Topics in Parsing and Generation

Mark Johnson
Martin Kay

ESSLLI '97

1 Introduction

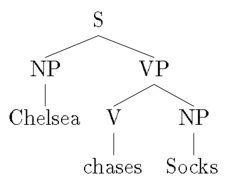
- A *grammar* pairs *surface forms* (strings of words) and *semantic forms*

$$\boxed{\text{Chelsea chases Socks}}$$
$$\updownarrow$$
$$\boxed{\text{chase}'(\text{Chelsea}', \text{Socks}')}$$

- Given one member of such a pair, how can we identify the other?
- Our grammars define these pairs *recursively*.

$$\text{S} \rightarrow \text{NP VP} \quad : \quad \text{S}' = \text{VP}'(\text{NP}')$$

- It is easy to compute these pairs from their recursive structure.



- But searching for this structure is hard!

2 Aside: Prolog notation (Datalog subset)

Atomic Clauses:

```
loves(socks, chelsea).    % "Socks loves Chelsea"
loves(X, X).              % "Everyone loves themselves"
loves(X, Y).              % "Everyone loves everyone"
```

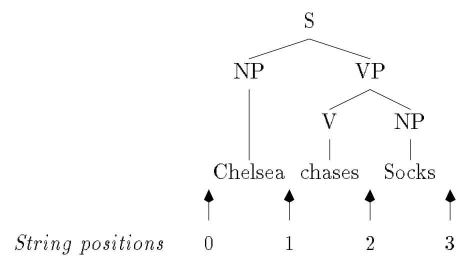
Non-Atomic Clauses:

```
licks(X, Y) :- loves(X, Y), cat(X).
```

% "If X loves Y and X is a cat, then X licks Y".

3 Formalizing Grammars: String Positions

$A(l, r)$ represents a constituent A from l to r.



np(0,1). v(1,2). np(2,3). vp(1,3). s(0,3).

4 Formalizing Grammars: CFG Rules

A CFG production

$$A \rightarrow B_1 \ldots B_n$$

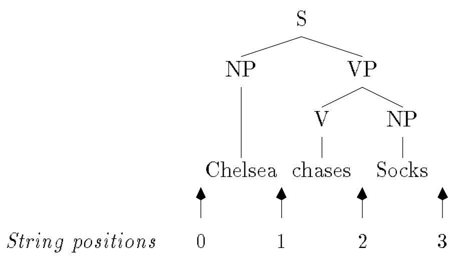
is expressed as the Horn clause

$$\forall x_0 \ldots x_n . A(x_0, x_n) \ \leftarrow \ B_1(x_0, x_1) \wedge \ldots \wedge B_n(x_{n-1}, x_n)$$

or in Prolog notation

```
a(Ws0, Ws) :- b1(Ws0, Ws1), ..., bn(WsN1, Ws).
```

5 Formalizing Grammars: Example




```
s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).          % S → NP VP
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).          % VP → V NP
```

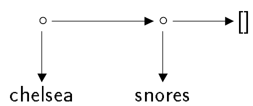
```
np(0, 1).     v(1, 2).     np(2, 3).
```

6 Aside: Prolog notation for lists

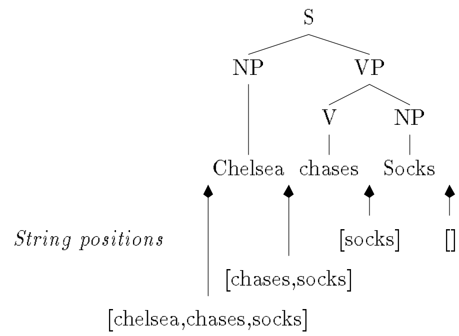
- Two kinds of lists

  The empty list: written [].

  Non empty lists: first element X, remainder Y; written $[X|Y]$.
- Example: [chelsea|[snores|[ ]]]
- Abbreviation: [chelsea,snores]

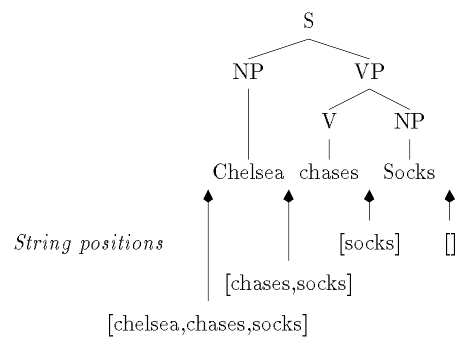


7 String Suffixes as String Positions




```
np([chelsea,chases,socks], [chases,socks]).
v([chases,socks], [socks]).
np([socks], []).
```

8 General Lexical Entries




```
np([chelsea|Ws], Ws).     np([socks|Ws], Ws).
v([chases|Ws], Ws).       v([snores|Ws], Ws).
```

9 Lexical Entries

```
s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).      % S → NP VP
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).      % VP → V NP
```

Integer String Positions

$$\frac{\texttt{np(0,1).} \quad \texttt{v(1,2).} \quad \texttt{np(2,3).}}{\texttt{s(0,3).}}$$

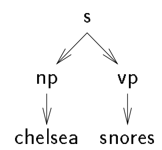
Suffix Strings

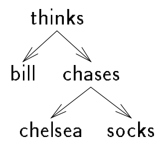
$$\frac{\texttt{np([chelsea|Ws], Ws).} \;\; \texttt{v([chases|Ws], Ws).} \;\; \texttt{np([socks|Ws], Ws).}}{\texttt{s([chelsea,chases,socks], []).}}$$

- The integer string position entries can be regarded as *instantiations* of the general suffix string entries.

10 Aside: Prolog terms

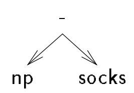
s(np(chelsea,vp(v(snores)))) thinks(bill, chases(chelsea,socks))

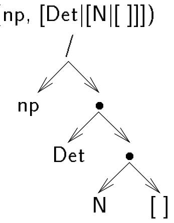




np-socks np/[Det,N]

-(np,socks) /(np, [Det|[N|[ ]]])





11 Aside: Equality and uninterpreted function symbols

- Equality is *identity* in Prolog.
- The only solution that Prolog finds to the query

  $$\mathsf{fatherOf(chelsea) = X}$$

  is

  $$\mathsf{X = fatherOf(chelsea)}$$

  but not

  $$\mathsf{X = bill.}$$

- The only solution that Prolog finds to

  $$\mathsf{f(a,b,c) = f(X,Y,Z)}$$

  is

  $$\mathsf{X = a, Y = b, Z = c.}$$

12 Semantic interpretation

Introduce extra arguments that range over semantic values.

s(*S semantics, Left, Right*).
np(*NP semantics, Left, Right*).
vp(*S semantics, NP_{subj} semantics, Left, Right*).
v(*S semantics, NP_{subj} semantics, NP_{obj} semantics, Left, Right*).

```
s(chase(chelsea,socks),      [chelsea,chases,socks], []).
np(chelsea,                  [chelsea,chases,socks], [chases,socks]).
np(socks,                    [socks], []).

vp(chases(chelsea,socks), chelsea,          [chases,socks], []).
v(chases(chelsea,socks), chelsea, socks,    [chases,socks], [socks]).
```

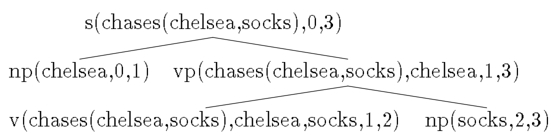
13 Semantic interpretation

```
np(chelsea, [chelsea|Ws], Ws).
```

```
np(socks, [socks|Ws], Ws).
v(chases(Subj,Obj), Subj, Obj, [chases|Ws], Ws).
v(snores(Subj), Subj, [snores|Ws], Ws).

s(S, Ws0, Ws) :-
        np(Subj, Ws0, Ws1),
        vp(S, Subj, Ws1, Ws).

vp(S, Subj, Ws0, Ws) :-
        v(S, Subj, Ws0, Ws).
vp(S, Subj, Ws0, Ws) :-
        v(S, Subj, Obj, Ws0, Ws1),
        np(Obj, Ws1, Ws).
```

14 Proof trees

```
np(chelsea, 0, 1).
v(chases(Subj,Obj), Subj, Obj, 1, 2).
np(socks, 2, 3).

s(S, Ws0, Ws) :-
        np(Subj, Ws0, Ws1), vp(S, Subj, Ws1, Ws).

vp(S, Subj, Ws0, Ws) :-
        v(S, Subj, Obj, Ws0, Ws1), np(Obj, Ws1, Ws).
```



15 A Typology of Parsing and Generation algorithms

Instantiated arguments: String positions in parsing, semantic form in generation.

