischen Regel-Arten zurecht
  - Tilgungsregeln
  - zyklische Kettenregeln
  - links- und rechts-rekursive Regeln
- Analyserichtung:
  - links-nach-rechts
  - Mischung aus Top-Down und Bottom-Up

Earley-Algorithmus

Von Jay Earley wurde 1970 ein Chart-Parsing-Verfahren vorgestellt.

- kein Backtracking
  - gleichzeitiges Verfolgen aller Alternativen
  - am Ende des Parsings sind sämtliche alternativen Syntaxanalysen in der Chart
- leicht in imperativen Programmiersprachen (C, Modula, etc.) zu implementieren
- von grosser Wichtigkeit bei praktischen Anwendungen der Computerlinguistik
- seit 1970 wurden zahlreiche Verbesserungen gegenüber Earleys Verfahren vorgeschlagen

Earley-Algorithmus

Das Verfahren von Earley besteht im wesentlichen aus drei Komponenten:

- Predictor
- Scanner
- Completer

Nach einer Initialisierungsphase werden diese drei Komponenten für jeden Knoten in der Chart aufgerufen,

- jeweils solange, bis keine neuen Kanten mehr hinzukommen.

Initialisierung

Sei *A* das Startsymbol der Grammatik.

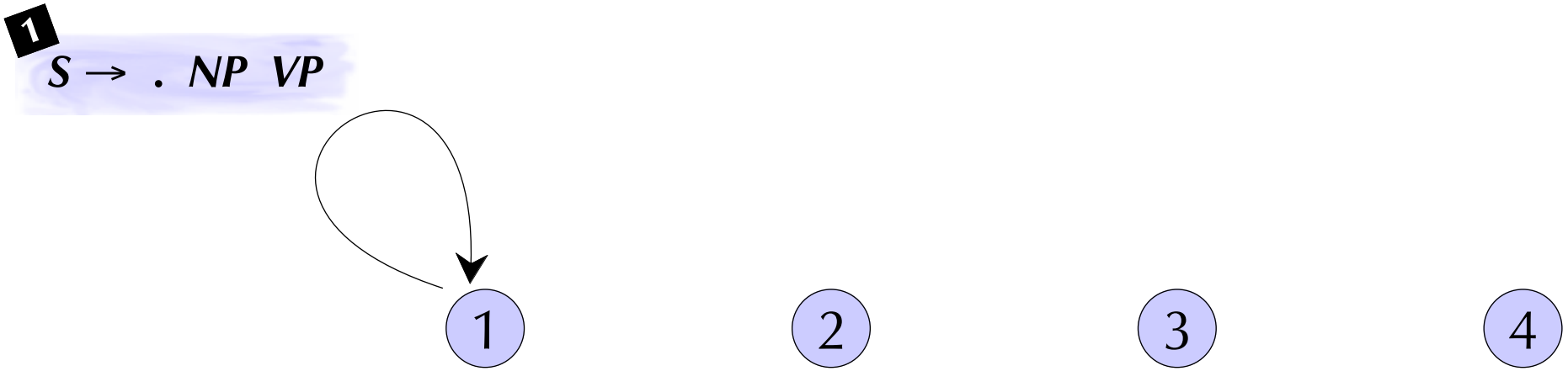
Für jede Grammatikregel, die *A* expandiert (d.h. die von der Form $A \rightarrow \alpha$ ist),

- füge eine Kante $\langle 1, 1 \rangle A \rightarrow . \alpha$ zur Chart hinzu.

Initialisierung: Beispiel

Beispiel-Chart nach der Initialisierung:

- (willkürliche) Annahmen für dieses Beispiel:
  - Startsymbol sei *S*
  - *S* → *NP VP* sei die einzige Regel für *S* in der Grammatik
  - zu analysierende Terminal-Kette: »the cat sings« (→ Chart umfasst vier Knoten)



Predictor

Der Predictor

- expandiert Nichtterminale
- sagt voraus, welche Grammatikregeln zum Ziel führen könnten
- Vorgehensweise: Top-Down

Sei V_k jener Knoten, den der Predictor zu bearbeiten habe.

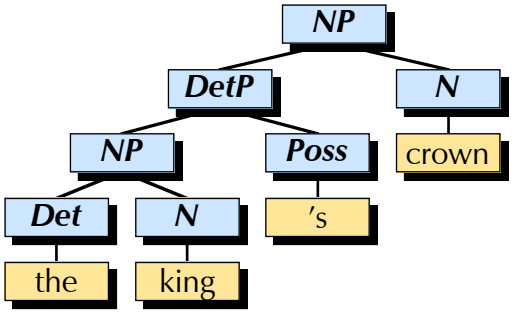
- Für jede Kante der Form $\langle V_i, V_k \rangle\ X \rightarrow \alpha\ .\ B\ \gamma$
- und jede Regel der Form $B \rightarrow \beta$
  - füge eine Kante $\langle V_k, V_k \rangle\ B \rightarrow .\ \beta$ zur Chart hinzu

