Deforesting LF

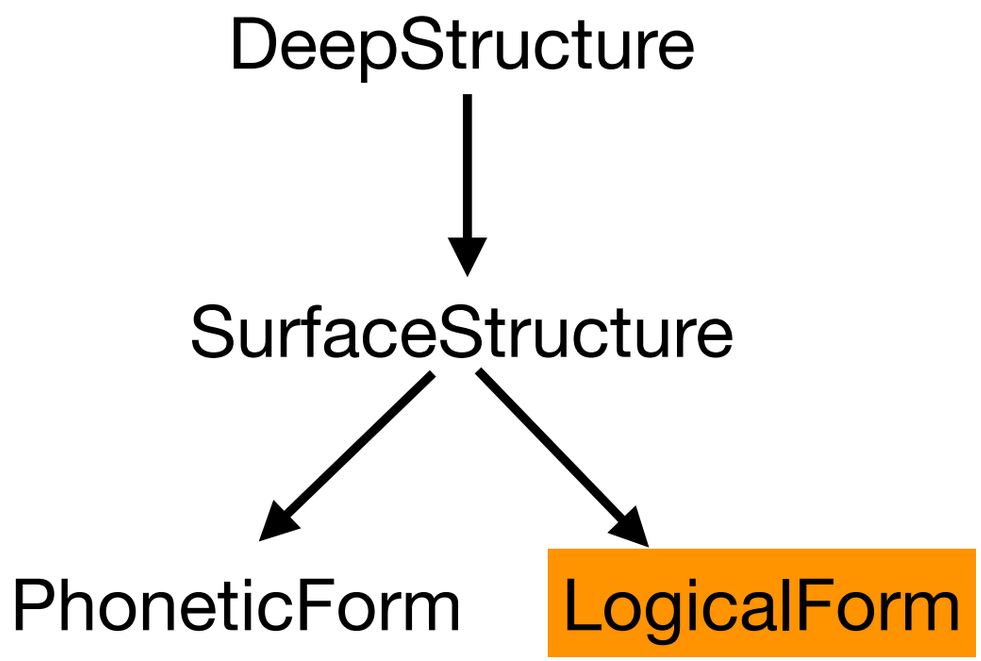
John Hale

MICHIGAN STATE UNIVERSITY

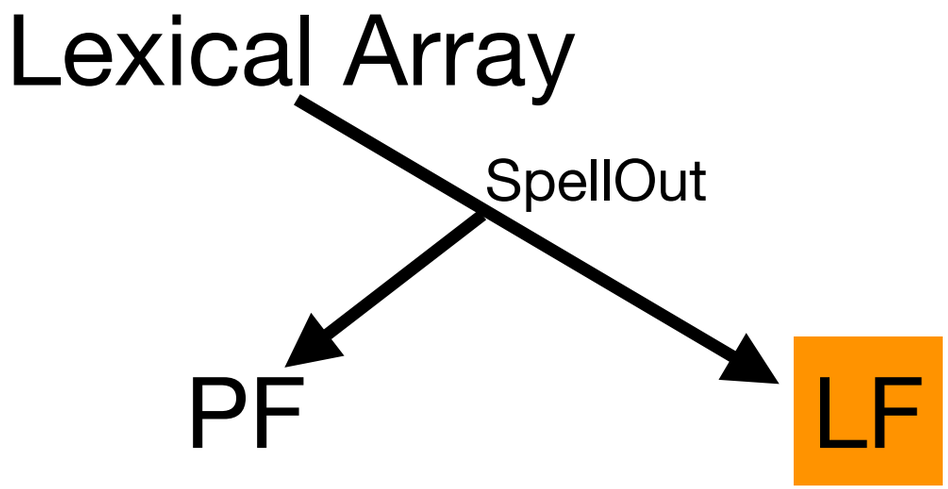
Mathematics of Language
July 29th 2007

LF in competence grammar

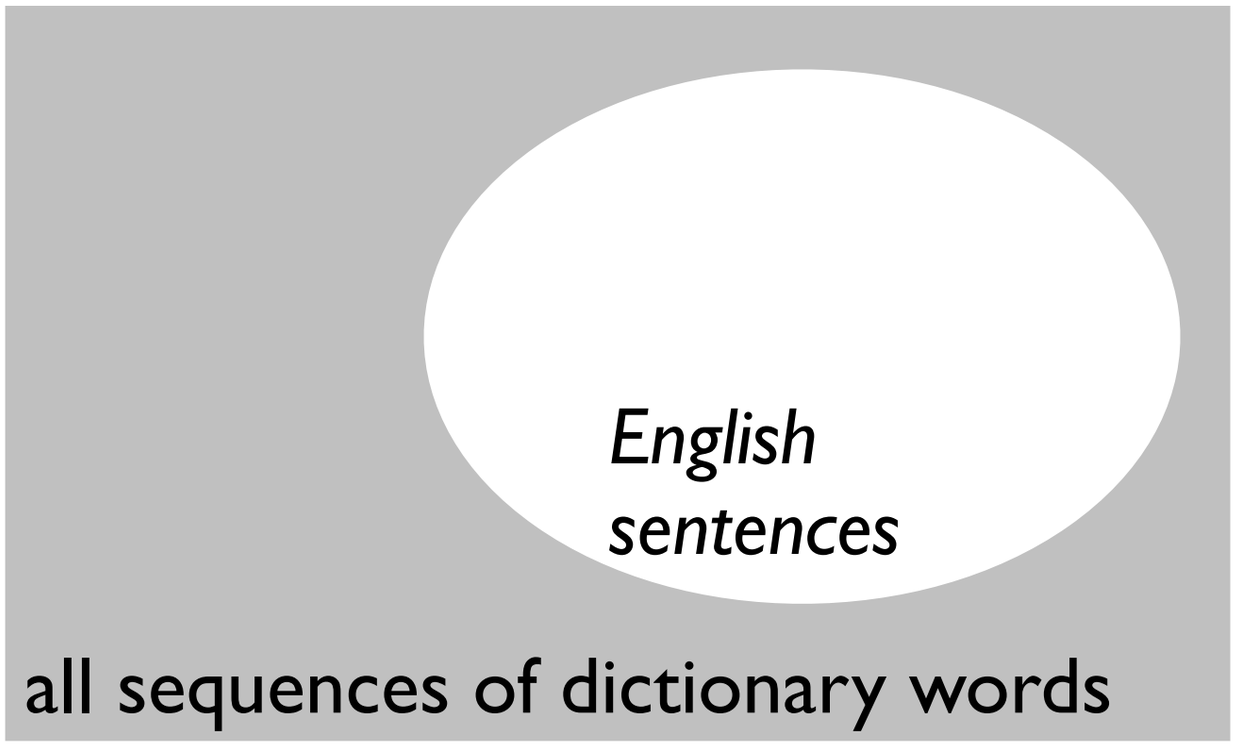
Government-Binding ca. 1981



Minimalism ca. 1994

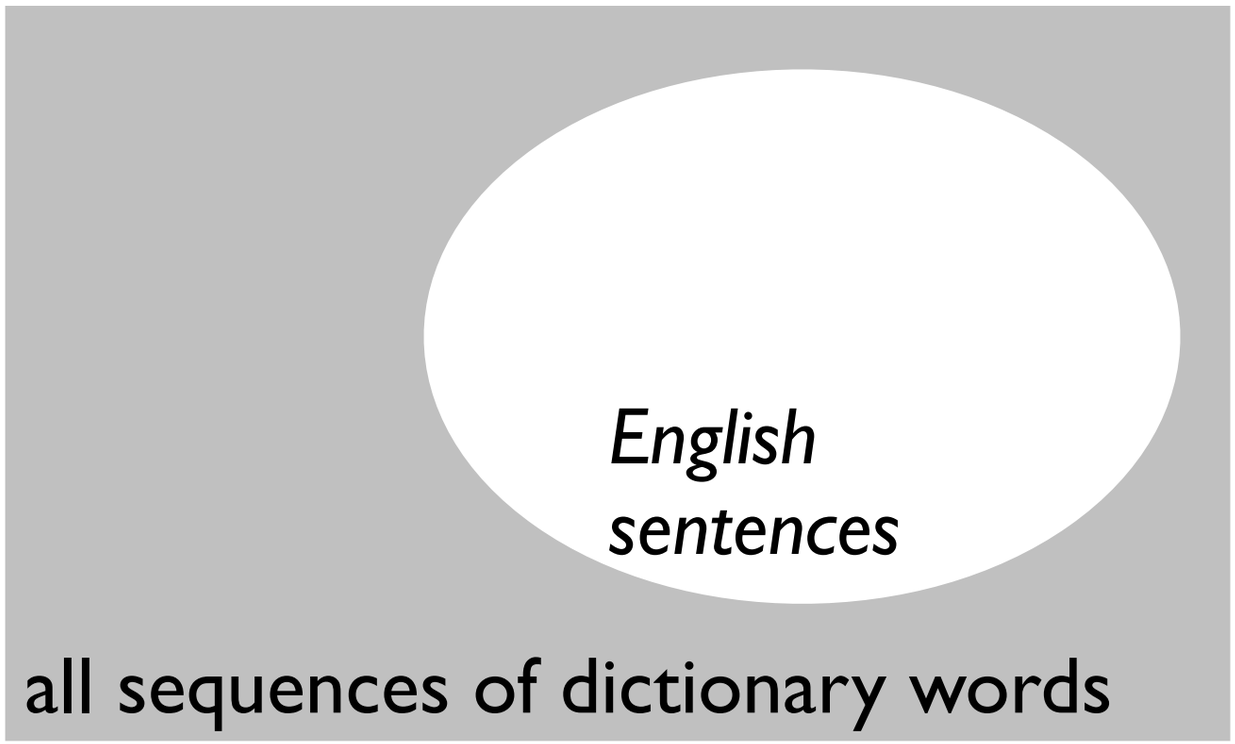


Competence grammar as computational-level theory



A sound/meaning pair (s,i) is an English sentence iff there exists a derivation....

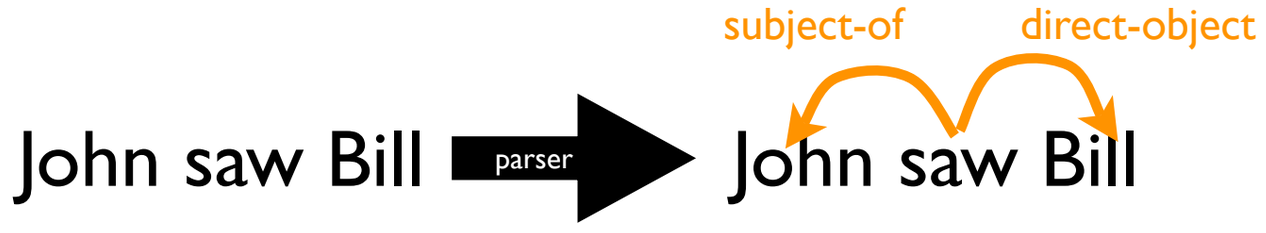
Competence grammar as computational-level theory



A sound/meaning pair (s,i) is an English sentence iff there exists a derivation....