Selection rule: Left to right, head first.

Proof tree search: Pre-order (top-down), Post-order (bottom-up), In-order (left-corner).

Memoization: Non-deterministic (exponential), Memoizing

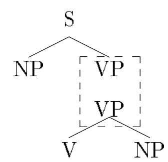
16 Resolution proof rule

$$\frac{A \leftarrow B_1 \wedge \ldots \wedge \underline{B_i} \wedge \ldots \wedge B_m. \quad B \leftarrow C_1 \wedge \ldots \wedge C_n.}{\theta(A \leftarrow B_1 \wedge \ldots \wedge C_1 \wedge \ldots \wedge C_m \wedge \ldots \wedge B_m)}$$

where $\theta = \mathrm{mgu}(B_i, B)$.

- Resolution as "plugging"

$$\frac{\begin{array}{c}\mathsf{s(Ws0,\ Ws)\ \text{:-}\ np(Ws0,\ Ws1),\ vp(Ws1,\ Ws).} \\ \mathsf{vp(Vs0,\ Vs)\ \text{:-}\ v(Vs0,\ Vs1),\ np(Vs1,\ Vs).}\end{array}}{\mathsf{s(Ws0,\ Ws)\ \text{:-}\ np(Ws0,\ Ws1),\ v(Ws1,\ Vs1),\ np(Vs1,\ Ws).}}$$



17 Bottom up, left to right parsing

$$\cfrac{\cfrac{\mathsf{vp(Ws0,\ Ws)\ \text{:-}\ v(Ws0,\ Ws1),\ np(Ws1,\ Ws).} \quad \mathsf{v(1,2).}}{\mathsf{vp(1,\ Ws)\ \text{:-}\ np(2,\ Ws).}} \quad \mathsf{np(2,3).}}{\mathsf{vp(1,3).}}$$

$$\cfrac{\cfrac{\mathsf{s(Ws0,\ Ws)\ \text{:-}\ np(Ws0,\ Ws1),\ vp(Ws1,\ Ws).} \quad \mathsf{np(0,1).}}{\mathsf{s(0,\ Ws)\ \text{:-}\ vp(1,\ Ws).}} \quad \mathsf{vp(1,3).}}{\mathsf{s(0,3).}}$$

- Right to left, and head first can also be implemented using resolution

18 Top down parsing using resolution

$$\frac{\textsf{s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).} \quad \textsf{np(0,1).}}{\textsf{s(0, Ws) :- vp(1, Ws).}}$$

$$\frac{\textsf{s(0, Ws) :- vp(1, Ws).} \quad \textsf{vp(Vs0, Vs) :- v(Vs0, Vs1), np(Vs1, Vs).}}{\textsf{s(0, Ws) :- v(1, Vs), np(Vs, Ws).}}$$

$$\frac{\textsf{s(0, Ws) :- v(1, Vs), np(Vs, Ws).} \quad \textsf{v(1,2).}}{\textsf{s(0, Ws) :- np(2, Ws).}}$$

$$\frac{\textsf{s(0, Ws) :- np(2, Ws).} \quad \textsf{np(2,3).}}{\textsf{s(0, 3).}}$$

- The general resolution rule is *redundant*!

19 Parsing strategies using resolution

- Seek restrictions on resolution which avoid redundancy
- **Soundness** is guaranteed (resolution is sound).
  (*sound* $\equiv$ all conclusions found are correct).
- **Completeness** must be shown for each parsing algorithm
  (*complete* $\equiv$ all correct conclusions are found).
- All standard parsing algorithms are sound and complete.
- Some psycholinguistically motivated algorithms are incomplete.

20 Semantic forms as by-product.

$$\frac{\begin{array}{c}\textsf{vp(S, Subj, Ws0, Ws) :- v(S, Subj, Obj, Ws0, Ws1), np(Obj, Ws1, Ws).}\\ \textsf{v(chases(Subj, Obj), Subj, Obj, 1, 2).}\end{array}}{\textsf{vp(chases(Subj,Obj), Subj, 1, Ws) :- np(Obj, 2, Ws).}}$$

$$\frac{\textsf{vp(chases(Subj,Obj), Subj, 1, Ws) :- np(Obj, 2, Ws).} \quad \textsf{np(socks), 2, 3).}}{\textsf{vp(chases(Subj,socks), Subj, 1, 3).}}$$

$$\frac{\textsf{s(Sem, Ws0, Ws) :- np(Subj, Ws0, Ws1), vp(Sem, Subj, Ws1, Ws).} \quad \textsf{np(chelsea, 0, 1).}}{\textsf{s(Sem, 0, Ws) :- vp(Sem, chelsea, 1, Ws).}}$$

$$\frac{\textsf{s(Sem, 0, Ws) :- vp(Sem, chelsea, 1, Ws).} \quad \textsf{vp(chases(Subj,socks), Subj, 1, 3).}}{\textsf{s(chases(chelsea,socks), 0, 3).}}$$

21 CFG Top down parsing

Goal: To parse *Chelsea chases Socks*

Configurations: *List of predicted categories* $\vdash$ *Remaining Input*

S $\vdash$ NP V NP	*% start symbol*
NP VP $\vdash$ NP V NP	*% S $\rightarrow$ NP VP*
VP $\vdash$ V NP	*% pop NP*
V NP $\vdash$ V NP	*% VP $\rightarrow$ V NP*
NP $\vdash$ NP	*% pop V*
$\vdash$	*% pop NP*

22 Top down parsing

(Selected Literal Deletion, Ancestor resolution)

Input:
- a goal A to prove,
- a set of clauses $\mathcal{P}$, and
- a *selection function s* which maps nonatomic clauses to a member of their body.

Output: A set of instances A' of A such that $\mathcal{P} \vdash A$.

Algorithm: Call **Top-down**($A \leftarrow A$).

Top-down(C): If C is the atomic clause A' then add A' to the list of solutions. Otherwise, resolve each clause in $\mathcal{P}$ with C on $s(C)$ and call **Top-down** on each of the resulting clauses.

