CORRECTIONS
(Revised 2009)

for

Natural Language Processing for Prolog Programmers

by Michael A. Covington

Englewood Cliffs, NJ: Prentice Hall, 1994

This document contains a list of corrections to the book, plus a set of replacements for certain pages, mainly those that have errors in diagrams.

The corrected pages can be printed out and attached inside your book in front of the pages that they replace. Make sure your PDF printing software is set to print actual size rather than enlarging to fit the paper.

This is a partial set of corrections and may be extended in the future.

Downloads for this book are now located at:
http://www.ai.uga.edu/ftplib/natural-language-book

Michael A. Covington
www.ai.uga.edu/mc
Revised February 10, 2009

Known errors in the first printing of

Natural Language Processing for Prolog Programmers
Michael A. Covington
Englewood Cliffs, NJ: Prentice-Hall, 1993 (1994)
ISBN 0-13-629213-5

Thanks to Mark Plaksin, Sven Hartrumpf, Benjamin Yuen,
Sebastian Varges, and others for pointing these out.

---

p. 7 "every verb has hundreds of forms" should read
"every verb has over 100 forms".

p. 21 The printed text is correct, but the file 'dosengl.pl'
distributed with the book had some parentheses
missing in the definition of make_string/2, which
should read as follows:

```
make_string([H|T],Result) :-
   name(H,Hstring),
   make_string(T,Tstring),
   append(Hstring,Tstring,Result).

make_string([],[]).
```

p. 32 last paragraph: "four clauses" should be "five clauses".

p. 42 line 5 from bottom: "looks as" should be "looks at".

p. 44 caption to fig. 3.4: delete "or both".
(Actually, a grammar rule could change both things, but
there is no example of it here.)

p. 47 para. 3: the "wrong" rule should actually be:

```
s --> np, { write(L2) }, vp.    % WRONG!
```

p. 149 [[agr:[person:2]]] should be [agr:[person:2]]

p. 165 line 4, after "find them all." add footnote:

> Sebastian Varges points out that the parser will also
> loop on a set of rules of the form:
>
> ```
> S -> X Y
> Y -> X
> X -> 0                  (0 is null set symbol)
> ```
>
> Although finite, this grammar has a troubling ambiguity
> of a kind that apparently does not occur in natural
> language: an empty constituent can occur either as an X,
> or as a Y immediately following an X.

p. 170 ex. 6.5.2.2: "first clause of chart" should be

"second clause of parse" (with 'parse'
in typewriter type).

p. 206 ex. 7.3.2.1: "both in logical notation and in Prolog"
should be "in logical notation; the Prolog versions are
already in restrictor-scope format".

p. 271 In the program listing, the fifth clause of split_suffix
should be:

```
% y changes to i after consonant, before suffix beg. w vowel
split_suffix([C,i,X|Rest],[C,y],Suffix) :-
   \+ vowel(C), \+ (X = i), vowel(X), suffix([X|Rest],Suffix).
```

p. 273 In the program listing, the fifth clause of split_suffix
should be:

```
% y changes to i after consonant, before suffix beg. w vowel
split_suffix([C,i,X|Rest],[C,y],Suffix,Cat) :-
   \+ vowel(C), \+ (X = i), vowel(X), suffix([X|Rest],Suffix,Cat).
```

p. 274 “karakho...” (in 4 places) should be “karahko...”

p. 279 s. 9.4.5. para. 2. after "match the input" add foonote:

> As the cuts in Fig. 9.7 suggest, this implementation ignores some subtleties. Instead of cuts, a more sophisticated system would use a fifth argument to distinguish 'others' arcs, which apply to input that does not match the rules, from situations in which a rule is violated. See Ritchie et al. 1992:21-26.

p. 289 para. 2: "Quintus Prolog, and Arity Prolog"
should be "and Quintus Prolog".

p. 301 line 6: delete "with either".

p. 304 At bottom, delete the sentence
"These predicates...Chapter 3."

p. 312 Footnote 4 is erroneous and should be ignored.
(Note the corrected 'testmisc.pl' available herewith.)

Appx. A This appendix was originally going to be Chapter 2,
and in a few places it says that something "will be"
covered in a subsequent chapter. Change to "was"
wherever applicable.

p. 319 line 10, and files 'readatom.pl', 'readchar.pl', 'readat2.pl':
"char_type/1" should be "char_type/3".

p. 321 2nd line of listing, and in file 'namenum.pl':
"complete_atomics" should be "complete_line/3".

p. 323 line 11: same correction as p. 319 line 10.

$$S \rightarrow NP\ VP$$
$$VP \rightarrow V\ (NP)$$

This will account for structures such as those in Figure 4.4.