Performance model as algorithmic-level theory

Given s, calculate i.



Claims in this talk

Yes

Competence Hypothesis

Reasonable performance models incorporate competence grammar

(Chomsky 65)

No

LF Causal Relevance Hypothesis

LF is causally implicated in language processing

(Berwick & Weinberg 84)

Form of argument

Yes **Competence Hypothesis**

- a combinator parser `s` whose program text directly mirrors the grammar
- a quantifier raising function `qr` that builds LFs, given surface structures as input
- a function `convert` that maps LFs to wffs in a logical language

No **LF Causal Relevance Hypothesis**

- a program computing the same function as `convert` ∘ `qr` can be derived that never builds LFs

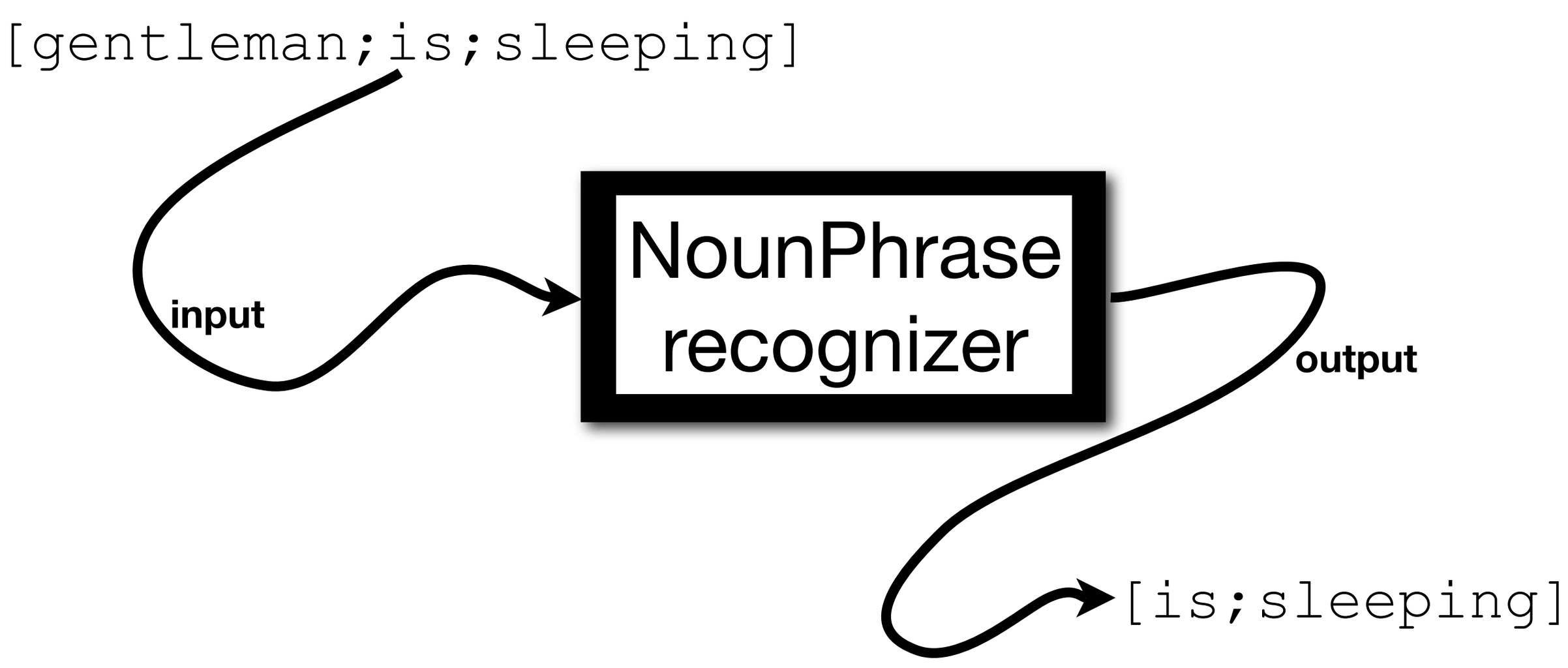
Plan of talk

1. Combinator parsers

2. Quantifier Raising

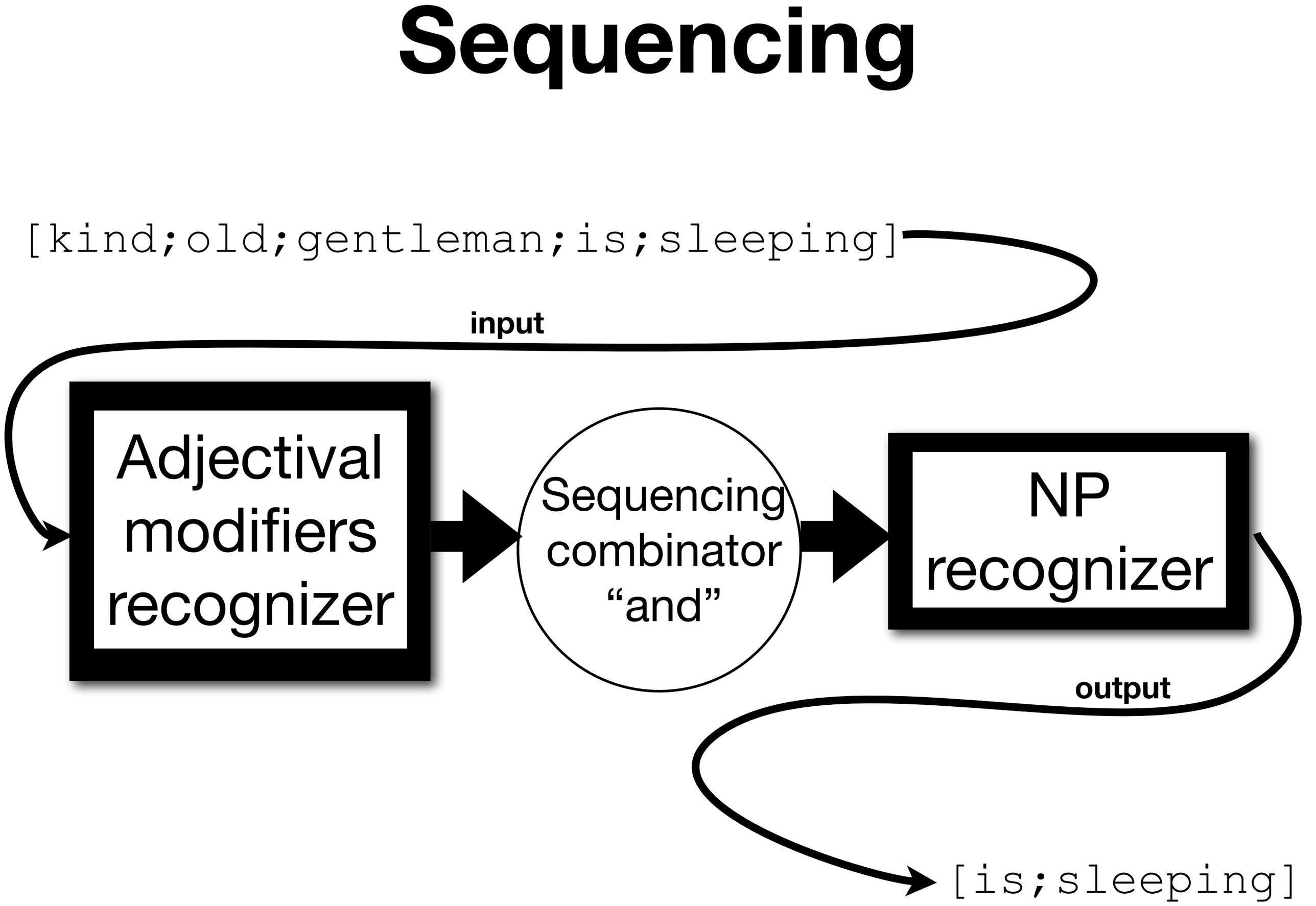
3. How to get rid of LF in a parser by using deforestation

Combinator recognizers



(Burge 75, Leermakers 93...)

Sequencing


[kind;old;gentleman;is;sleeping]
input
Adjectival modifiers recognizer
Sequencing combinator “and”
NP recognizer
output
[is;sleeping]