23 Top down parsing

```
s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).
vp(Ws0, Ws) :- v(Ws0, Ws).
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).

np(0, 1).  v(1, 2).  np(2, 3).
```

```
s(0,3):-s(0,3)
s(0,3):-np(0,A),vp(A,3)
s(0,3):-vp(1,3)
s(0,3):-v(1,3)    s(0,3):-v(1,B),np(B,3)
                  s(0,3):-np(2,3)
                  s(0,3)
```

24 Top down parsing

```
np(chelsea, [chelsea|Ws], Ws).
np(socks, [socks|Ws], Ws).
v(chases(Subj,Obj), Subj, Obj, [chases|Ws], Ws).
v(snores(Subj), Subj, [snores|Ws], Ws).

s(S, Ws0, Ws) :- np(Subj, Ws0, Ws1), vp(S, Subj, Ws1, Ws).
vp(S, Subj, Ws0, Ws) :- v(S, Subj, Ws0, Ws).
vp(S, Subj, Ws0, Ws) :- v(S, Subj, Obj, Ws0, Ws1), np(Obj, Ws1, Ws).
```

```
s(A,[socks,snores],[])
np(B,[socks,snores],C)
vp(A,B,C,[])
vp(A,socks,[snores],[])
v(A,socks,[snores],[])    v(A,socks,D,[snores],E)
                          np(D,E,[])
A=snores(socks)
```

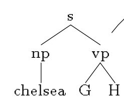

25 Parse trees

- We can return parse trees as an extra argument.
- Preterminals: Label-Terminal, Nonterminals: Label/Subtrees.

```
s(s/[NP,VP], Ws0, Ws) :-          % S → NP VP
        np(NP, Ws0, Ws1),
        vp(VP, Ws1, Ws).

vp(vp/[V], Ws0, Ws) :-            % VP → V
        v(V, Ws0, Ws).
vp(vp/[V,NP], Ws0, Ws) :-         % VP → V NP
        v(V, Ws0, Ws1),
        np(NP, Ws1, Ws).

np(np-chelsea, 0, 1).    v(v-chases, 1, 2).    np(np-socks, 2, 3).
```

26 Parse trees and search trees

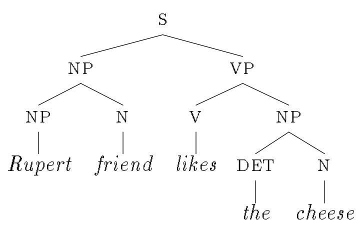
```
s(A,0,3):-s(A,0,3)
s(s/[B,C],0,3):-np(B,0,D),vp(C,D,3)
s(s/[np-chelsea,E],0,3):-vp(E,1,3)
s(s/[np-chelsea,vp/[F]],0,3):-v(F,1,3)    s(s/[np-chelsea,vp/[G,H]],0,3):-v(G,1,I),np(H,I,3)
                                          s(s/[np-chelsea,vp/[v-chases,J]],0,3):-np(J,2,3)
                                          s(s/[np-chelsea,vp/[v-chases,np-socks]],0,3)
```



- The negative literals in a SLD (top down) proof are are children and the right siblings of the ancestors of the last literal expanded.

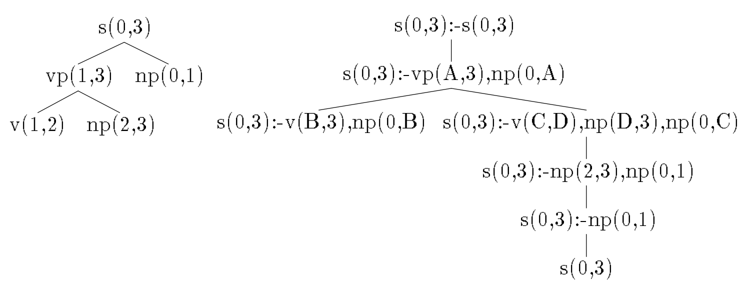
27 Parse trees and search trees example

- The negative literals in a SLD (top down) proof are the children and the right siblings of the ancestors of the last literal expanded.



28 Selection rule and Head first parsing

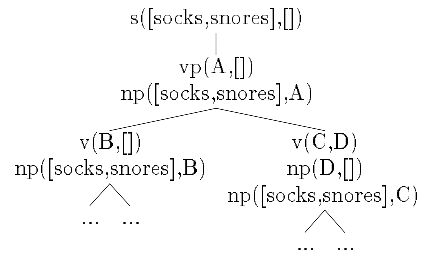
```
s(Ws0, Ws) :- vp(Ws1, Ws), np(Ws0, Ws1).
vp(Ws0, Ws) :- v(Ws0, Ws).
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).

np(0, 1).  v(1, 2).  np(2, 3).
```



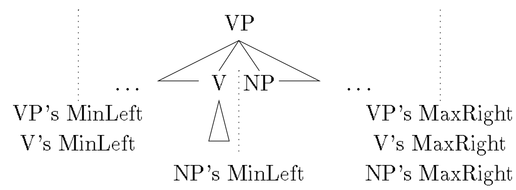
29 Suffix strings and head first parsing

- An instantiated left string position narrows lexical lookup.
- If suffix strings are used as string positions, head first parsing may fail to terminate.

- Integer string positions inherently restrict attention to substrings.



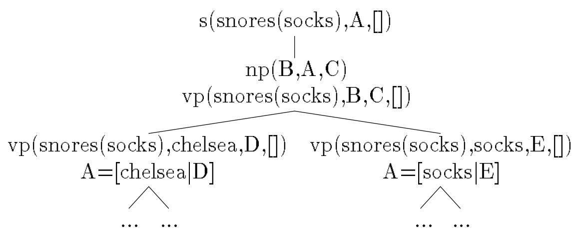
30 Top down prediction for head first parsing




```
vp(Left, Right, MinLeft, MaxRight) :-          % VP → V NP
        v(Left, Mid, MinLeft, MaxRight),
        np(Mid, Right, Mid, MaxRight).

v(1, 2, MinLeft, MaxRight) :- 1 >= MinLeft , MaxRight >= 2.
```

31 Top-down generation

- Integer string position encoding impossible.
- Head first selection seems necessary for termination.



32 Semantic monotonicity

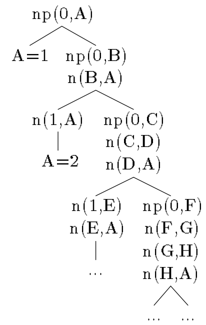
- In our grammars, every constituent's semantic form is a *subterm* of the root S node's semantic form.
- Unify each lexical entry's semantic form with some subterm of the root semantic form (and uniformly replace with integers if desired).
- A semantic counterpart to integer string positions

The goal s(chases(chelsea,socks), Ws0, []) introduces just these lexical entries.

```
v(chases(chelsea,socks), chelsea, socks, [chases|Ws], Ws). % V → chases
np(chelsea, [chelsea|Ws], Ws).                             % NP → chelsea
np(socks, [socks|Ws], Ws).                                 % NP → socks
```

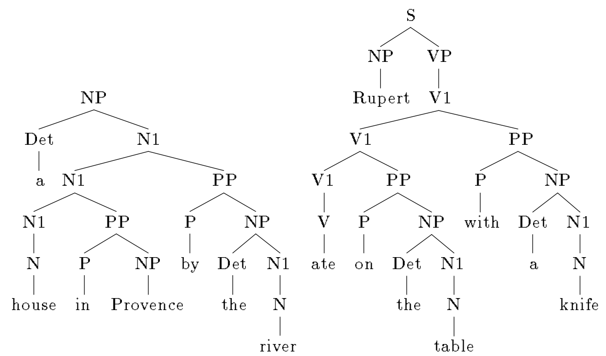
33 Left Recursion and non-termination

```
np(Ws0, Ws) :-          % NP → NP N
    np(Ws0, Ws1),       % as in ``Chelsea's friend''
    n(Ws1, Ws).

np(0, 1).    n(1, 2).
```

34 Head recursion

- Similiar problems occur in head first parsing and generation.
- Post-modification structures are often both head and left recursive.