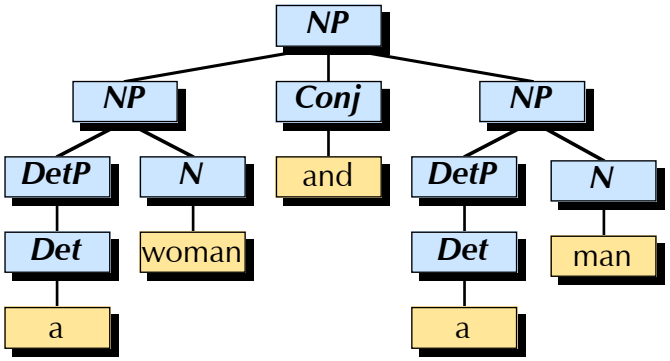
Predictor: Beispiel

Das Beispiel für den Predictor nimmt folgende Grammatikregeln an:

- *S → NP VP*
- *NP → NP Conj NP*
- *NP → DetP N*
- *DetP → Det*
- *DetP → NP Poss*

In der Chart sei bereits eine Kante ⟨1, 1⟩ *S → . NP VP* vorhanden.



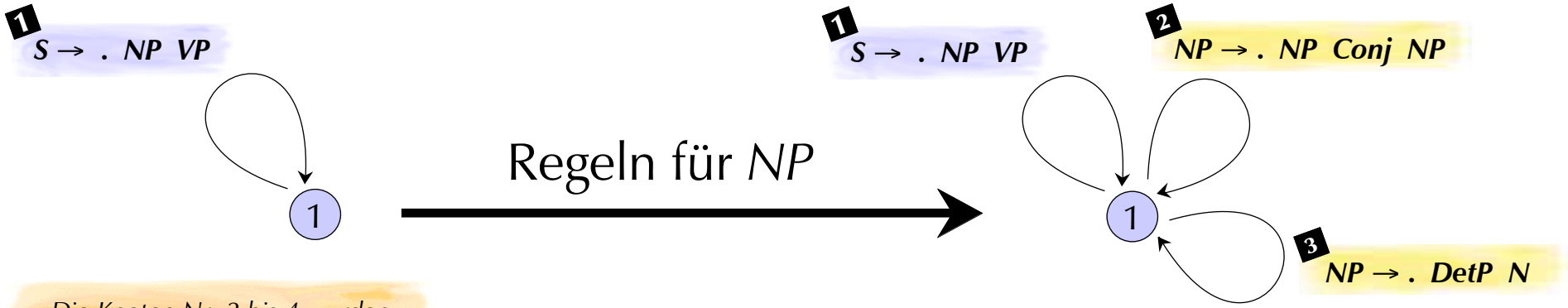


linguistische Motivation für links-rekursive Grammatikregeln

Predictor: Beispiel

Predictor für Knoten Nr. 1, erster Schritt:

- Weil eine aktive Kante eine *NP* benötigt, werden anhand der *NP*-Regeln aktive Kanten in die Chart eingefügt.
- Da unsere Grammatik zwei Regeln für *NP* enthält, werden entsprechend zwei aktive Kanten eingefügt.

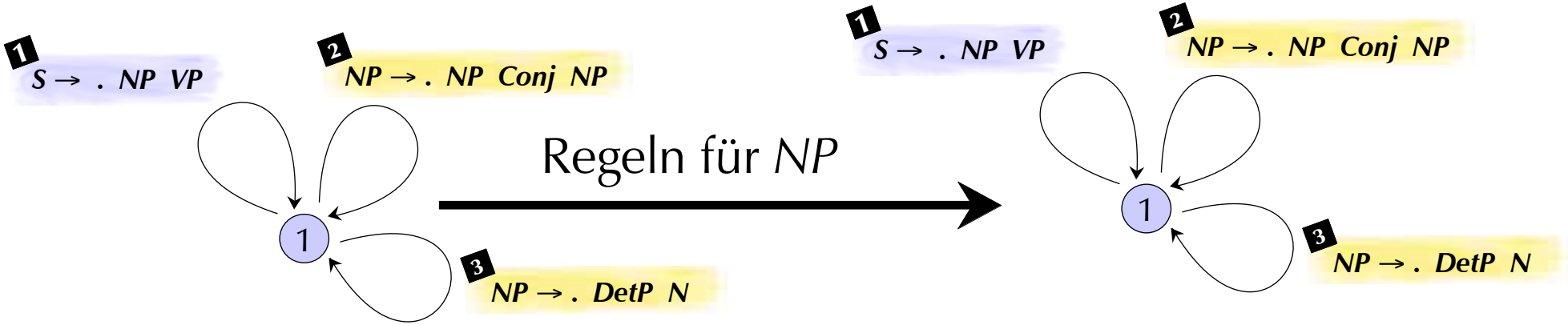


Die Knoten Nr. 2 bis 4 wurden aus Platzgründen weggelassen, sind aber eigentlich in der Chart

Predictor: Beispiel

Predictor für Knoten Nr. 1, zweiter Schritt (1):