Alternatives

```
[kissing;is;fun]
[kissing;mathematical;linguists;is;fun]
```

Transitive VP recognizer

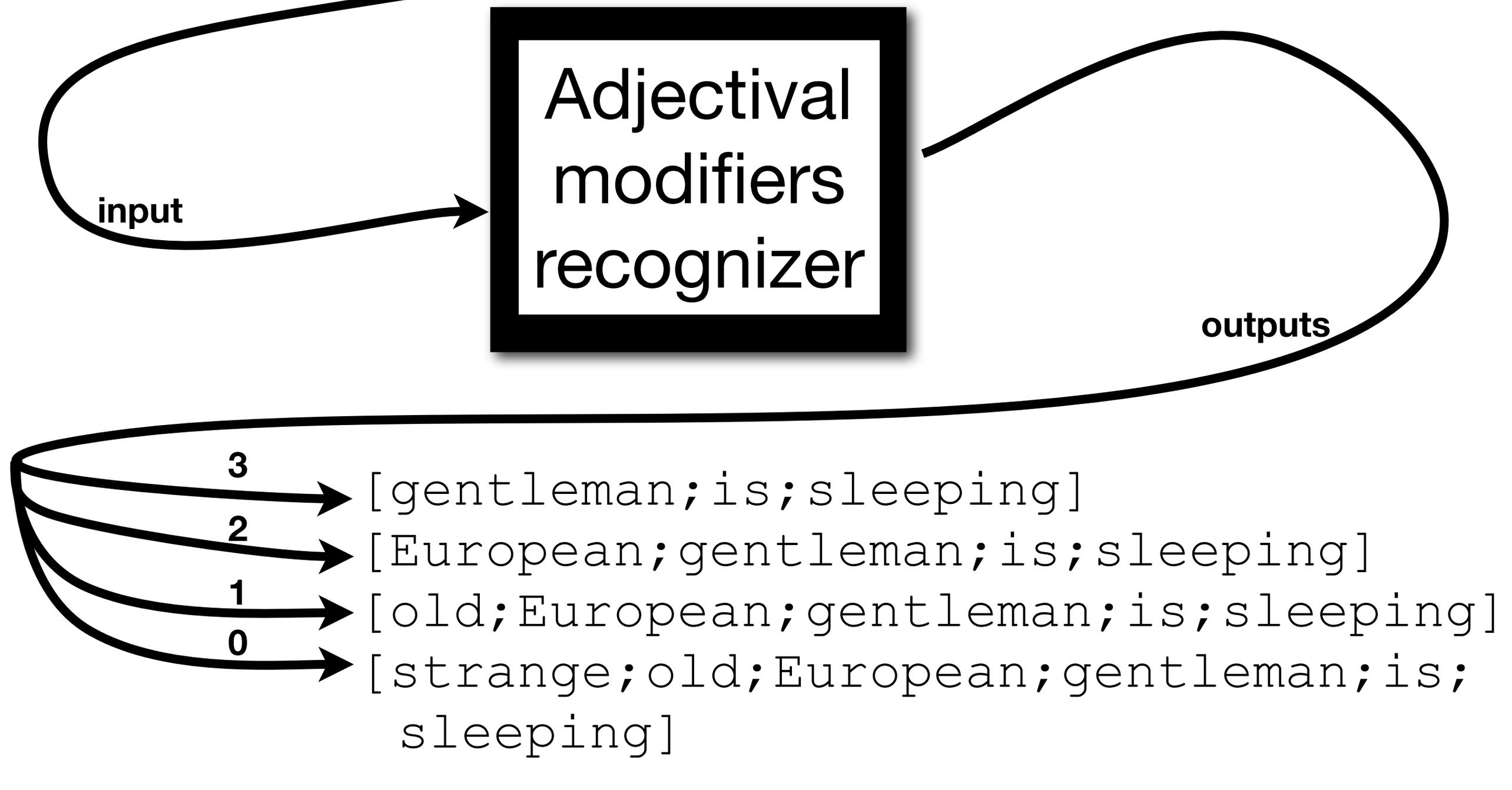
Alternation combinator “or”

Intransitive VP recognizer

```
[is;fun]
```

Nondeterminism

[strange;old;European;gentleman;is;sleeping]



Suffixes of Strings

$w =$

MoL is a great place to meet mathematical linguists

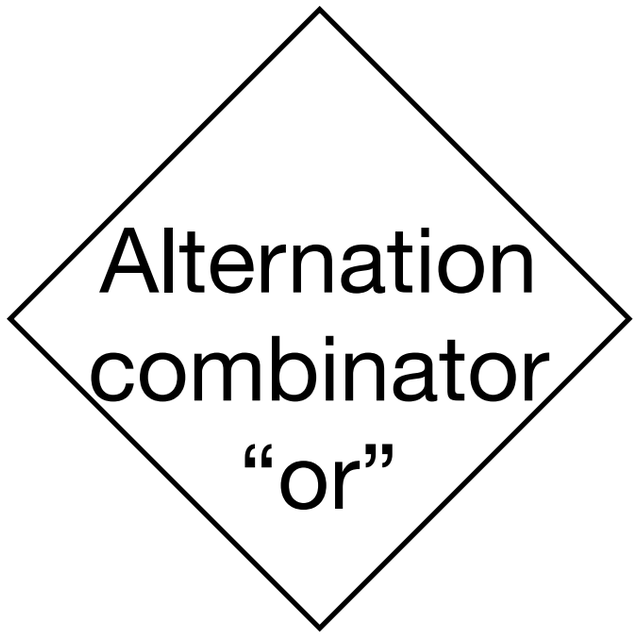
$\texttt{suffixes}(w) =$

{MoL is a great place to meet mathematical linguists,
is a great place to meet mathematical linguists,
a great place to meet mathematical linguists,
great place to meet mathematical linguists,
place to meet mathematical linguists,
to meet mathematical linguists,
meet mathematical linguists,
mathematical linguists,
linguists,
ϵ}

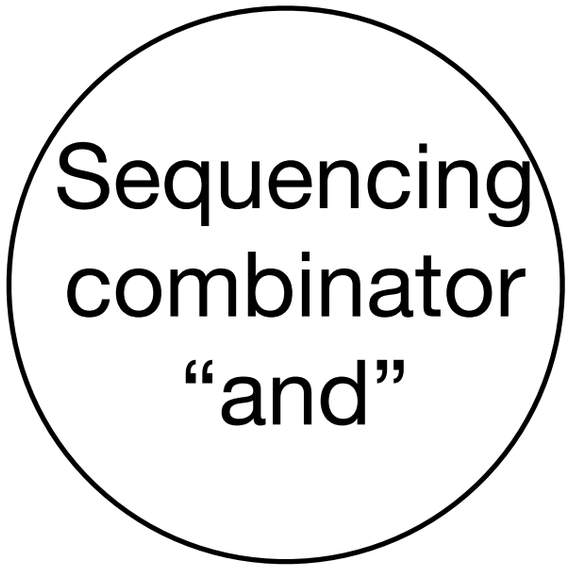
Caml datatype

```
(* encode string positions with suffixes *)
module Suffix =
  struct
    type t = string list
    let compare = compare
  end
module SuffixSet = Set.Make(Suffix)

(* can succeed many ways, encode nondeterminism with a set *)
type recognizer = SuffixSet.elt → SuffixSet.t
```

Combinators




```
let union = SuffixSet.union

let (alt : recognizer → recognizer → recognizer) =
  fun a b inp → union  (a inp) (b inp)
```




```
let map f s = SuffixSet.fold
        (fun x prev -> union (f x) prev)
        s
        SuffixSet.empty

let (seq : recognizer → recognizer → recognizer) =
  fun a b inp → map b (a inp)
```

Terminal recognizers

```
let gentleman = function inp →match inp with
      []  →SuffixSet.empty   (* inp has no head *)
|  t :: ts  →if t="gentleman"          (* inp has a head... *)
      then SuffixSet.singleton  ts     (* which we seek *)
      else SuffixSet.empty              (* not the one we want *)
```

Grammar

IP	→	DP I1
DP	→	D1
NP	→	N1
VP	→	V1
CP	→	C1
I1	→	I0 VP
D1	→	D0 NP
N1	→	N0 CP
N1	→	N0
V1	→	V0
C1	→	C0 IP
I0	→	will
D0	→	the
N0	→	idea
V0	→	suffice
C0	→	that

ambiguity

Recognizer

```
let rec ip words = (dp &. i1)
      words
and dp words = d1
      words
and np words = n1
      words
and vp words = v1
      words
and cp words = c1
      words
and i1 words = (i0 &. vp)
      words
and d1 words = (d0 &. np)
      words
and n1 words = ((n0 &. cp) |. n0)
      words
and v1 words = v0
      words
and c1 words = (c0 &. ip)
      words
and i0 = terminal "will"
and d0 = terminal "the"
and n0 = terminal "idea"
and v0 = terminal "suffice"
and c0 = terminal "that"
```

Sequencing combinator "and"

Alternation combinator "or"

From recognizers to analyzers

```
(* an outcome is a result and a remaining symbol list *)
type ('symbol,'result) outcome = 'result * ('symbol list)

(* an analyzer is a function from remaining symbols to a sequence of outcomes *)
type ('symbol,'result) analyzer = 'symbol list →('symbol, 'result) outcome Seq.t
```

From recognizers to analyzers

```
(* an outcome is a result and a remaining symbol list *)
type ('symbol,'result) outcome = 'result * ('symbol list)

(* an analyzer is a function from remaining symbols to a sequence of outcomes *)
type ('symbol,'result) analyzer = 'symbol list →('symbol, 'result) outcome Seq.t
```

pair up the remainder with some arbitrary type of result

From recognizers to analyzers

```
(* an outcome is a result and a remaining symbol list *)
type ('symbol,'result) outcome = 'result * ('symbol list)

(* an analyzer is a function from remaining symbols to a sequence of outcomes *)
type ('symbol,'result) analyzer = 'symbol list → ('symbol, 'result) outcome Seq.t
```

- pair up the remainder with some arbitrary type of result

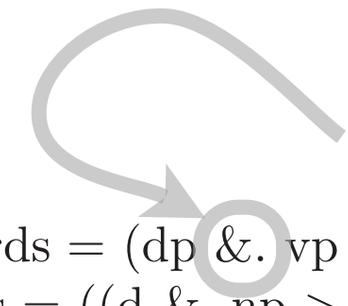
- handle nondeterminism with a collection of outcomes

A combinator parser

```
let rec s words = (dp &. vp >. branch2 "s") words
and dp words = ((d &. np >. branch2 "dp") |. (pn >. branch1 "dp")) words
and d = term ["every";"some"] >. branch1 "d"
and pn = term ["cecil";"dexter";"john"] >. branch1 "pn"
and np words = (((kstar adj) &. nbar) >. function (premodifiers,n1) →Node("np",premodifiers@[n1])) words
and nbar words = (adjoin2 pp "nbar" n) words
and n = term ["city";"body";"scale";"man";"woman"] >. branch1 "n"
and adj = term ["italian"] >. branch1 "adj"
and vp words = (((v &. dp) >. branch2 "vp") |. ((v &. dp &. pp) >. branch3 "vp")) words
and pp words = (((p &. dp) >. branch2 "pp")) words
and p = term ["in"] >. branch1 "p"
and v = term ["met";"saw";"played";"loves"] >. branch1 "v"
```

```
val s :
(string, string tree) analyzer = <fun>
```