35 Head recursion in generation

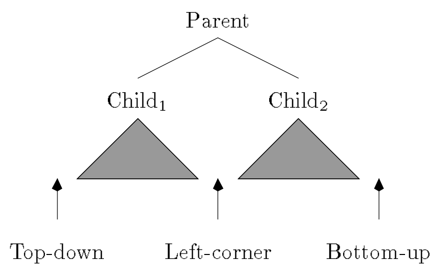
- SUBCAT-list or ARG-list style grammars lead to a head recursion in generation.

```
s(Sem, Ws0, Ws) :-                    % S → NP V1
```

```
          np(Arg, Ws0, Ws1),
          v1(Sem, [Arg], Ws1, Ws).

v1(Sem, Subcat, Ws0, Ws) :-                                  % V1 → V1 NP
          v1(Sem, [Arg|Subcat], Ws0, Ws1),
          np(Arg, Ws1, Ws).

v1(snores(Subj), [Subj], [snores|Ws], Ws).                   % V1 → snores
v1(chases(Subj,Obj), [Obj,Subj], [chases|Ws], Ws).           % V1 → chases
v1(gives(Subj,Obj,Obj2), [Obj2,Obj,Subj], [gives|Ws], Ws).   % V1 → gives
```

36 Search Strategies

	Top-down	**Left-corner**	**Bottom-up**
Traversal:	Pre-order	In-order	Post-order
Enumeration Order:	Parent First subtree Second subtree	First subtree Parent Second subtree	First subtree Second subtree Parent

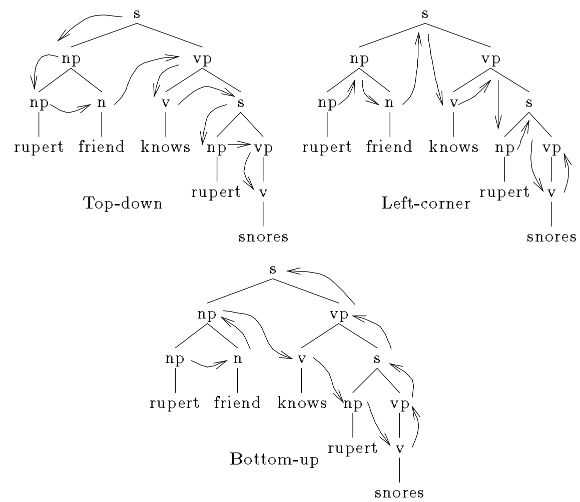


37 Recognition points and node enumeration

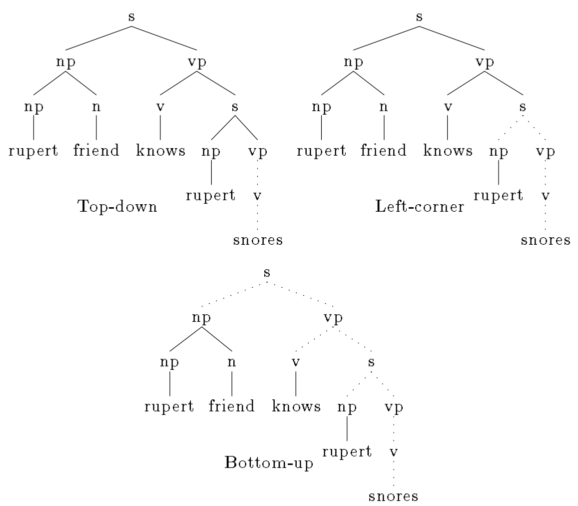
- The recognition points determine the order in which nodes are constructed, and hence the portion of the tree constructed at any point in the parse.

S → NP VP	NP → NP N
VP → V	VP → V S
NP → Rupert	N → friend
V → knows	V → snores

38 Node enumeration order



39 Partial parse trees



40 Partial parses

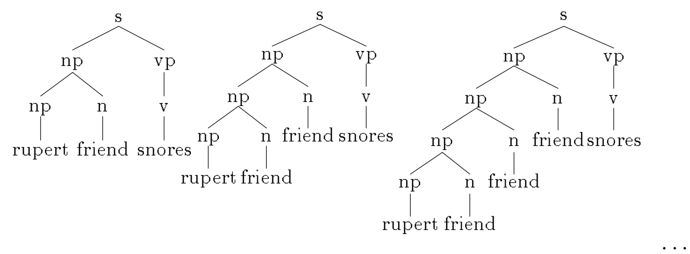
Top down: The nodes remaining to be expanded are the children and the right siblings of the ancestors of the last node expanded.

Bottom up: The nodes to be attached are the left siblings of the ancestors of the last literal reduced.

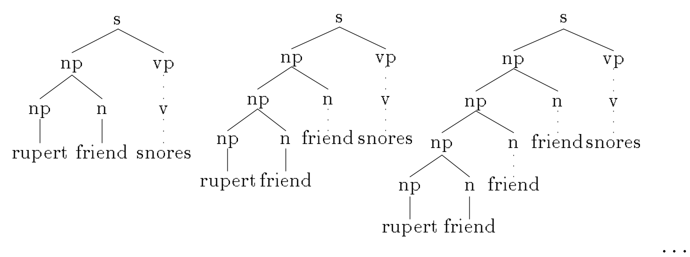
- Top down parsing (and its head first variant) is the only parsing strategy in which the partial parse is always *connected*.
- Connectivity is important, as communication across disconnected components can be difficult in complex grammars.
- But top down does everything possible before looking at next word!

41 Non-termination in top-down parsing

An infinite sequence of trees with the prefix rupert friend.



42 Partial parse trees in top down search

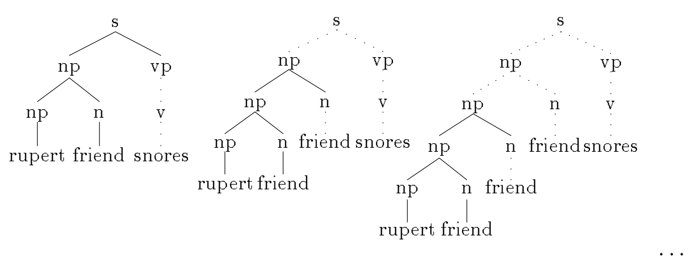


- At *Rupert's friend* a top down parser posits infinitely many partial trees.

⇒ Left recursion is a fundamental problem for top down parsing!

- Fiddling with the selection rule does not help!

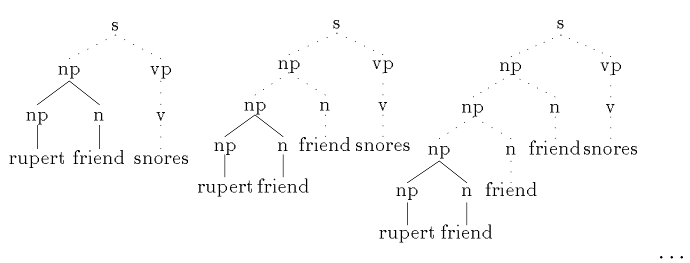
43 Partial parse trees in left corner search



- A left corner search only posits *two* different partial parse trees at *Rupert's friend*.

⇒ Left corner parsing can terminate on left recursive structures.

44 Partial parse trees in bottom up search



- A bottom up search only posits *one* partial parse tree at *Rupert's friend*.

⇒ Bottom up parsing can terminate on left recursive structures.

45 Introduction to left corner parsing

- Left corner and bottom up parsing methods terminate in the face of left recursion.

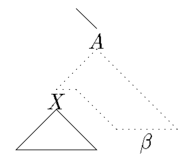
- Left corner parsing can be described by means of a *transformation*, effectively compiling the parsing algorithm.

  A top down parser using $\mathcal{LC}(G)$ simulates a left corner parser using G.

- The transformation can be applied to Horn programs, and so can be used with head first parsing and generation.

- The transformation can be applied *selectively*, permitting mixed top down and left corner parsing.