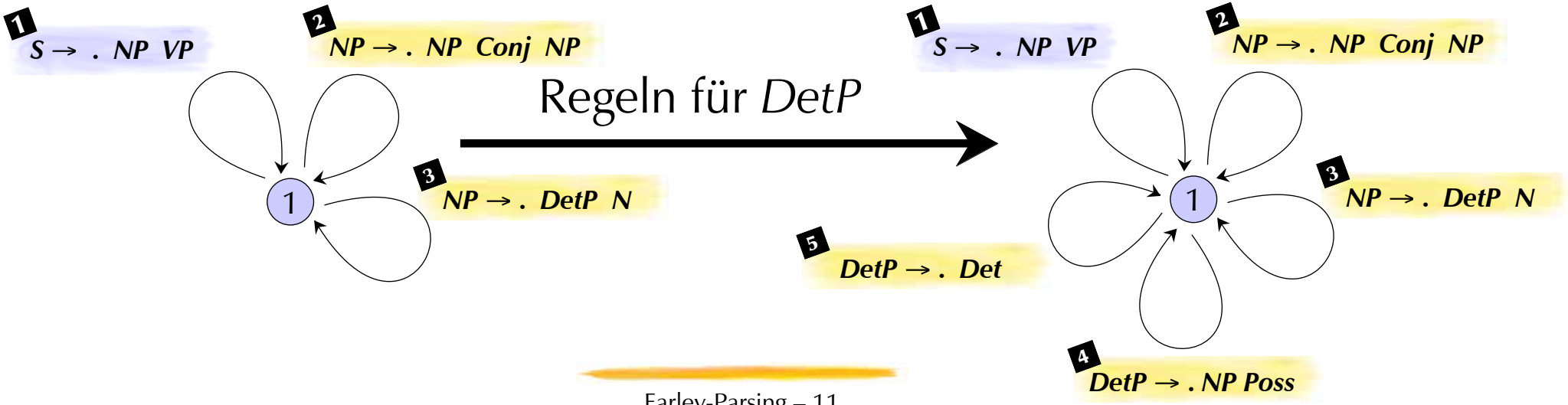
- Die frisch eingefügte Kante Nr. 2 benötigt eine *NP*.
- Daher werden anhand der *NP*-Regeln neue aktive Kanten in die Chart eingefügt.
- Allerdings kommen so keine *neuen* Kanten in die Chart
- → keine endlose Rekursion, trotz linksrekursiver Grammatik!



Predictor: Beispiel

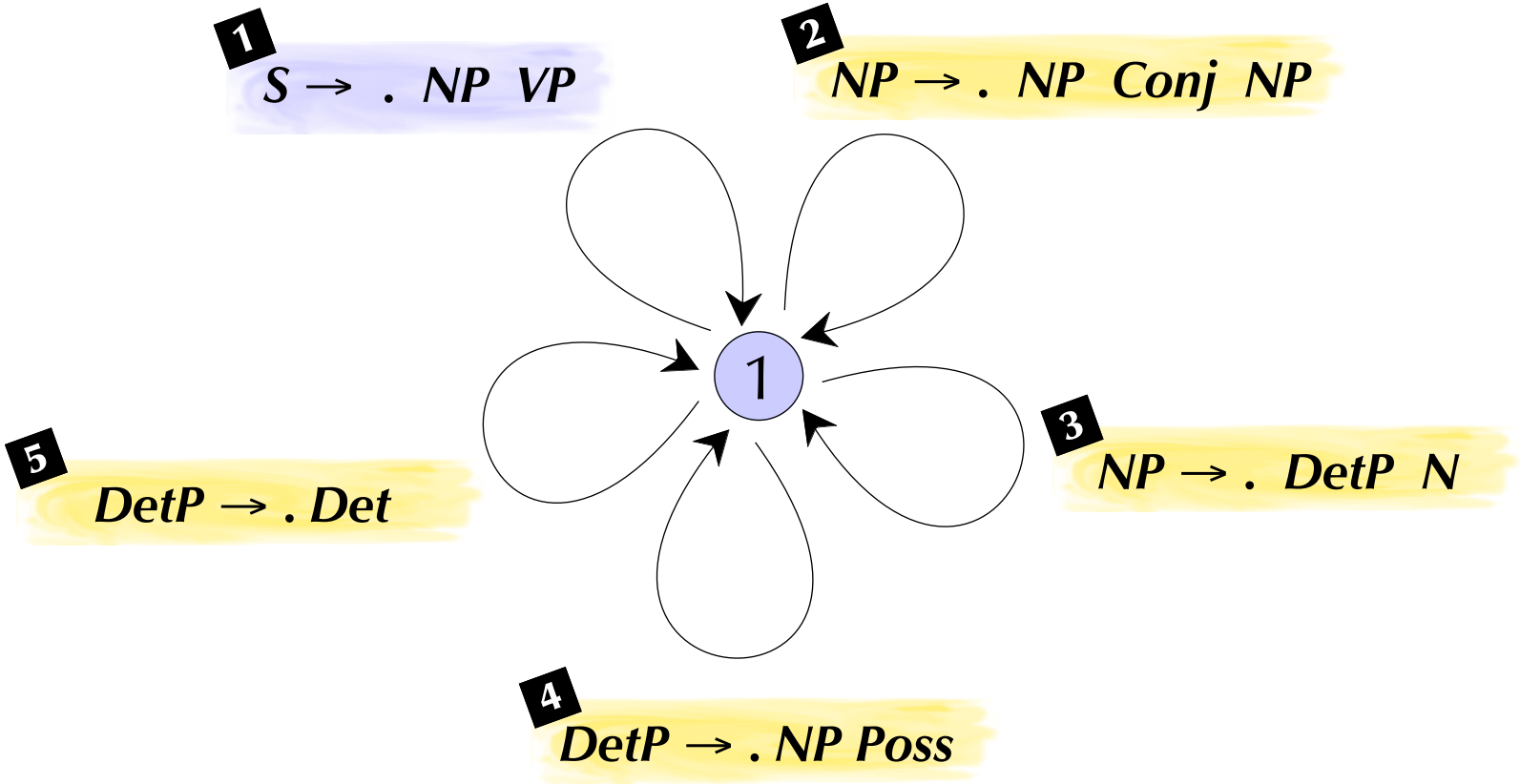
Predictor für Knoten Nr. 1, zweiter Schritt (2):