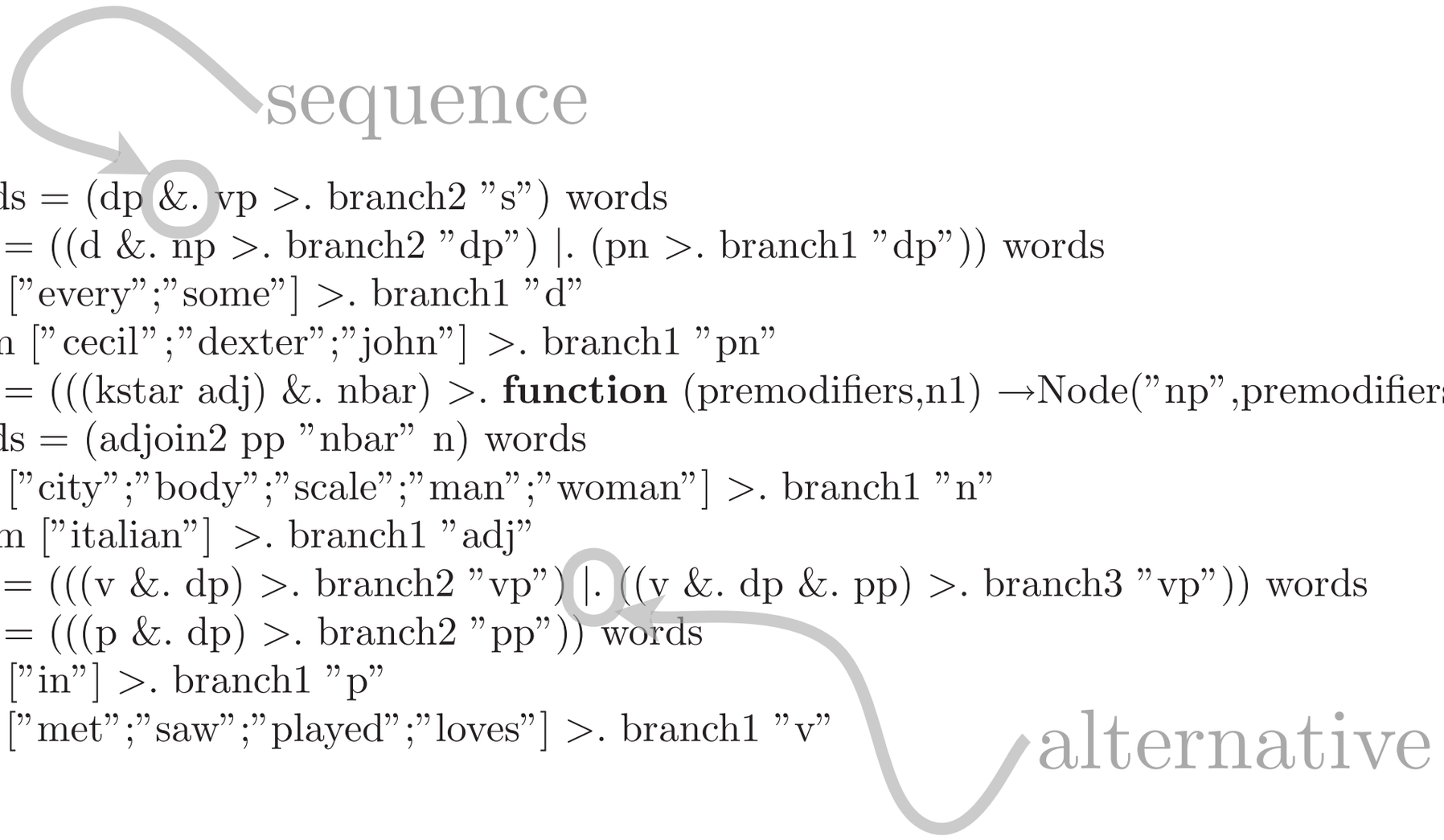
A combinator parser




```
let rec s words = (dp &. vp >. branch2 "s") words
and dp words = ((d &. np >. branch2 "dp") |. (pn >. branch1 "dp")) words
and d = term ["every";"some"] >. branch1 "d"
and pn = term ["cecil";"dexter";"john"] >. branch1 "pn"
and np words = (((kstar adj) &. nbar) >. function (premodifiers,n1) →Node("np",premodifiers@[n1])) words
and nbar words = (adjoin2 pp "nbar" n) words
and n = term ["city";"body";"scale";"man";"woman"] >. branch1 "n"
and adj = term ["italian"] >. branch1 "adj"
and vp words = (((v &. dp) >. branch2 "vp") |. ((v &. dp &. pp) >. branch3 "vp")) words
and pp words = (((p &. dp) >. branch2 "pp")) words
and p = term ["in"] >. branch1 "p"
and v = term ["met";"saw";"played";"loves"] >. branch1 "v"
```

```
val s :
(string, string tree) analyzer = <fun>
```

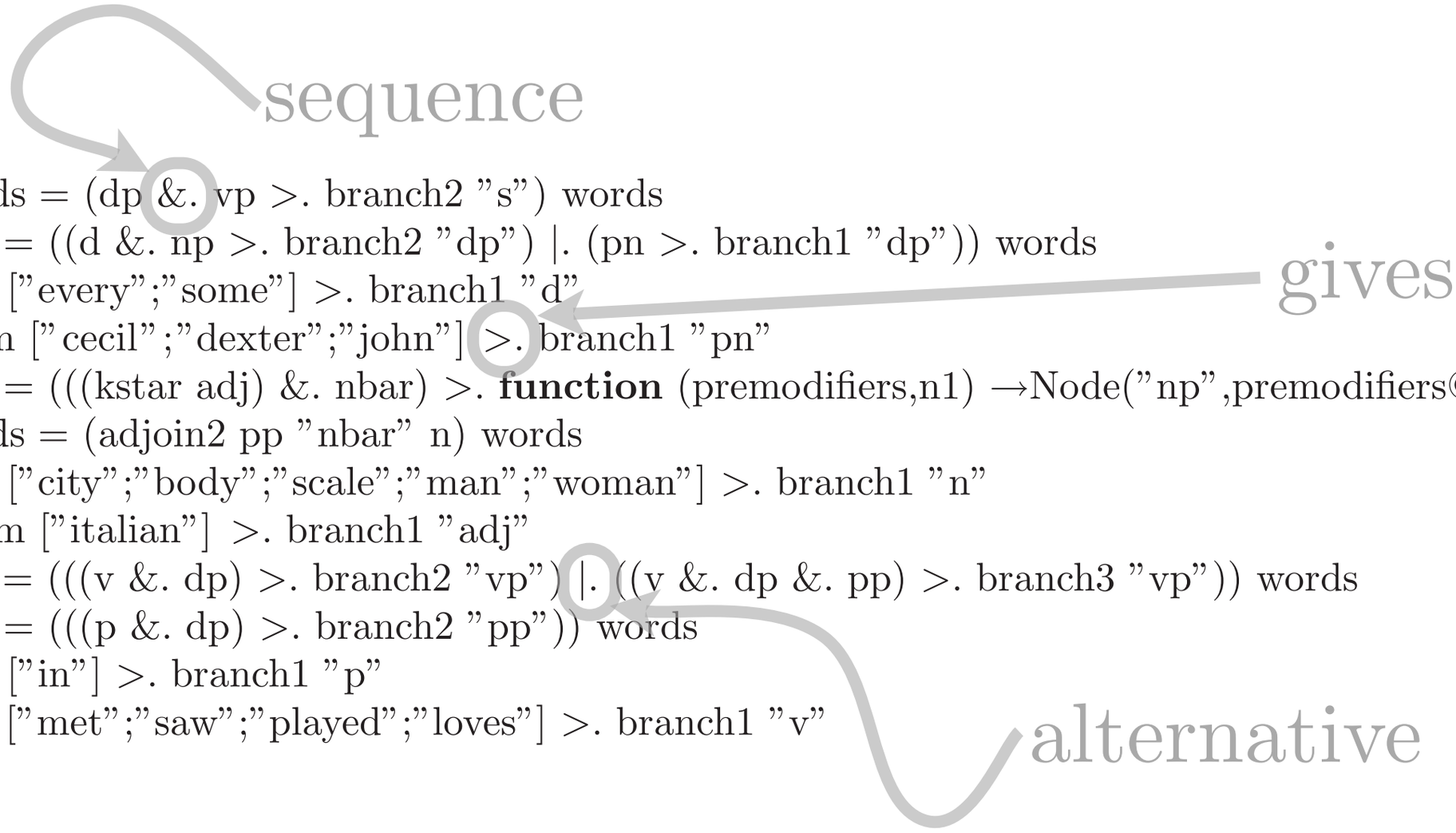
A combinator parser




```
let rec s words = (dp &. vp >. branch2 "s") words
and dp words = ((d &. np >. branch2 "dp") |. (pn >. branch1 "dp")) words
and d = term ["every";"some"] >. branch1 "d"
and pn = term ["cecil";"dexter";"john"] >. branch1 "pn"
and np words = (((kstar adj) &. nbar) >. function (premodifiers,n1) →Node("np",premodifiers@[n1])) words
and nbar words = (adjoin2 pp "nbar" n) words
and n = term ["city";"body";"scale";"man";"woman"] >. branch1 "n"
and adj = term ["italian"] >. branch1 "adj"
and vp words = (((v &. dp) >. branch2 "vp") |. ((v &. dp &. pp) >. branch3 "vp")) words
and pp words = (((p &. dp) >. branch2 "pp")) words
and p = term ["in"] >. branch1 "p"
and v = term ["met";"saw";"played";"loves"] >. branch1 "v"
```

```
val s :
(string, string tree) analyzer = <fun>
```

A combinator parser




```
let rec s words = (dp &. vp >. branch2 "s") words
and dp words = ((d &. np >. branch2 "dp") |. (pn >. branch1 "dp")) words
and d = term ["every";"some"] >. branch1 "d"
and pn = term ["cecil";"dexter";"john"] >. branch1 "pn"
and np words = (((kstar adj) &. nbar) >. function (premodifiers,n1) →Node("np",premodifiers@[n1])) words
and nbar words = (adjoin2 pp "nbar" n) words
and n = term ["city";"body";"scale";"man";"woman"] >. branch1 "n"
and adj = term ["italian"] >. branch1 "adj"
and vp words = (((v &. dp) >. branch2 "vp") |. ((v &. dp &. pp) >. branch3 "vp")) words
and pp words = (((p &. dp) >. branch2 "pp")) words
and p = term ["in"] >. branch1 "p"
and v = term ["met";"saw";"played";"loves"] >. branch1 "v"
```

```
val s :
(string, string tree) analyzer = <fun>
```