46 Left corner categories



- The relationship between category A–X in the left-corner transform grammar $\mathcal{LC}_1(G)$ and categories in the original grammar G.

- The dotted part of the tree is dominated by A–X in $\mathcal{LC}_1(G)$, i.e.,

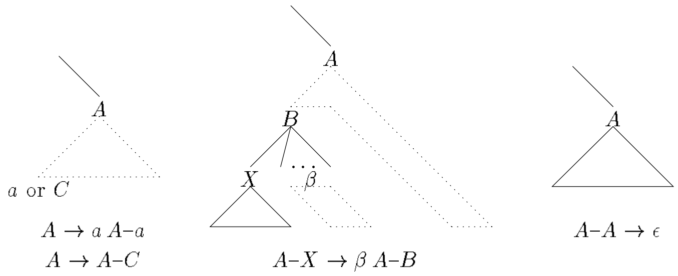
$$A\text{–}X \to^{+} \beta$$

47 Left corner grammar transform

The *left-corner transform* of a CFG $G = (V, P, T, S)$ is a grammar $\mathcal{LC}(G) = (V', P_1, T, S)$, where P_1 contains all of the following productions:

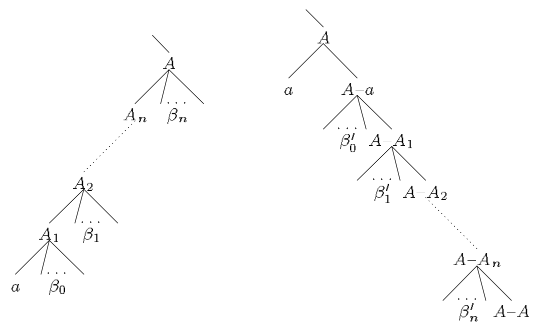
$$\begin{array}{ll} A \to a\, A\text{–}a & \text{for all } A \in V,\ a \in T. \\ A \to A\text{–}C & \text{for all } A \in V,\ C \to \epsilon \in P. \\ A\text{–}X \to \beta\, A\text{–}B & \text{for all } A \in V,\ B \to X\,\beta \in P \\ A\text{–}A \to \epsilon & \text{for all } A \in V. \end{array}$$

48 Left corner productions



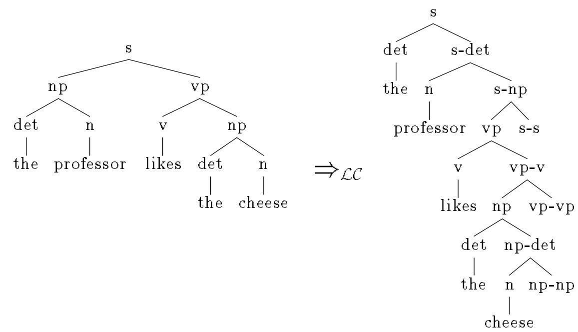
The configurations of nodes of the analysis tree in which the various production schemata of $\mathcal{LC}(G)$ apply. The tree fragments drawn in solid lines would already have been recognized by a top-down recognizer using $\mathcal{LC}(G)$ by the time this production was used, while the dotted tree fragments correspond to predictions still to be instantiated.

49 Left recursion into right recursion



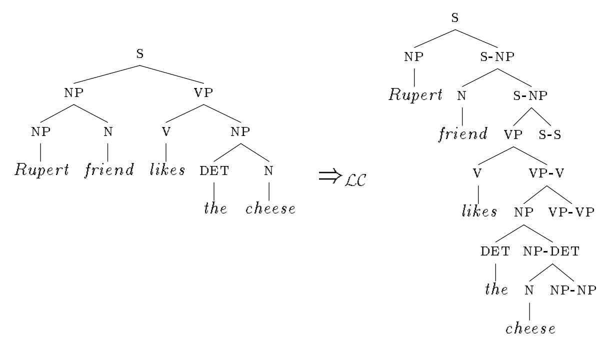
The structural relationship between analysis trees of G (on the left) and parse trees of $\mathcal{LC}_1(G)$ (on the right). In this diagram, the trees β'_i are obtained by applying the left-corner transform to each of the trees β_i.

50 Left corner tree transform example 1



- Every analysis tree node not on a left-most branch corresponds to an empty node in the left corner tree.

51 Left corner tree transform example 2



52 Filtering useless A–X categories

- The grammar $\mathcal{LC}(G)$ usually contains *useless nonterminals*, i.e., nonterminals which can never appear in any complete derivation.

Example: VP → N VP–N

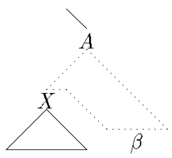
- The observation about left corner categories A–X suggests a pruning.

Define the left corner *link relation* $\triangleleft$ as follows:

$$\mathsf{link0}(A, X) \quad \text{iff} \quad \exists\beta.\, A \to X\beta \in P,$$

Let link be the reflexive and transitive closure of link0. Then a category A–X is *useless* in $\mathcal{LC}_1(G)$ (i.e., derives no sequence of terminals) unless $\mathsf{link}(A, X)$

53 Filtering useless A–X categories



- $\mathsf{link}(A, X)$ iff the dotted part of the tree can actually be constructed using G.
- Reduces the size of the search tree by removing productions from $\mathcal{LC}(G)$.
- Filtering using the link relation does not affect the set of parses found.
- The link relation table is easy to construct for CFGs.
- The link relation can be used even if $\mathcal{LC}(G)$ is not constructed explicitly.

54 Example of link relation

S → NP VP
NP → Det N
VP → V
VP → V NP

```
link(X, X).      % because every category can link with itself
link(s, np).     % S → NP ...
```



```
link(s, det).    % S → NP ..., and NP → Det ...
link(vp, v).     % VP → V ...
link(np, det).   % NP → Det ...
```

55 The link relation

- The link relation is *undecidable* for unification grammars and Prolog programs!
- The link relation propagates *non-local dependencies* across disconnected partial parse tree fragments.
(E.g., WH-dependencies)
- It is especially important in grammars where non-local constraints ensure that the search space is finite (e.g., GB)
- A weaker, *computable* link relation linkc that satisfies

$$\forall A, X.\mathsf{link}(A, X) \to \mathsf{linkc}(A, X)$$

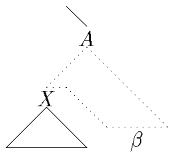
may still provide useful filtering.
(Abstract interpretation and abstraction, a.k.a. restriction)
- One must ensure that the linkc relation does not introduce additional choice points!

56 Partial ϵ-removal

(Also known as: Composition, Tail recursion optimization, Eager attachment, Partial Evaluation)

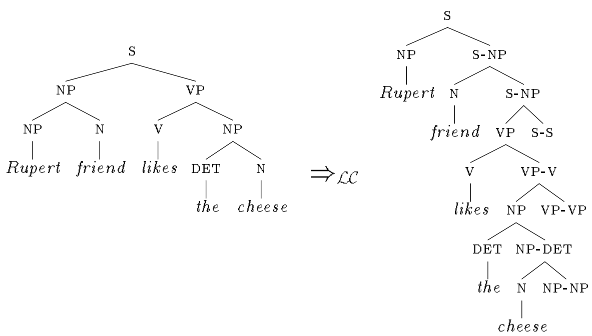
$A \to a\, A\text{–}a$	for all $A \in V, a \in T$.
$A \to A\text{–}C$	for all $A \in V, C \to \epsilon \in P$.
$C \to \epsilon$	for all $C \to \epsilon \in P$.
$A\text{–}X \to \beta\, A\text{–}B$	for all $A \in V, B \to X\,\beta \in P$.
$A\text{–}X \to \beta$	for all $A \to X\,\beta \in P$.