- Die frisch eingefügte Kante Nr. 3 benötigt eine *DetP*.
- Daher werden anhand der *DetP*-Regeln neue aktive Kanten in die Chart eingefügt.
- Die Grammatik enthält zwei Regeln für *DetP*, also kommen zwei neue aktive Kanten zur Chart hinzu.



Predictor: Beispiel

Chart nach Laufen des Predictors für Knoten Nr. 1:



Scanner

Der Scanner

- konsumiert Terminal-Symbole aus der Eingabekette
- könnte auch eine Schnittstelle zum Lexikon aufrufen (evtl. mit Morphologie-Komponente)
- hier vorgestellt: Verbesserung gegenüber der Original-Version von Earley
  - Earley würde Präterminal-Regeln wie *Det* → *the* vom Predictor verarbeiten lassen
  - das ist ineffizient, weil auch Terminale vorausgesagt werden, die gar nicht Teil der Eingabe-Terminalkette sind
  - bei uns kümmert sich daher der Scanner um solche Regeln

Scanner

Sei V_k jener Knoten, den der Scanner zu bearbeiten habe, und sei w das k-te Terminalsymbol der Eingabekette.

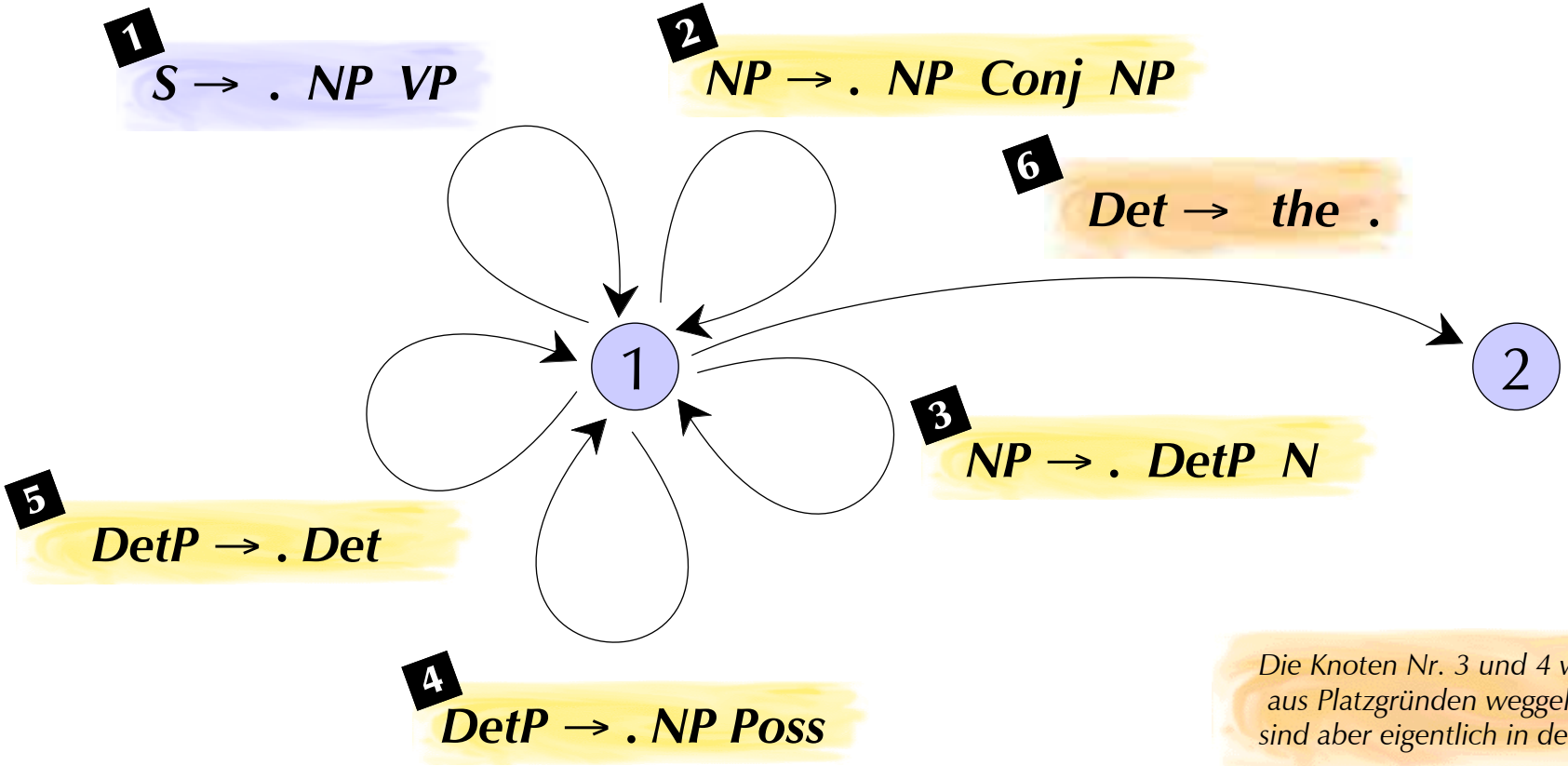
mit anderen Worten: w sei das k-te Token im zu parsenden Satz