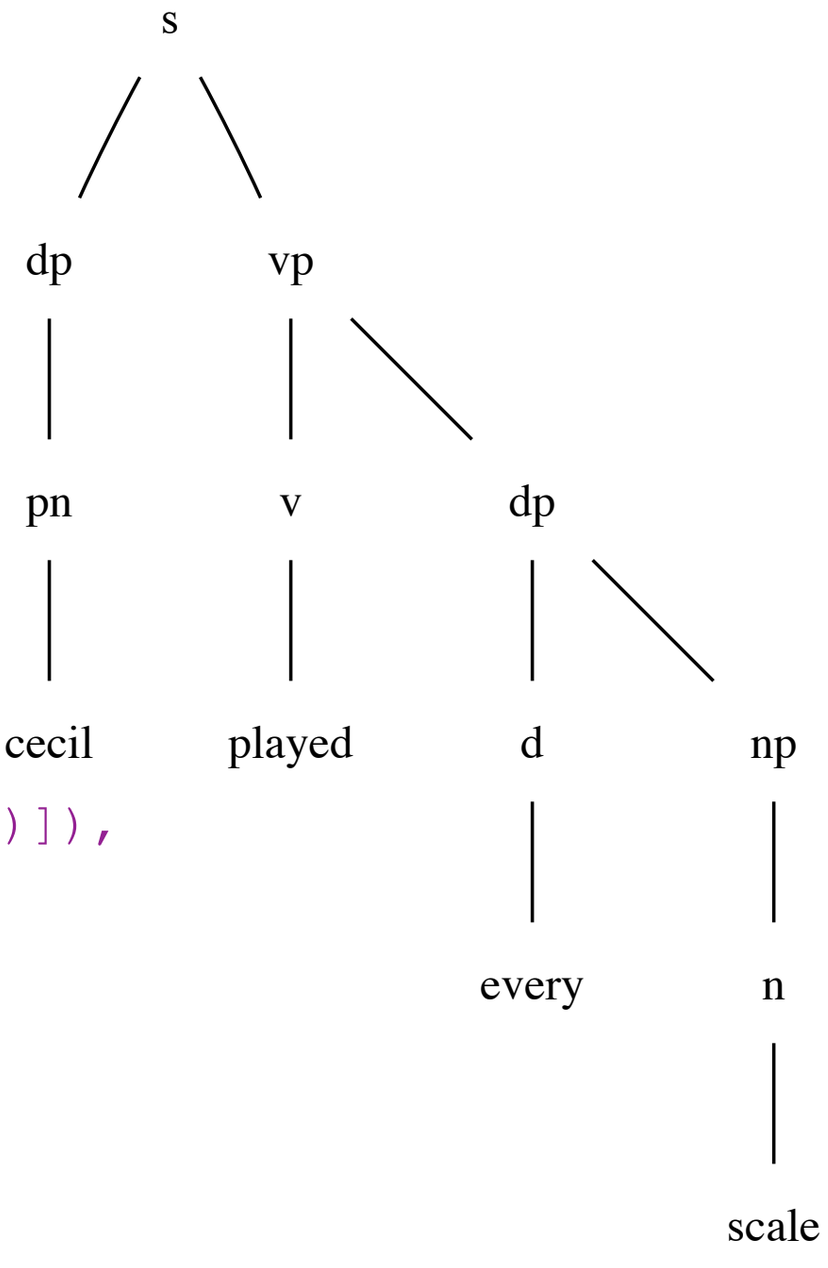
Outputs

start symbol=entry point

```
# Seq.hd (s ["cecil";"played";"every";"scale"]);;
- : (string, string tree) outcome =
(Node ("s",
  [Node ("dp", [Node ("pn", [Node ("cecil", [])])]);
   Node ("vp",
    [Node ("v", [Node ("played", [])]);
     Node ("dp",
      [Node ("d", [Node ("every", [])]);
       Node ("np", [Node ("n", [Node ("scale", [])])])])])]),
[])
```

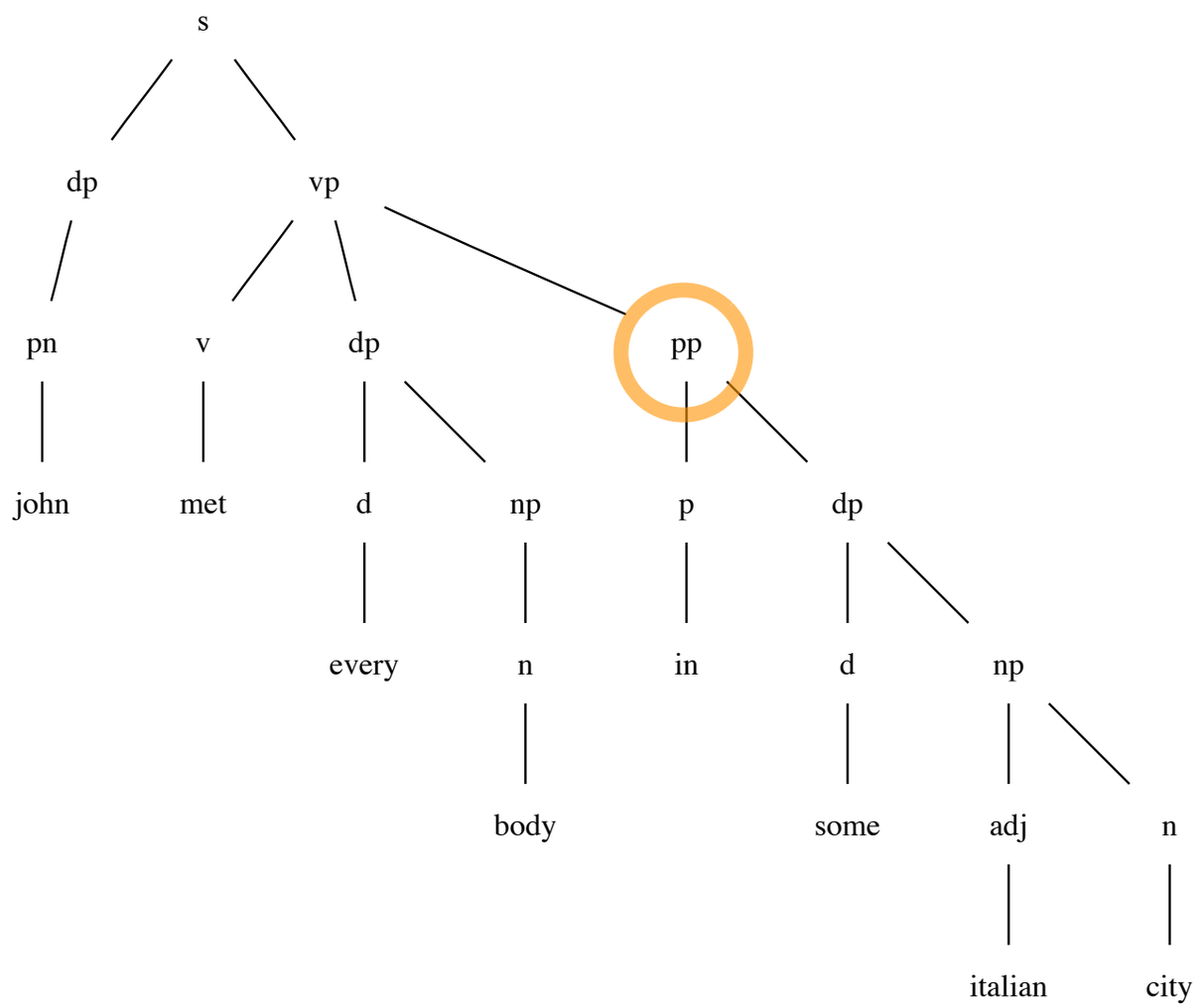
empty remainder

⇒ recognition succeeded

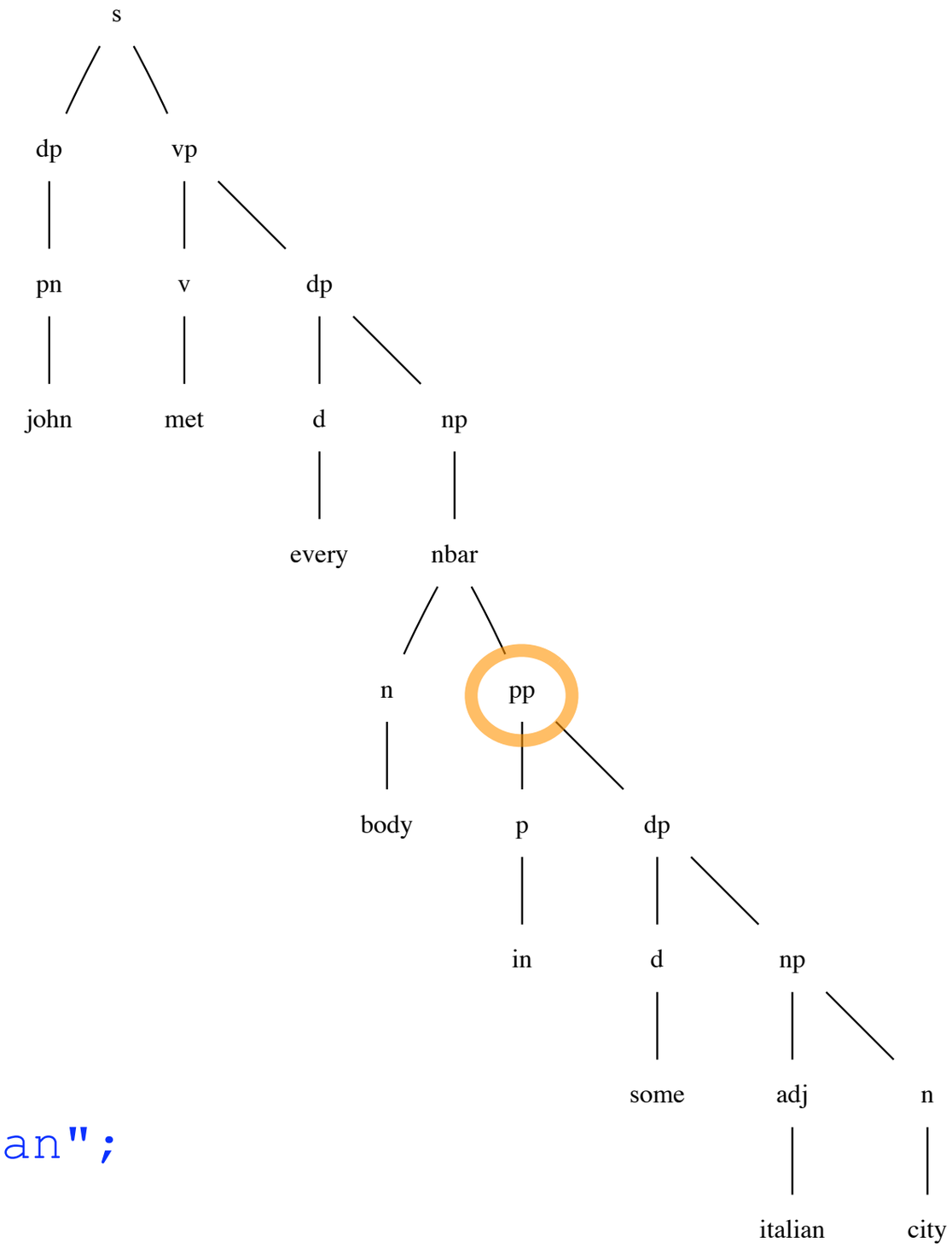


Syntactic ambiguity

High PP attachment



Low PP attachment




```
#Seq.count λx.true (s ["john"; "met";
"every"; "body"; "in"; "some"; "italian";
"city"]));;
- : int = 2
```