57 Interpretation of A–X categories after ϵ-removal

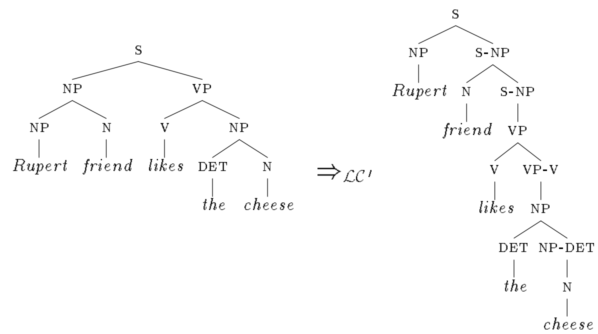


- The dotted portion of the tree must contain one or more nodes
- This affects the link relation: it is the transitive (but not reflexive) closure of the link0 relation (where $\mathsf{link0}(A, X) \iff A \rightarrow X\beta$).

58 Transformed trees without partial ϵ-removal



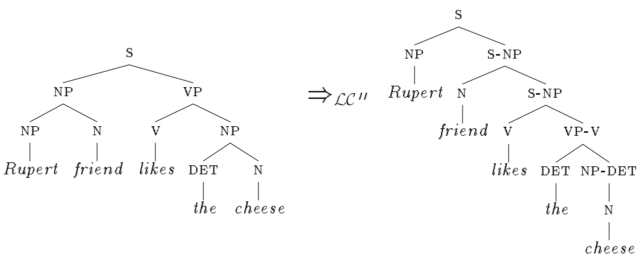
59 Transformed trees with partial ϵ-removal



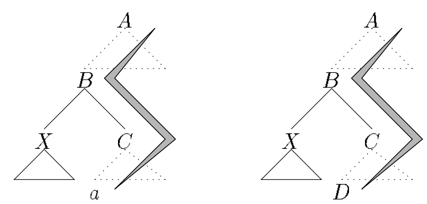
60 Partial evaluation of left corner grammar

$S \rightarrow a\ S\text{–}a$	for all $a \in T$.
$S \rightarrow S\text{–}C$	for all $C \rightarrow \epsilon \in P$.
$S \rightarrow \epsilon$	if $S \rightarrow \epsilon \in P$.
$A\text{–}X \rightarrow A\text{–}B$	for all $A \in V, B \rightarrow X \in P$.
$A\text{–}X \rightarrow \epsilon$	for all $A \rightarrow X \in P$.
$A\text{–}X \rightarrow a\ A\text{–}B$	for all $A \in V, a \in T, B \rightarrow X\ a \in P$.
$A\text{–}X \rightarrow a$	for all $a \in T, A \rightarrow X\ a \in P$.
$A\text{–}X \rightarrow a\ C\text{–}a\ A\text{–}B$	for all $A, C \in V, a \in T, B \rightarrow X\ C \in P$.
$A\text{–}X \rightarrow a\ C\text{–}a$	for all $C \in V, a \in T, A \rightarrow X\ C \in P$.
$A\text{–}X \rightarrow C\text{–}D\ A\text{–}B$	for all $A, C \in V, B \rightarrow X\ C, D \rightarrow \epsilon \in P$.
$A\text{–}X \rightarrow C\text{–}D$	for all $C \in V, A \rightarrow X\ C, D \rightarrow \epsilon \in P$.
$A\text{–}X \rightarrow A\text{–}B$	for all $A \in V, B \rightarrow X\ C, C \rightarrow \epsilon \in P$.
$A\text{–}X \rightarrow \epsilon$	for all $A \rightarrow X\ C, C \rightarrow \epsilon \in P$.

61 Transformed trees after useless node removal



62 Unresolved goals in a left corner parse



A–$X \to a\, C$–$a\ A$–B $\quad$ A–$X \to C$–$D\ A$–B

The highly distinctive "zig-zag" or "lightning bolt" configuration of nodes in the analysis tree characteristic of an unresolved goal in a left corner parse.

63 Selective left corner grammar transform

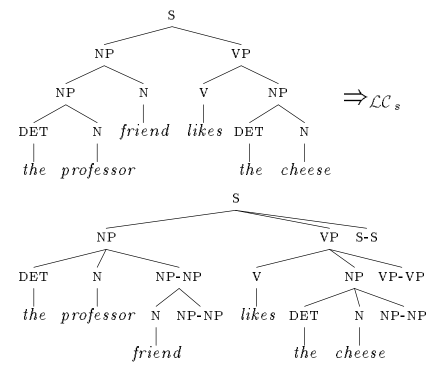
Let $G = (V, P, T, S)$ and let $P = P_{lc} \cup P_{td}$ where P_{lc} contains no epsilon productions. The *selective left-corner transform* of a G with respect to P_{lc} and P_{td} is the grammar $\mathcal{LC}_s(G) = (V', P_1, T, S)$, where P_1 contains all of the following productions:

$A \to a\ A$–a	for all $A \in V,\ a \in T$.
$A \to \beta\, A$–B	for all $A \in V,\ B \to \beta \in P_{td}$.
A–$X \to \beta\ A$–B	for all $A \in V,\ B \to X\,\beta \in P_{lc}$
A–$A \to \epsilon$	for all $A \in V$.

64 Example of selective LC transform

Top down
S → NP VP
VP → V NP
NP → Det N

Left corner
NP → NP N



65 Partial ϵ-removal in selective LC

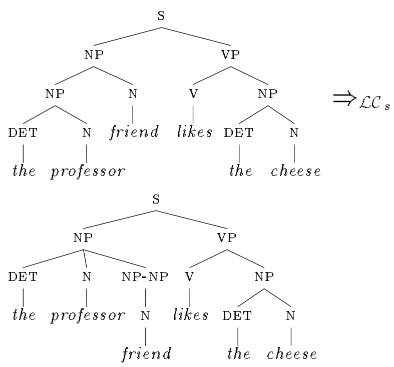
Let $G = (V, P, T, S)$ and let $P = P_{lc} \cup P_{td}$ where P_{lc} contains no epsilon productions. The *selective left-corner transform* with partial ϵ-removal of a G with respect to P_{lc} and P_{td} is the grammar $\mathcal{LC}_s(G) = (V', P_1, T, S)$, where P_1 contains all of the following productions:

$A \to a\ A$–a	for all $A \in V,\ a \in T$.
$B \to \beta$	for all $B \to \beta \in P_{td}$.
$A \to \beta\, A$–B	for all $A \in V,\ B \to \beta \in P_{td}$.
B–$X \to \beta$	for all $B \to X\,\beta \in P_{lc}$
A–$X \to \beta\ A$–B	for all $A \in V,\ B \to X\,\beta \in P_{lc}$

66 Example of selective LC transform with partial ϵ-removal

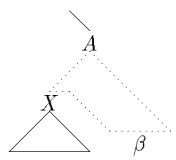
Top down
S → NP VP
VP → V NP
NP → Det N

Left corner
NP → NP N



67 Application of selective LC transform

- Head corner generation. Head recursive "chain" rules recognized left corner, non recursive "non-chain" rules recognized top-down.
- With partial ϵ-removal, tree fragments recognized top down are always connected. This simplifies the link relation, since remaining left corner categories only involve left corner rules.



The link relation is the transitive closure of the link0 relation, where $\mathsf{link0}(A, X) \iff A \to X\beta \in P_{lc}$.

68 Selective left corner program transform

Input: A set of Horn clauses $\mathcal{P}_{td}$ and a set of non-atomic Horn clauses $\mathcal{P}_{lc}$.