- Für jede Regel der Form $B \rightarrow w$
  - füge eine Kante $\langle V_k, V_{k+1} \rangle$ $B \rightarrow w$. zur Chart hinzu

Meistens wird allerdings eine Lexikon-Komponente nach allen Lesarten für w gefragt werden.

Scanner: Beispiel

Chart nach Laufen des Scanners für Knoten 1:



Die Knoten Nr. 3 und 4 wurden aus Platzgründen weggelassen, sind aber eigentlich in der Chart

Completer

Der Completer

- wendet die Fundamental-Regel des Chart-Parsings an
- vervollständigt aktive Kanten durch Kombination mit inaktiven Kanten
- Vorgehensweise: Bottom-Up

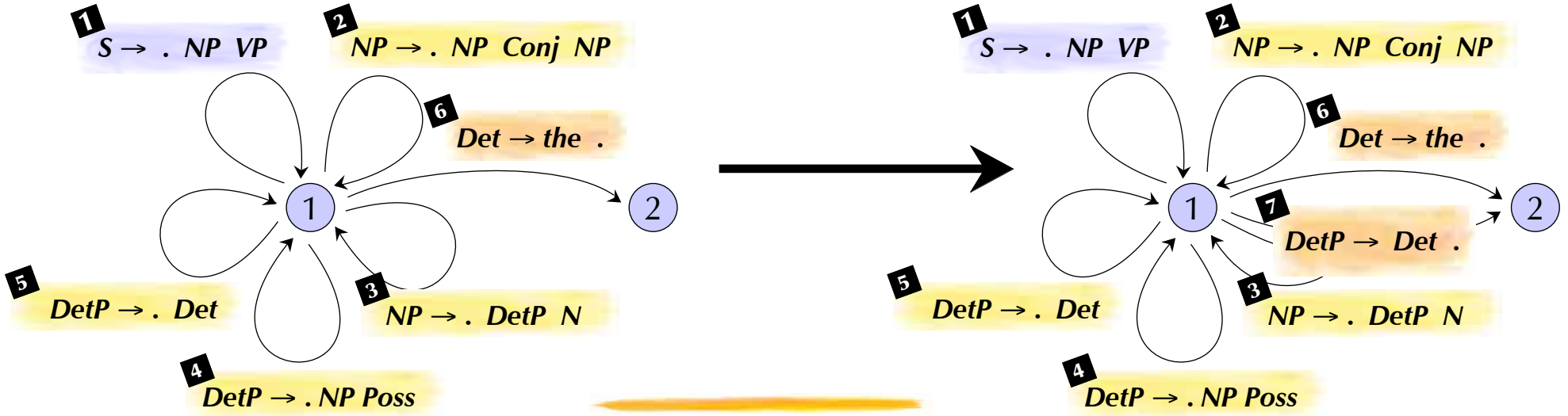
Sei V_k jener Knoten, den der Completer zu bearbeiten habe.

- Für jede [aktive] Kante der Form $\langle V_i, V_j \rangle\ X \rightarrow \alpha \quad .\ B\ \gamma$
- und jede [inaktive] Kante der Form $\langle V_j, V_k \rangle\ B \rightarrow \beta \quad .$
  - füge eine Kante $\langle V_i, V_k \rangle\ X \rightarrow \alpha\ B\ .\ \gamma$ zur Chart hinzu

Completer: Beispiel

Completer für Knoten Nr. 2, erster Schritt:

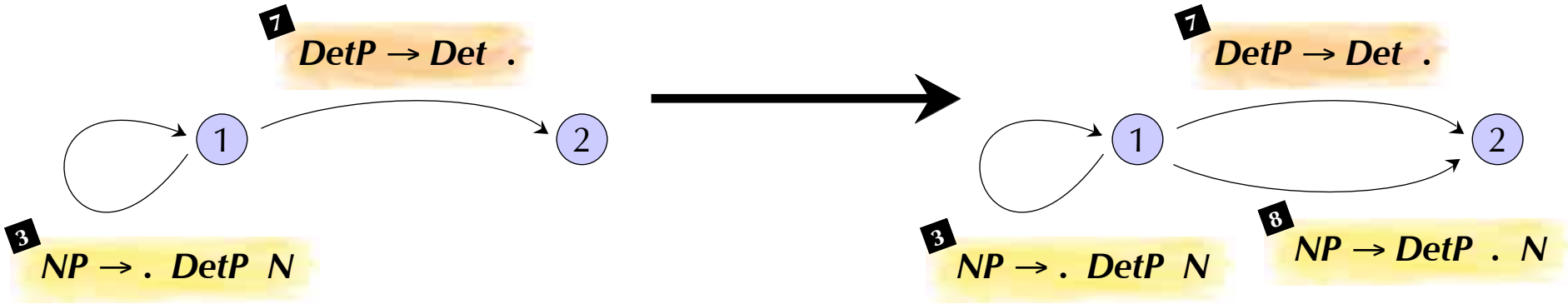
- Inaktive Kante 6 endet hier. Ihre Kategorie ist *Det*
  → suche nach aktiven Kanten,
  - die ein *Det* benötigen
  - und die in Knoten 1 (= Startknoten der inaktiven Kante) enden.
- Kante 4 passt: Fundamentalregel → füge neue Kante (7) zur Chart.



Completer: Beispiel

Completer für Knoten Nr. 2, zweiter Schritt:

- Inaktive Kante 7 endet hier. Ihre Kategorie ist *DetP*
  → suche nach aktiven Kanten,
  - die ein *DetP* benötigen
  - und die in Knoten 1 (= Startknoten der inaktiven Kante) enden.
- Kante 3 passt: Fundamentalregel → füge neue Kante (8) zur Chart.



Earley-Parsing

Wenn der Completer für Knoten 2 nicht mehr weiterkommt …

- Predictor für Knoten 2
- Scanner für Knoten 2
- Completer für Knoten 3
- Predictor für Knoten 3
- Scanner für Knoten 3
- Completer für Knoten 4
- …

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.

Earley-Parsing in Prolog

Nachfolgend wird eine Implementation des Earley-Verfahrens vorgestellt.

- Quelle:
  - Michael A. Covington: Natural Language Processing for Prolog Programmers. Prentice-Hall: Englewood-Cliffs (New Jersey), 1994. ISBN 0-13-629213-5. S. 176 – 190.
- … allerdings wurden einige Änderungen vorgenommen

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]).
```

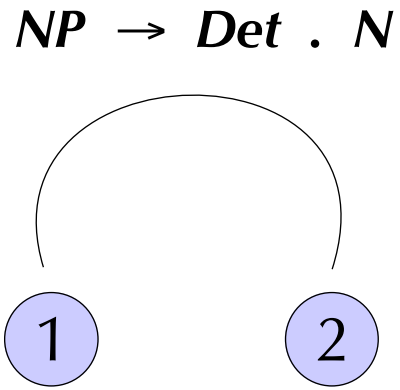
Grammatikregeln

```
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

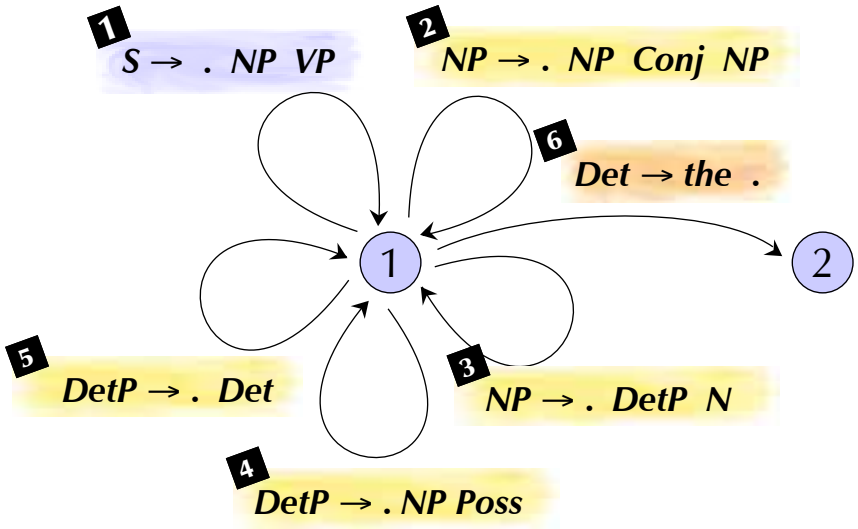
In Prolog geschriebene Chart-Parser verwenden üblicherweise eine von zwei Arten zum Speichern aller Kanten:

- 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: Beispiel



Kanten:

- `edge(1, 1, s, [], [np, vp]).`
- `edge(1, 1, np, [], [np, conj, np]).`
- `edge(1, 1, np, [], [detp, n]).`
- `edge(1, 1, detp, [], [np, poss]).`
- `edge(1, 1, detp, [], [det]).`
- `edge(1, 2, det, [the], []).`

earley.pl

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

```
% ============================================================
% earley.pl
%
% A chart parser which uses Earley's algorithm.
%
% Caution
% -------
% There are two problems with this program:
% - rules whose right-hand side is empty (epsilon rules) do
%   not work
% - the test for edge subsumption is not implemented
%
% These are left as exercises.
%
% Example Query:
% ?- parse(s, [the,cat,sings]).
%
% The program was taken from [Covington, 1994], but it
% has been modified to some extent.
% ============================================================

% Representation of edges
% -----------------------
% All edges (both inactive and active) are asserted as Prolog
% facts edge/5. They have the following form:
%
% edge(StartVertex, EndVertex, Category, Found, ToFind)
% 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 categories which have already been
%              found (i.e. the part "before the dot"),
%              in reversed order (for efficiency reasons)
% 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 (= edges inserted for "the cat sings")
% -----------------------------------------------
% edge(1, 1, s,   [], [np, vp]).       % <1,1> S --> . NP VP
% edge(1, 1, np,  [], [np, conj, np]). % <1,1> NP --> . NP Conj NP
% edge(1, 1, np,  [], [det, n]).       % <1,1> NP --> . Det N
% edge(1, 2, det, [the], []).          % <1,2> Det --> the .
% edge(1, 2, np,  [det], [n]).         % <1,2> NP --> Det . N
% edge(2, 3, n,   [cat], []).          % <2,3> N --> cat .
% edge(1, 3, np,  [n, det], []).       % <1,3> NP --> Det N .
% edge(1, 3, np,  [np], [conj, np]).   % <1,3> NP --> NP . Conj NP
% edge(1, 3, s,   [np], [vp]).         % <1,3> S --> NP . VP
% edge(3, 3, vp,  [], [v]).            % <3,3> VP --> . V
% edge(3, 3, vp,  [], [v, np]).        % <3,3> VP --> . V NP
% edge(3, 4, v,   [sings], []).        % <3,4> V --> sings .
% edge(3, 4, vp,  [v], [np]).          % <3,4> VP --> V . NP
% edge(3, 4, vp,  [v], []).            % <3,4> VP --> V .
% edge(1, 4, s,   [vp, np], []).       % <1,4> S --> NP VP .


% Tell Prolog that edge/5 will be modified dynamically, i.e.
% with assert and retract.
:- dynamic(edge/5).

test :-
   parse(s, [the, cat, sees, a, dog]).

parse(StartSymbol, Sentence) :-
   clear_chart,
   write('Initialization'), nl,
   foreach(
     rule(StartSymbol, RHS),
     add_to_chart(edge(
     /* StartVertex */ 1,
     /* EndVertex */   1,
     /* Category */    StartSymbol,
     /* Found */       [],
     /* ToFind */      RHS))),
   process(Sentence, /* first vertex */ 1, LastVertex),
   !,
   % Test whether there exists any edge which spans the
   % entire chart, is of category StartSymbol and inactive.
   edge(
     /* StartVertex */ 1,
     /* EndVertex */   LastVertex,
     /* Category */    StartSymbol,
     /* Found */       _,
     /* ToFind */      []).


process([], LastVertex, LastVertex).
process([Word|Words], Vertex, LastVertex) :-
   nl, write('Predictor for '), write(Vertex), write(':'), nl,
   predictor(Vertex),

   nl, write('Scanner for '), write(Vertex), write(':'), nl,
   scanner(Word, Vertex, NextVertex),

   nl, write('Completer for '), write(NextVertex), write(':'), nl,
   completer(NextVertex),

   process(Words, NextVertex, LastVertex).
```

```
% ------------------------------------------------------------
% predictor(+Vertex)
% ------------------------------------------------------------
% For every active edge which is ending at Vertex and is look-
% ing for a symbol FirstToFind, predict new active edges of
% category FirstToFind.
%
% Example
% =======
% Assume that <1,3> s --> np . vp is in the chart, and that
% predictor(3) is called. This leads to a call of
% predict(vp, 3) -- which will in turn lead to an insertion
% of <3,3> vp --> . v and <3,3> vp --> . v np

predictor(Vertex) :-
   foreach(
     edge(/*StartVertex*/ _, /* EndVertex */ Vertex,
          /* Category */  _, /* Found */     _,
          /* ToFind*/     [FirstToFind | _]),
     predict(FirstToFind, Vertex)).


% ------------------------------------------------------------
% predict(+Nonterminal, +Vertex)
% ------------------------------------------------------------
% A helper predicate for predictor(+Vertex).
%
% Example
% =======
% Assume: rule(s,[np,vp]). rule(np,[det,n]).
% Upon calling predict(s, 1), the following edges will be added:
% <1,1> s  --> . np vp
% <1,1> np --> . det n

predict(Nonterminal, Vertex) :-
   rule(Nonterminal, [First|Rest]),
   add_to_chart(edge(
     /* StartVertex */ Vertex,
     /* EndVertex */   Vertex,
     /* Category */    Nonterminal,
     /* Found */       [],
     /* ToFind */      [First|Rest])),
   predict(First, Vertex),
   fail.

predict(_,_).
```

```
% ------------------------------------------------------------
% scanner(+Word, +Vertex, -NextVertex)
% ------------------------------------------------------------
% Looks up a word in the lexicon. For every reading of Word,
% a corresponding inactive preterminal edge is inserted
% into the chart.

scanner(Word, Vertex, NextVertex) :-
   NextVertex is Vertex + 1,
   foreach(
     word(Category, Word),
     add_to_chart(edge(
     /* StartVertex */ Vertex,
     /* EndVertex */   NextVertex,
     /* Category */    Category,
     /* Found */       [Word],
     /* ToFind */      []))).
```

```
% ----------------------------------------------------------------
% completer(+Vertex)
% ----------------------------------------------------------------
% For every inactive edge ending at Vertex, complete all
% possible higher constituents.

completer(Vertex) :-
   foreach(
     edge(
     /* StartVertex */ InactiveStart,
     /* EndVertex */   Vertex,
     /* Category */    InactiveCategory,
     /* Found */       _,
     /* ToFind */      []),
     complete(InactiveStart, Vertex, InactiveCategory)).


% ----------------------------------------------------------------
% complete(+InactiveStart, +InactiveEnd, +InactiveCategory)
% ----------------------------------------------------------------
% Applies the Fundamental Rule of Chart Parsing.

complete(InactiveStart, InactiveEnd, InactiveCategory) :-
   edge(
     /* StartVertex */ ActiveStart,
     /* EndVertex */   InactiveStart,
     /* Category */    ActiveCategory,
     /* Found */       ActiveFound,
     /* ToFind */      [InactiveCategory|RestToFind]),
   add_to_chart(edge(
     /* StartVertex */ ActiveStart,
     /* EndVertex */   InactiveEnd,
     /* Category */    ActiveCategory,
     /* Found */       [InactiveCategory|ActiveFound],
     /* ToFind */      RestToFind)),

   % If the result of the completion is an inactive edge,
   % call the completer recursively.
   (/* if this condition holds */ RestToFind == []
    -> complete(ActiveStart, InactiveEnd, ActiveCategory)),

   fail.

complete(_, _, _).
```

```
% ----------------------------------------------------------------
% add_to_chart
% ----------------------------------------------------------------
% Edges are only added to the chart if a "similar" edge is not
% already part of the chart. Note that the definition of
% "similar" is not trivial; many textbooks discuss this
% in depth.
% The implementation of add_to_chart below leads to problems
% with some grammars; a better implementation will be provided
% with the solutions to the exercises.

add_to_chart(Edge) :-
   \+ call(Edge),
   assert_edge(Edge).


% ----------------------------------------------------------------
% clear_chart
% ----------------------------------------------------------------
% Removes all edges from the chart.

clear_chart :-
   retractall(edge(_,_,_,_,_)).


% ----------------------------------------------------------------
% 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(Vfrom, Vto, Category, Found, ToFind)) :-
   write('<'), write(Vfrom), write(','),
   write(Vto), 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),
   !.
```

```
% ---------------------------------------------------------------
% Phrase Structure Rules
% ---------------------------------------------------------------
% Note that these rules do not contain pre-terminal rules,
% for instance "N --> cat". Therefore, the part below is
% not the entire context-free phrase structure grammar.

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


% ---------------------------------------------------------------
% Lexicon
% ---------------------------------------------------------------
% Of course, the lexicon could be a more complicated program,
% for instance it could provide some sort of morphological
% processing.

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

Aufgaben: Earley-Parsing

Programmiertechniken der Computerlinguistik 2 · Sommersemester 1999

1. Earley-Parser

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

Mache Dir klar, wie dieser Parser funktioniert.

2. Kanten-Subsumption

Die nachfolgende Grammatik mit komplexen Nichtterminal-Symbolen wird von [Covington, 1994] als Beispiel angegeben, wieso der Subsumptions-Test bei Chart-Parsern nötig ist. Benutze den Earley-Parser aus der letzten Lektion und lasse den Satz »the cat sees a dog« als S parsen.