Plan of talk

1. Combinator parsers

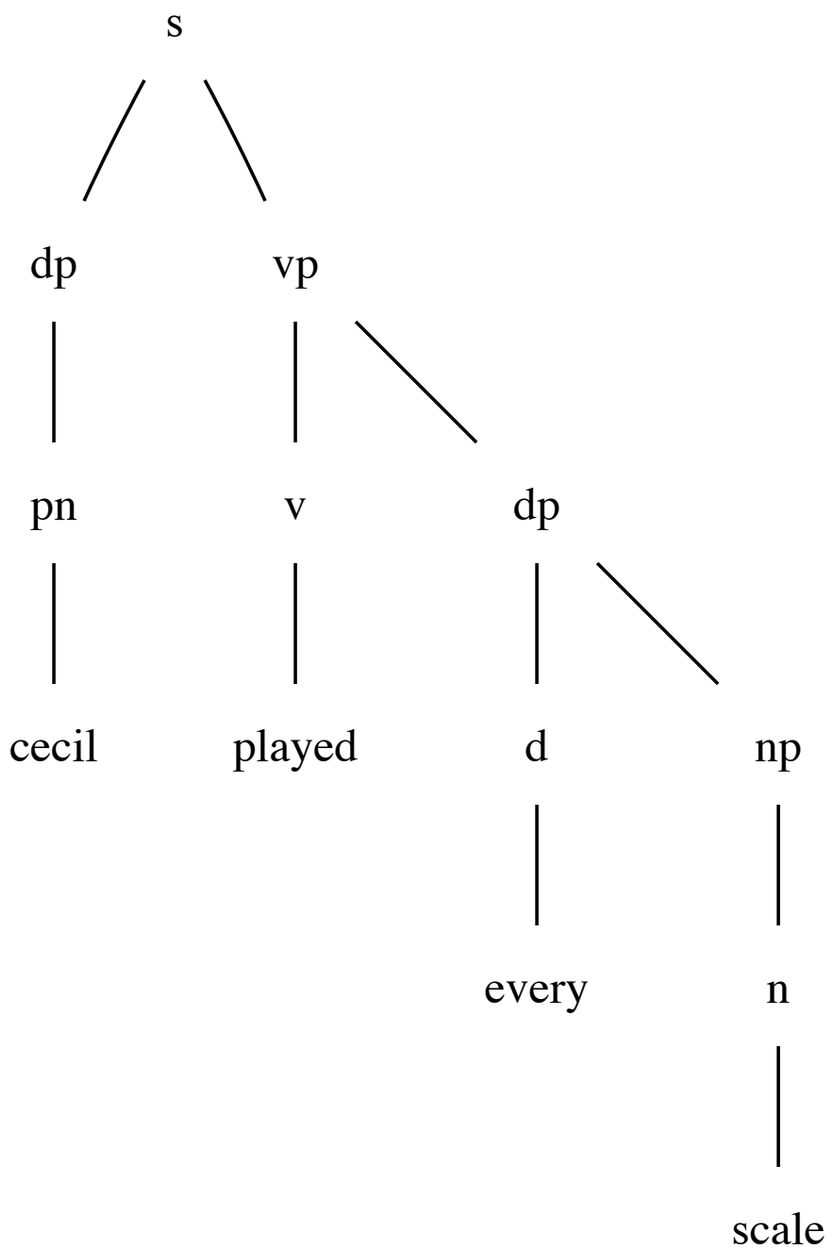
2. Quantifier Raising

3. How to get rid of LF in a parser by using deforestation

Quantifiers

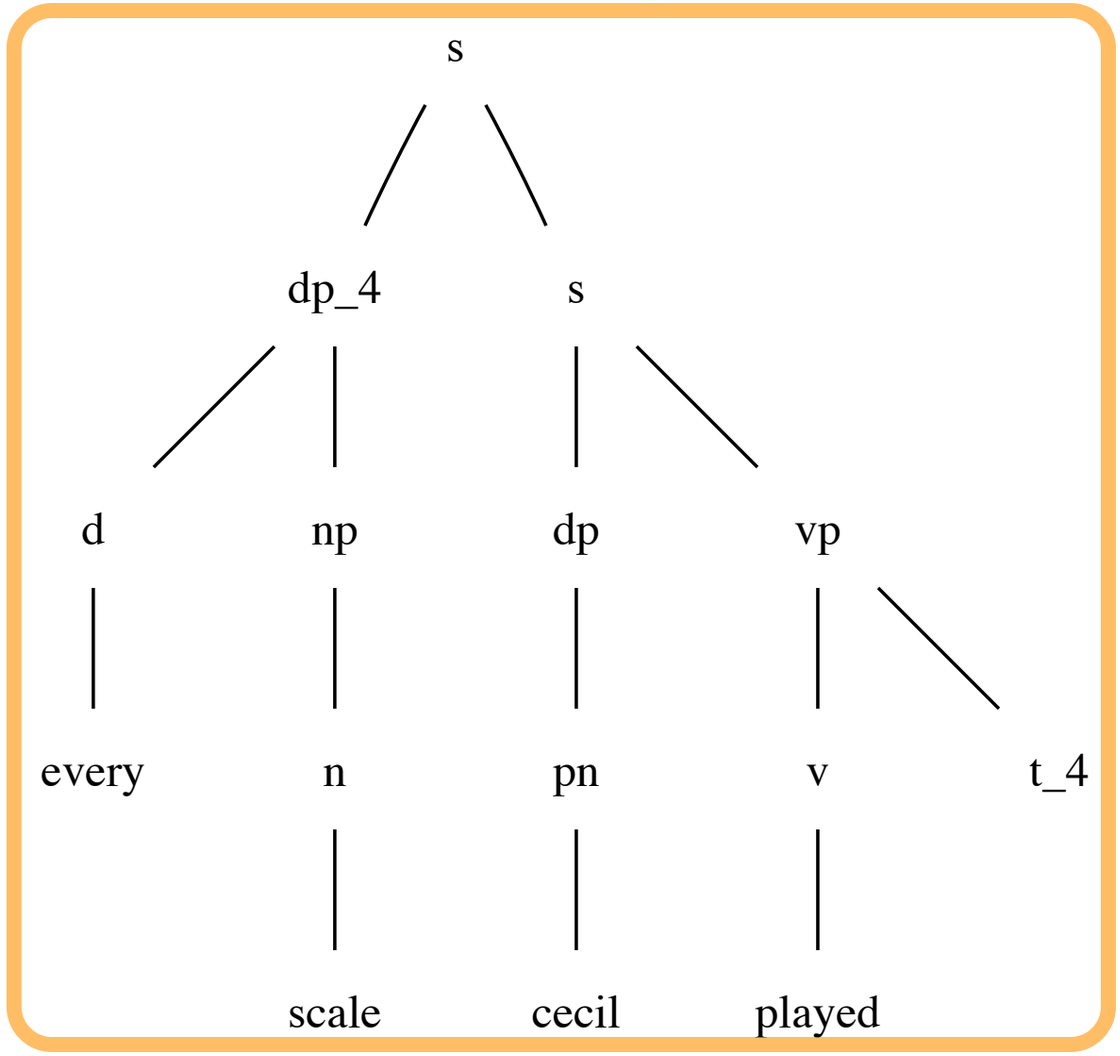
all, some, most, more than twenty,...

Quantifier Raising is a transformation

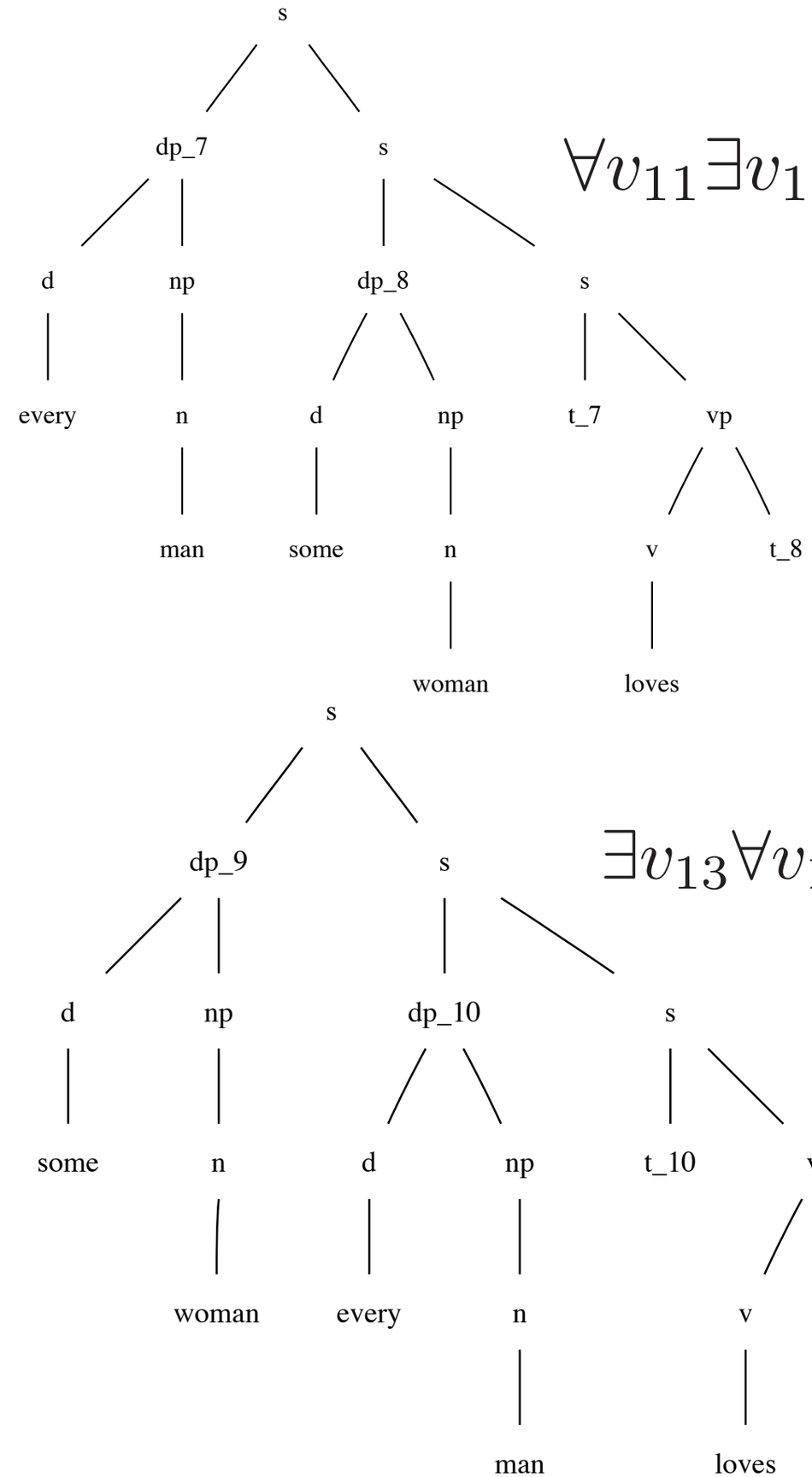


Adjoin

a Logical Form



Quantifier Scope Ambiguity

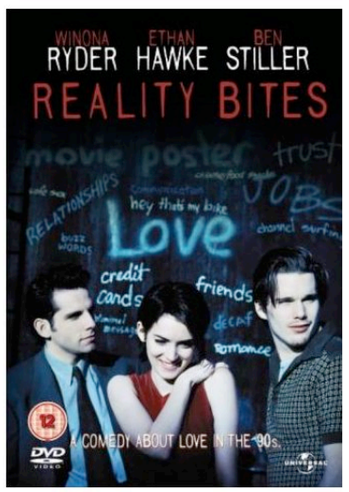


$$\forall v_{11} \exists v_{12} \,.\, \text{man}(v_{11}) \rightarrow \text{woman}(v_{12}) \wedge \text{loves}(v_{12}, v_{11})$$