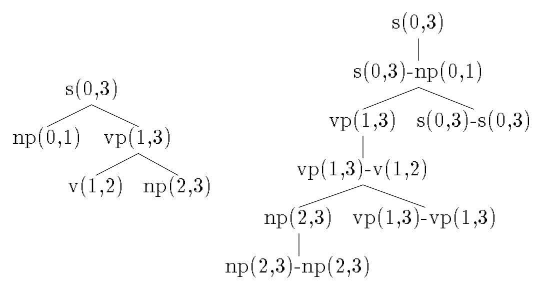
Output: A set of Horn clauses $\mathcal{P}'$ for which each SLD proof using $\mathcal{P}'$ corresponds to a selective left-corner proof using $\mathcal{P}_{td} \cup \mathcal{P}_{lc}$.

$$
\begin{aligned}
\mathcal{P}_{tdx} &= \{\ A(x) \leftarrow B_1(t_1), \ldots, B_n(t_n), A\_B(x,t) \\
&\qquad : B(t) \leftarrow B_1(t_1), \ldots, B_n(t_n) \in \mathcal{P}_{td};\ \} \\
\mathcal{P}_{lcx} &= \{\ A\_B_1(x,t_1) \leftarrow B_2(t_2), \ldots, B_n(t_n), A\_B(x,t) \\
&\qquad : B(t) :\text{-}\ B_1(t_1), B_2(t_2), \ldots B_n(t_n) \in \mathcal{P}_{lc};\ \} \\
\mathcal{P}_{term} &= \{\ A\_A(X,X) \}
\end{aligned}
$$

$$\mathcal{P}' = \mathcal{P}_{tdx} \cup \mathcal{P}_{lcx} \cup \mathcal{P}_{term}$$

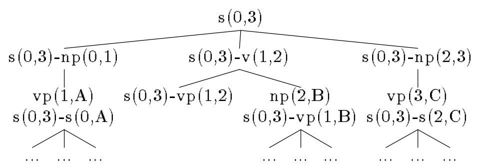
69 Left corner transform proof trees

```
np(0, 1).  v(1, 2).  np(2, 3).

s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).     % Left first!
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).
vp(Ws0, Ws) :- v(Ws0, Ws).
```



70 Left corner transform search tree

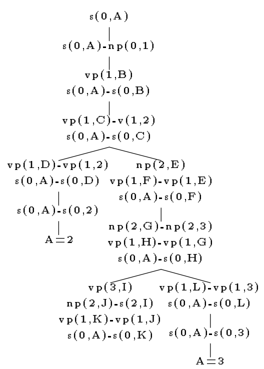
```
link(_Ancestor, _Descendant).          % Any atoms may link
```



- Without link constraints, lexical items are picked at random and combined in a *blind* bottom up fashion.
- Without memoization, this is done for every constituent!

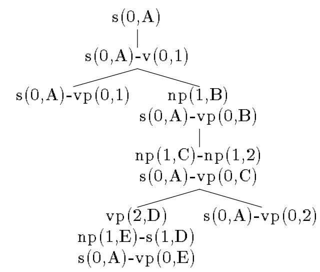
71 Left corner search tree

```
 % Atoms sharing first string position may link
linked(C1, C2) :- arg1(C1, A), arg1(C2, A).

arg1(s(Ws0,_Ws), Ws0).    arg1(np(Ws0,_Ws), Ws0).
arg1(vp(Ws0,_Ws), Ws0).   arg1(v(Ws0,_Ws), Ws0).
```



72 Extraneous search with weak link relation

```
v(0, 1).   np(1, 2).                              % No subject NP

s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).      % Needs a subject NP!
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).
vp(Ws0, Ws) :- v(Ws0, Ws).
```

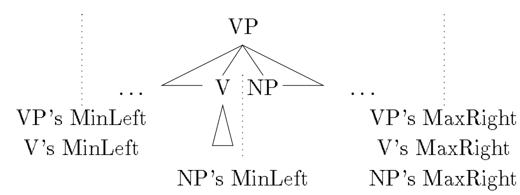


73 Restrictive link relation

```
s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).      % Needs a subject NP!
vp(Ws0, Ws) :- v(Ws0, Ws1), np(Ws1, Ws).
vp(Ws0, Ws) :- v(Ws0, Ws).

link(A, A).   link(s(Ws0,_), np(Ws0,_)).    link(vp(Ws0,_), v(Ws0,_)).
```

- The search tree has only one node, as the left corner category s(0,A)-v(0,1) does not satisfy the link relation.

74 Top down prediction for head first parsing (review)



```
vp(Left, Right, MinLeft, MaxRight) :-            % VP → V NP
        v(Left, Mid, MinLeft, MaxRight),
        np(Mid, Right, Mid, MaxRight).

v(1, 2, MinLeft, MaxRight) :- 1 >= MinLeft , MaxRight >= 2.
```

75 Head corner parsing

- Head first selection rule.
- Top down prediction is essential for termination!

```
v(1, 2, MinLeft, MaxRight) :- 0 >= MinLeft, MaxRight >= 1.

s(Left, Right, MinLeft, MaxRight) :-             % S → NP VP
        vp(Mid, Right, MinLeft, MaxRight),
        np(Left, Mid, MinLeft, Mid).
```

- Linkage reflects proof structure (rather than phrase structure), so categories are linked with their heads.
- link predicate must propagate top down predictions.

```
link(X, X).
link(s(_, _, MinLeft, MaxRight), vp(_, _, MinLeft, MaxRight)).
link(s(_, _, MinLeft, MaxRight), v(_, _, MinLeft, MaxRight)).
```

76 Head corner parsing proof tree

```
                    s(0,3,0,3)
                        |
               s(0,3,0,3)-v(1,2,0,3)
              /                      \
      np(2,3,2,3)                s(0,3,0,3)-vp(1,3,0,3)
          |                       /                  \
np(2,3,2,3)-np(2,3,2,3)    np(0,1,0,1)      s(0,3,0,3)-s(0,3,0,3)
                               |
                    np(0,1,0,1)-np(0,1,0,1)
```


77 Head corner parsing search tree

```
                          s(0,A,0,3)
                              |
                    s(0,A,0,3)-v(1,2,0,3)
                   /                     \
  s(0,A,0,3)-vp(1,2,0,3)               np(2,B,2,3)
            |                     s(0,A,0,3)-vp(1,B,0,3)
        np(0,1,0,1)                        |
  s(0,2,0,3)-s(0,2,0,3)           np(2,3,2,3)-np(2,3,2,3)
           A=2                    s(0,A,0,3)-vp(1,3,0,3)
            |                              |
 np(0,1,0,1)-np(0,1,0,1)          s(0,A,0,3)-vp(1,3,0,3)
  s(0,2,0,3)-s(0,2,0,3)                    |
           A=2                         np(0,1,0,1)
            |                     s(0,3,0,3)-s(0,3,0,3)
  s(0,2,0,3)-s(0,2,0,3)                   A=3
           A=2                             |
            |                     np(0,1,0,1)-np(0,1,0,1)
           A=2                    s(0,3,0,3)-s(0,3,0,3)
                                          A=3
                                           |
                                  s(0,3,0,3)-s(0,3,0,3)
                                          A=3
                                           |
                                          A=3
```

78 Head corner generation

- Head first selection rule, suffix string position encoding

```
s(Sem, Ws0, Ws) :- v1(Sem, [Subj], Ws1, Ws), np(Subj, Ws0, Ws1).
```

- Selective left corner
  - Semantically vacuous, head recursive "chain rules" recognized left corner

```
v1(Sem, Subcat, Ws0, Ws) :-
        v1(Sem, [Arg|Subcat], Ws0, Ws1),
        np(Arg, Ws1, Ws).
```

  - Semantically potent "non-chain rules" recognized top down

```
v1(chases(Subj,Obj), [Obj,Subj], [chases|Ws], Ws).
```

Productions which introduce semantic material (e.g., quantifier scoping rules) are also recognized top down

- Because only semantically vacuous rules are recognized left corner, the left corner category always has the same semantics as the ancestor. Thus the link relation can propagate semantic forms directly.

```
link(s(Sem, _, _), v1(Sem, _, _, _)).
```

79 Memoization: The basic idea

- Never do anything twice.
- Instead, remember everything you ever do, and
- recall the result (instead of redoing it) whenever you try to do it again.
- If there are only a *finite* number of *different* things you might do, eventually you will halt.
- If only it were so simple!