```
rule(s,  [np, vp]).
rule(np, [det, n]).
rule(vp, [verbal(0)]).
rule(vp, [verbal(X), rest_of_vp(X)]).
rule(verbal(X), [v(X)]).
rule(rest_of_vp(1), [np]).
rule(rest_of_vp(2), [np, vp]).

word(det, the).
word(det, a).
word(n, dog).
word(n, cat).
word(v(0), sings).    % Verb mit 0 Objekten
word(v(1), chases).   % Verb mit 1 Objekt
word(v(1), sees).     % Verb mit 1 Objekt
word(v(2), gives).    % Verb mit 2 Objekten
```

a) Überlege Dir, warum das Parsing fehlschlägt, obwohl es eigentlich gelingen sollte.

b) Verbessere den Earley-Parser, indem Du den Subsumtions-Test an geeigneter Stelle einbaust.

c) Wieso funktioniert dieselbe Grammatik mit dem Bottom-Up-Chart-Parser aus einer früheren Lektion, obwohl dieser ebenfalls mittels Unifikation testet, ob eine Kante bereits in der Chart ist?

Denke Dir eine Grammatik aus, welche den Bottom-Up-Chart-Parser zu Fall bringt.

3. Repräsentation der Kanten

Der vorgestellte Earley-Parser repräsentiert seine Kanten anders als der früher besprochene Bottom-Up-Chart-Parser. Wo genau liegen die Unterschiede? Welche Repräsentation ist besser?