each guy finds his own soulmate

$$\exists v_{13} \forall v_{14} \,.\, \text{woman}(v_{13}) \wedge [\text{man}(v_{14}) \rightarrow \text{loves}(v_{13}, v_{14})]$$

every guy loves Winona Ryder



QR is like Movement

✔[Which patients]$_i$ will a doctor make sure that we give t_i a tranquilizer?

✘[Which patients]$_j$ will a doctor examine the possibility that we give t_j a tranquilizer?

island constraint on Move α accounts for unacceptability

A doctor will examine the possibility that we give every new patient a tranquilizer.

✔ $\exists$ doctor d $\forall$ new-patient p

d will examine the possibility that we give p a tranquilizer

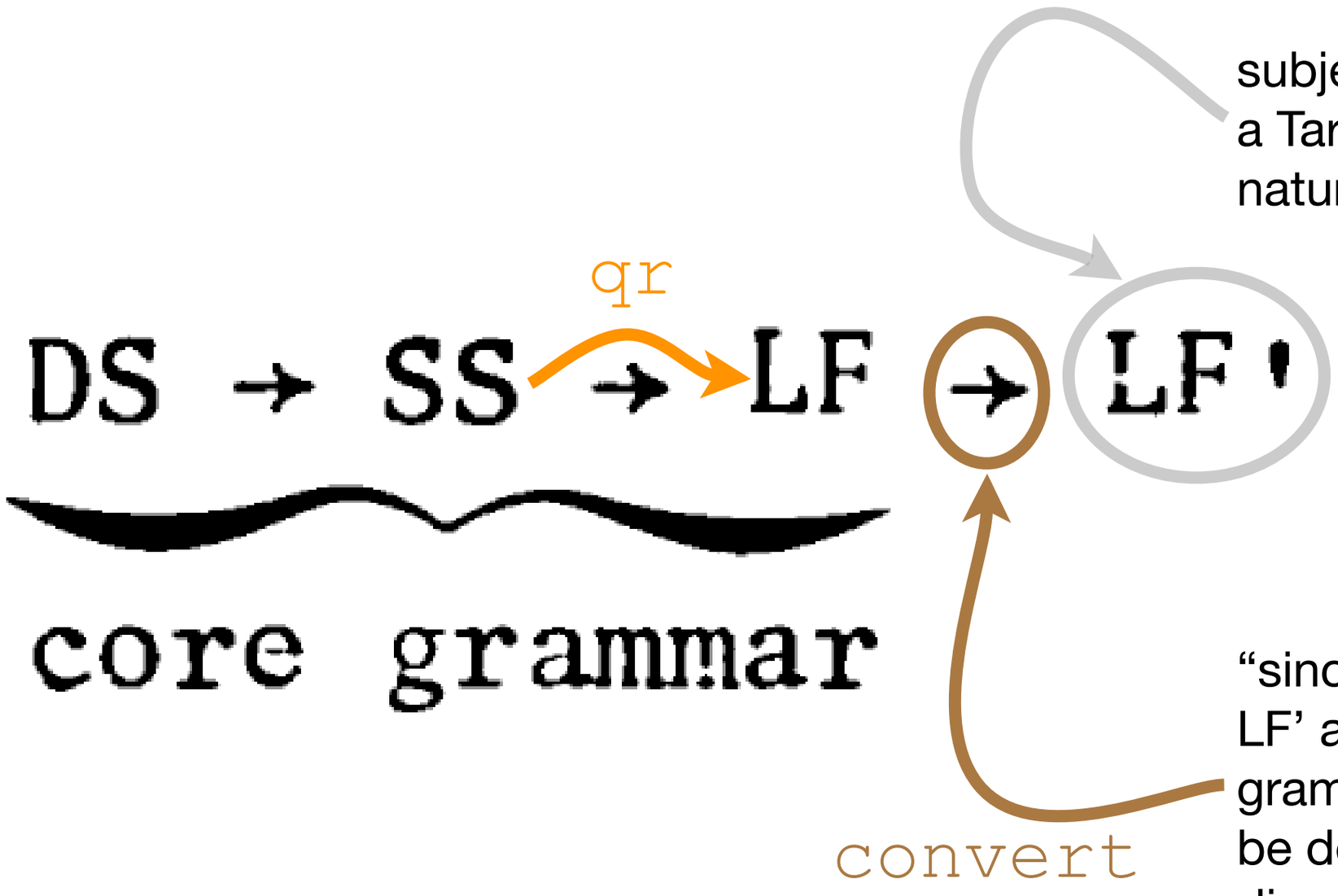
✘ $\forall$ new-patient p $\exists$ doctor d

d will examine the possibility that we give p a tranquilizer

island constraint on QR accounts for missing interpretation

Tanya Reinhart
Linguistics and Philosophy, 20:335-397, 1997

LF as an interface level



subject to the “recursive clauses of a Tarskian truth-condition theory for natural language quantification”

page 26

“since the rules mapping from LF to LF’ are not rules of sentence grammar, their functioning may also be determined by properties of discourse”

footnote 11 page 50

Robert May 1977
The Grammar of Quantification
MIT Dissertation

Natural parsing algorithm

1. parse input to obtain surface structures
2. apply `qr` to every quantified phrase
3. `convert` resultant LFs into LF' formulas

Natural parsing algorithm

1. parse input to obtain surface structures
2. apply `qr` to every quantified phrase
3. `convert` resultant LFs into LF' formulas

$$\texttt{convert} \circ \texttt{qr} \approx$$

```
foreach (ss,QDPplace) {
  tempLF ←qr (ss,QDPplace)
  return convert (tempLF)
}
```

Natural parsing algorithm

1. parse input to obtain surface structures
2. apply `qr` to every quantified phrase
3. `convert` resultant LFs into LF' formulas

`convert` ∘ `qr` ≈

```
foreach (ss,QDPplace) {
  tempLF ←qr (ss,QDPplace)
  return convert (tempLF)
}
```

intermediate data

Plan of talk

1. Combinator parsers
2. Quantifier Raising
3. How to get rid of LF in a parser by using deforestation

Deforestation

source-to-source program transformation

technique for getting rid of intermediate data

$$sum\,(map\ square\,(upto\,1\,n))$$

upto creates the list [1,2,...,n]

map square creates the list [1,4,...,n^2]

sum finally yields $\frac{1}{6}\left(2n^3+3n^2+n\right)$

Philip Wadler
Theoretical Computer Science 73:231-248, 1990

Motivation: efficiency

$$sum\,(map\;square\,(upto\,1\,n))$$

deforestation

$$h\,0\,1\,n$$

where

$$\begin{aligned} h\;a\,m\,n &= \mathbf{if}\,m > n \\ &\quad \mathbf{then}\,a \\ &\quad \mathbf{else}\,h\;(a + square\,m)\;(m+1)\;n \end{aligned}$$

no list allocations!

Wadler 90:

It is clear that each of the rules preserves equivalence

(1) $T[\![v]\!] = v$

(2) $T[\![c\ t_1\ \ldots\ t_k]\!] = c\ (T[\![t_1]\!])\ \ldots\ (T[\![t_k]\!])$

(3) $T[\![f\ t_1\ \ldots\ t_k]\!] = T[\![t[t_1/v_1, \ldots, t_k/v_k]]\!]$
where f is defined by $f\ v_1\ \ldots\ v_k = t$

(4) $T[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= \textbf{case}\ v\ \textbf{of}\ p'_1 : T[\![t'_1]\!] \mid \cdots \mid p'_n : T[\![t'_n]\!]$

(5) $T[\![\textbf{case}\ c\ t_1\ \ldots\ t_k\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= T[\![t'_i[t_1/v_1, \ldots, t_k/v_k]]\!]$
where $p'_i = c\ v_1\ \ldots\ v_k$

(6) $T[\![\textbf{case}\ f\ t_1\ \ldots\ t_k\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= T[\![\textbf{case}\ t[t_1/v_1, \ldots, t_k/v_k]\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
where f is defined by $f\ v_1\ \ldots\ v_k = t$

(7) $T[\![\textbf{case}\ (\textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_m : t_m)\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!]$
$= T[\![\textbf{case}\ t_0\ \textbf{of}$
$p_1 : (\textbf{case}\ t_1\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n)$
$\cdots$
$p_m : (\textbf{case}\ t_m\ \textbf{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n)]\!]$

replace functions with their definitions

push matching on constructors inside branches

Figure 4: Transformation rules for the Deforestation Algorithm

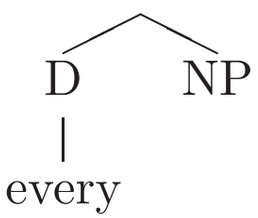
cf. Burstall & Darlington 77

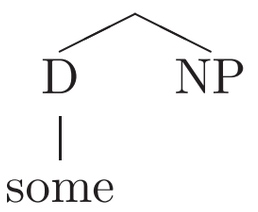
LF gone

no adjoined phrase markers created

$$\textbf{let}\ \ \mathrm{f}\ =\ \lambda\ (\mathrm{ss}\,,\mathrm{places})\ \rightarrow \mathrm{convert}\ (\mathrm{qr}\ (\mathrm{ss}\,,\mathrm{places}))$$

sequence of Wadler-steps

function f(*ss*,QDPplaces)
  if no more QDPplaces **then**
    `predicateify`(*ss*)
  else
    examine the next QDPplace p
    if $p =$ 

    then
      let restrictor = `predicateify`(NP)
      let v be a fresh variable name
      let body = ss with p replaced by an indexed trace
      $\forall\, v$ restrictor(v) $\rightarrow$ f (body, remaining QDPplaces)
    end if
    if $p =$ 

    then
      let restrictor = `predicateify`(NP)
      let v be a fresh variable name
      let body = ss with p replaced by an indexed trace
      $\exists\, v$ restrictor(v) $\wedge$ f (body, remaining QDPplaces)
    end if
  end if
end function