80 A Rose by Any Other Name ...

Chart parsing (computational linguistics)

Dynamic programming (theory of algorithms)

Memoization (logic programming)

- Exponential proof methods (e.g., top down, left corner) have tree structured search spaces.
- Memoizing proof methods have a graph structured search space.
- The reentrancies makes constraint manipulation difficult.

81 Bottom-up parsing

((semi-)naive evaluation)

Input:
- a set of clauses $\mathcal{P}$, and
- a *selection function s* which maps nonatomic clauses to a member of their body.

Output: A set of atoms A such that $\mathcal{P} \vdash A$.

Algorithm schema: Set $\Lambda = \mathcal{P}$.

For each atomic clause A and non atomic clause C in Λ
  If C resolves with A on $s(C)$ then add the resulting clause to Λ and repeat.

Return Λ.

82 Bottom-up example

```
s(Ws0, Ws) :- np(Ws0, Ws1), vp(Ws1, Ws).    % 1
vp(Vs0, Vs) :- v(Vs0, Vs1), np(Vs1, Vs).    % 2
np(0, 1).                                   % 3
s(0, Ws) :- vp(1, Ws).                      % 4 = 1+3
v(1, 2).                                    % 5
vp(1, Vs) :- np(2, Vs).                     % 6 = 2+5
np(2, 3).                                   % 7
vp(1, 3).                                   % 8 = 6+7
s(0, 3).                                    % 9 = 4+8
```

83 Bottom up generation

```
np(chelsea, [chelsea|Ws], Ws).
v(chases(Subj,Obj), Subj, Obj, [chases|Ws], Ws).
```

- With general lexical entries, "blind" bottom up parsing or generation does not terminate.
- We must instantiate the lexical entries in order to restrict the search
  - Parsing: integer string positions
  - Generation: constrain semantic values via semantic monotonicity
    `v(chases(chelsea,socks), chelsea, socks, [chases|Ws], Ws).`

  These both provide "top-down" filtering for bottom-up search
- Top down filtering: magic transform.

84 Lemma table proof procedure

(Johnson and Dörre, 1995 Proc. ACL)

- Works simultaneously on several *goals*
- Each goal is simplified by resolving with clauses from program
- Goals can be *selectively memoized*.
- Paper describes how to coroutine constraints through memoized goals.

85 Overview of memoizing program

- Each memoized goal g is associated with a unique integer i called its index
- The assignment of indices to memoized goals is recorded by asserting an atom $\mathsf{goal\_index}(g, i)$ into the Prolog database.
- As each solution s for g is found, an atom $\mathsf{index\_solution}(i, s)$ is asserted into the database.
- An *item* is a pair $i\#c$ of an index i and a clause c.
- The clause c is a consequence of the (object-level) program $\mathcal{P}$, and the index records the fact that c's positive literal is an instance of the memoized goal whose index is i.
- *Complete items* are items whose clause contains no negative literals
- *Incomplete items* are items with one or more negative literals.
- A *selection rule* associates each incomplete item with a *selected literal* which is one of its clause's negative literals.
- $\mathsf{index\_parent}(i, e)$ records the fact that the subgoal indexed by i was created to resolve with e.

86 Structure packing – overview

- There may be exponentially many different analyses of a single input
  - Parsing: PP attachment

  - Generation: Adjunction ordering
- Standard proof methods enumerate all analyses (exponential time)
- If there are only polynomially different constituents, a memoizing technique can find them all in polynomial time
- Question: Can we construct a representation of these analyses in polynomial time?
  - Distinguish *syntactically potent* and *structure-recording* features
  - Record how each constituent is produced

87 Ambiguity in parsing and generation

- Parsing: PP attachment, Relative clause attachment

  a [book [on the table] [near the door]]
  a [book [on the [table near the door]]]

- Generation: Adjunct ordering

  a [book [on the table] [with a red cover]]
  a [book [with a red cover] [on the table]]

88 Two uses for features

- **Syntactically potent features:** change of value affects grammaticality
  - Case features, agreement features, etc.
- **Structure recording features:** feature value is output of analysis
  - Semantic value in parsing, string in generation
  - A change in value should *not* affect grammaticality
- These two uses of features are not usually distinguished in grammatical theories ...
- but there may be processing advantages in separating them!

89 Syntactically potent vs. Structure recording features

- Usually *syntactically potent* features range over a *finite* set of values, while
- *structure recording* features grow recursively with the complexity of the utterance.
- Exploiting the difference
  - Divide rules into two parts
  - Compute syntactically potent features during analysis
    - Finite feature set ⇒ finite number of distinct constituents
  - Record the rules used to construct each constituent
  - Use this to construct structure representations

90 Analysis by instantiation

- Initial grammar clauses

```
s(Syn, s/[NP,VP], Ws0, Ws) :-
        np(Syn, NP, Ws0, Ws1), vp(Syn, VP, Ws1, Ws).
```

- Instantiate string positions and syntactically potent features

```
s(3sg, s/[NP,VP], 0, 10) :- np(3sg, NP, 0, 4), vp(3sg, VP, 4, 10).
s(3sg, s/[NP,VP], 0, 10) :- np(3sg, NP, 0, 5), vp(3sg, VP, 5, 10).
s(3sg, s/[NP,VP], 0, 10) :- np(3sg, NP, 0, 6), vp(3sg, VP, 6, 10).
...
```

- Delete useless clauses

```
s(3sg, s/[NP,VP], 0, 10) :- np(3sg, NP, 0, 4), vp(3sg, VP, 4, 10).
s(3sg, s/[NP,VP], 0, 10) :- np(3sg, NP, 0, 6), vp(3sg, VP, 6, 10).
```

- The remaining clauses, when evaluated, enumerate all possible tree structures

91 Packed parse representations

- Replace atomic clauses in memoizing proof


```
np(np-chelsea, 0, 1).
vp(vp/[v-chases,np-socks], 1, 3).
```

with

```
np(NP, 0, 1) :- np_0_1(NP).
vp(VP, 1, 3) :- vp_1_3(VP).
np_0_1(np-chelsea).
vp_1_3(vp/[v-chases,np-socks]).
```

- Now resolve these with clause

```
s(s/[NP,VP], Ws0, Ws) :- np(NP, Ws0, Ws1), vp(VP, Ws1, Ws).
```

yielding

```
s(s/[NP,VP], 0, 3) :- np_0_1(NP), vp_1_3(VP).
```

- and introduce a new clause

```
s(S, 0, 3) :- s_0_3(S).
s_0_3(s/[NP,VP]) :- np_0_1(NP), vp_1_3(VP).
```

- If there were several ways of finding an S from 0 to 3, we would have several clauses for s_0_3.

```
s(S, 0, 3) :- s_0_3(S).
s_0_3(s/[NP,VP]) :- np_0_1(NP), vp_1_3(VP).
s_0_3(s/[NP,VP]) :- np_0_2(NP), vp_2_3(VP).
```

92 Open questions

- Even if you successfully pack an exponential number of structure representations into a polynomial sized representation ...
- what do you do with the exponential number of representations?