Plan of talk

1. Combinator parsers

2. Quantifier Raising

3. How to get rid of LF in a parser by using deforestation

Conclusions

Yes **Competence Hypothesis**

No **LF Causal Relevance Hypothesis**

- grammar with LF ⊢ directly compositional parser

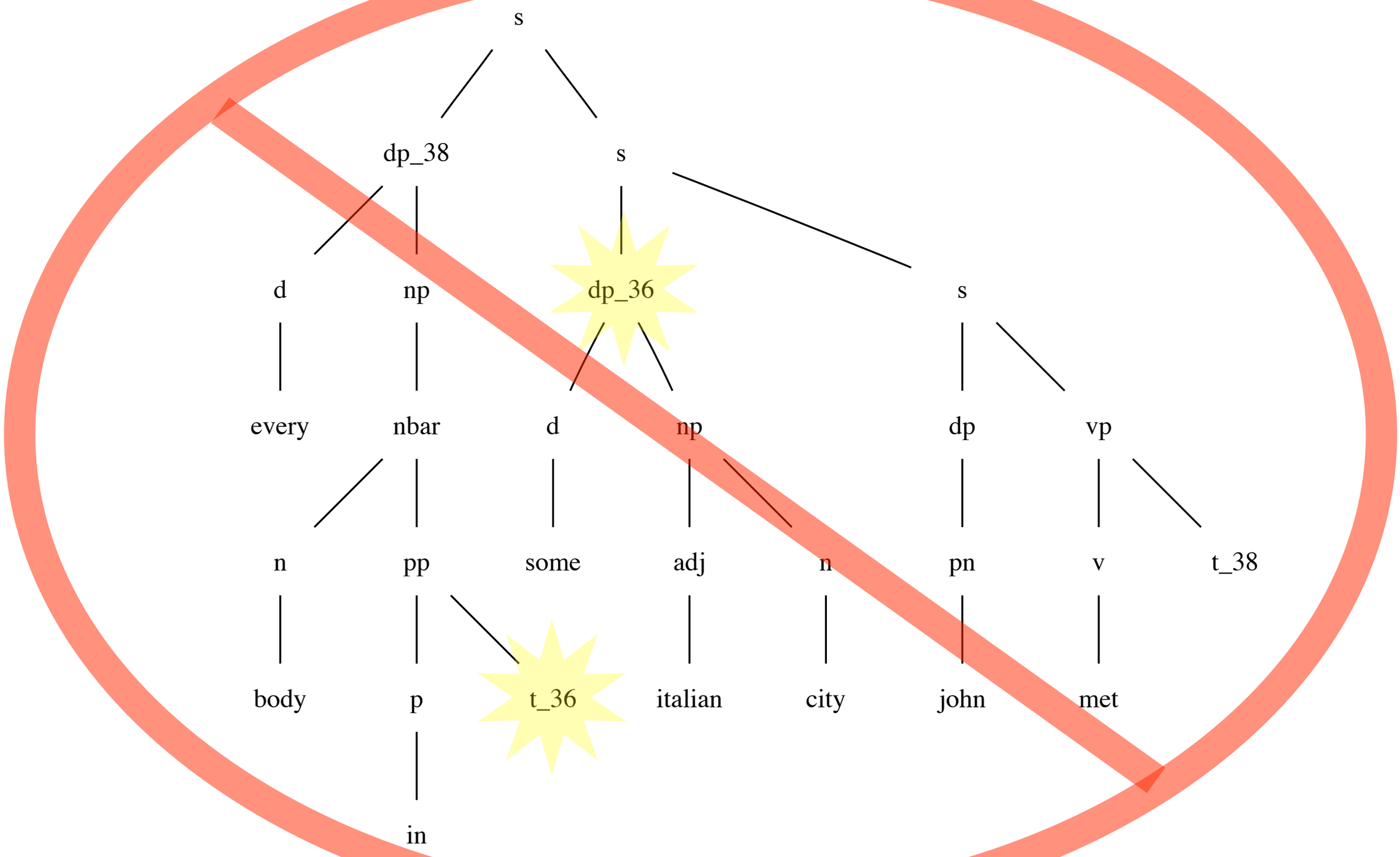
- psycholinguistic evidence for LF might becoming from SS operations

Bonus slides

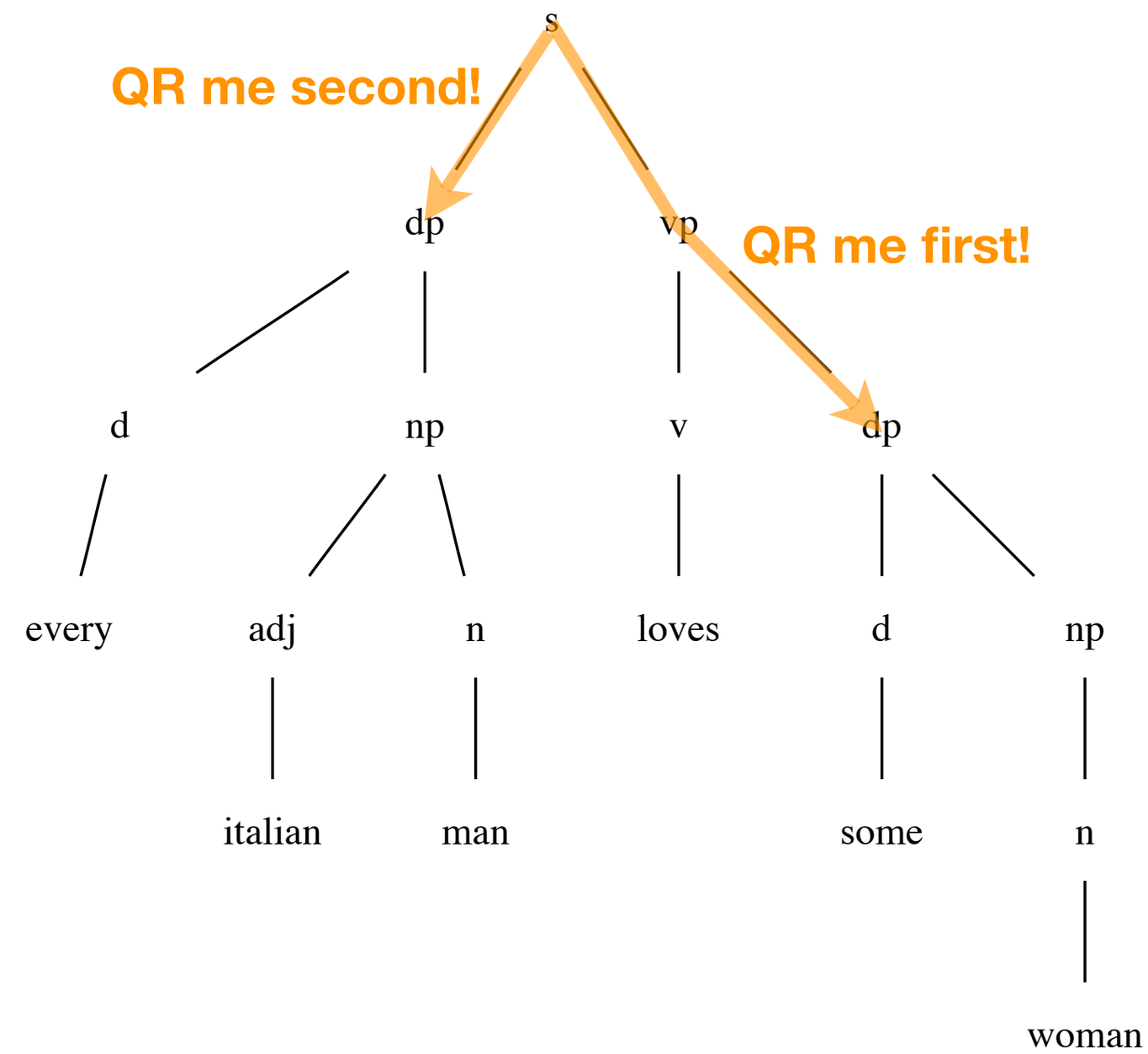
Constraints on LF

Proper Binding

Moved constituents must c-command their traces



Avoid generate/test



Enforce ProperBinding early, on
(tree,list of Gorn addresses of quantified phrases) pairs

Parser combinators

```
(* either p1 can parse the input, or p2 can *)
let (alternative : ('a,'b) analyzer →('a,'b) analyzer →('a,'b) analyzer) = fun p1 p2 inp
  →Seq.append ((p1 inp),(p2 inp))

(* use p2 to parse all the remainders p1 leaves behind. pair-up the results *)
let (sequence : ('a,'b) analyzer →('a,'c) analyzer →('a,('b * 'c)) analyzer) = fun p1 p2 inp →
          Seq.multiply (function (result1,remainder1) →
                        Seq.map
                        (function (result2,remainder2) →((result1, result2 ),remainder2))
                        (p2 remainder1)
                       ) (* outermost function takes individual outcomes to Seq.t of outcomes *)
                (p1 inp)

(* map the function f : 'b→'c over the result of parsing inp with p *)
let (gives : ('a,'b) analyzer →('b →'c) →('a,'c) analyzer)= fun p f inp →
  Seq.map (function (result,remainder) →(f result,remainder)) (p inp)
```

How to run it

```
#let result = ip
["the";"idea";"will";"suffice"];;
val result : SuffixSet.t = <abstr>

#SuffixSet.elements result;;
- : SuffixSet.elt list = [[]]
```

SuffixSet includes the empty remainder

⇒ recognition succeeded

⇒ "the idea will suffice" ∈ L(G)

How it says No

```
#let result2 = ip ["the";"half-baked intuition";
"will";"suffice"];;
val result2 : SuffixSet.t = <abstr>

#SuffixSet.empty = result2;;
- : bool = true
```

SuffixSet is empty

⇒ recognition failed (out-of-vocabulary)

⇒ "the half-baked intuition will suffice" ∉ L(G)