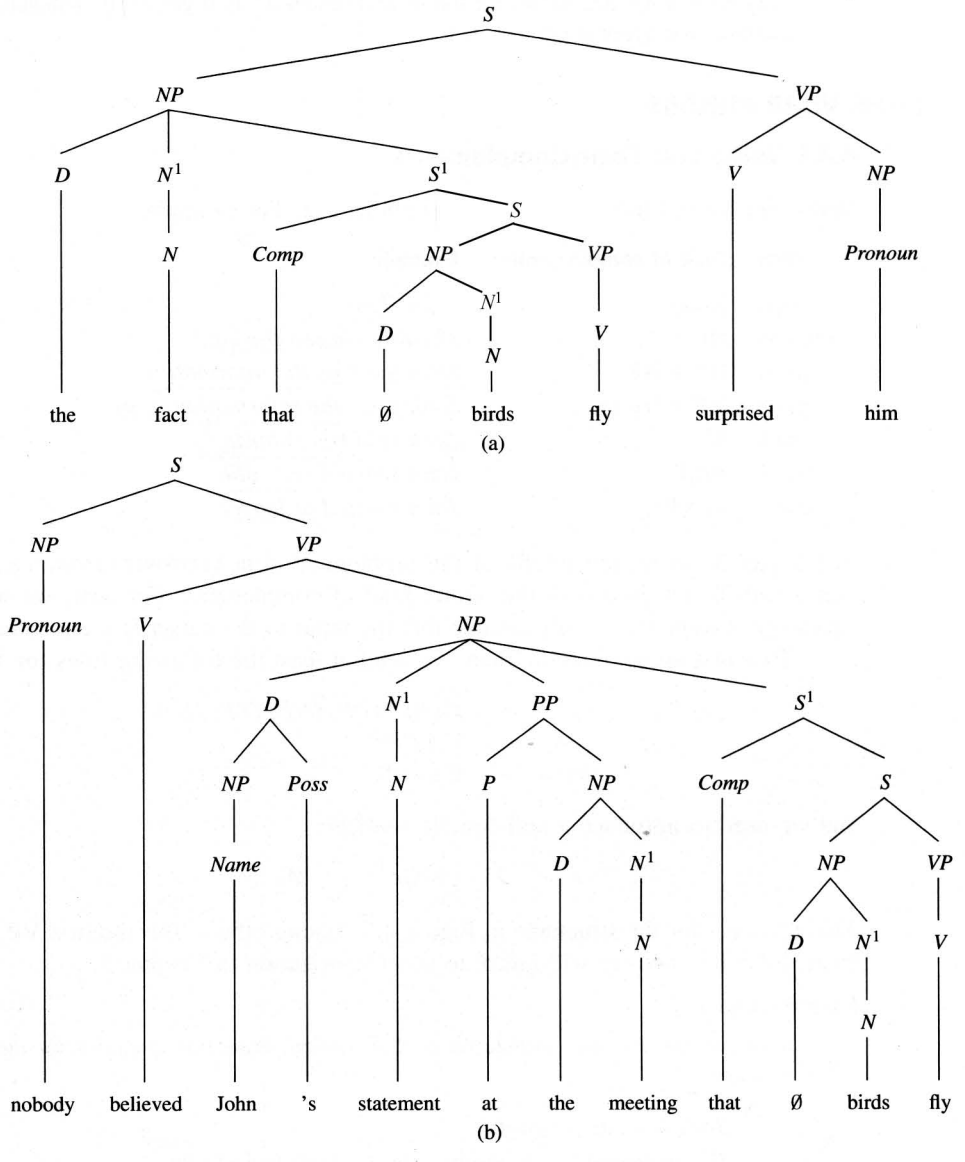
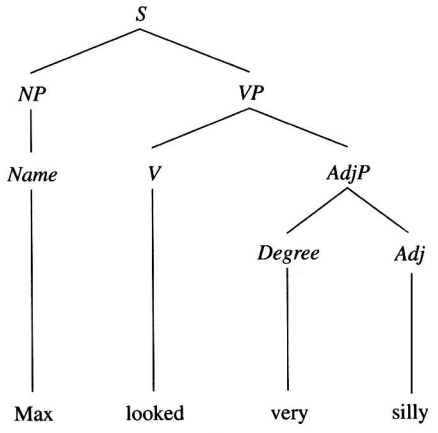


Figure 4.4 Sentences that contain complex noun phrases.



(f)

Figure 4.5 cont.

Exercise 4.4.1.2

Extend your parser to parse the sentences in Figure 4.5 and the sentences in the previous exercise. You need not include all expansions of the VP rule, as long as you include enough to parse the sentences here, and you are prepared to add more as needed later on.

4.4.2 Particles

A PARTICLE is a preposition without an object. Particles occur only with specific verbs which require them, such as *look up* or *throw out.* The same verbs also occur without the particles, with somewhat different meanings.

When present, the particle occurs in either of two positions, as shown in Figure 4.6 on page 94. Note that *Joe looked up the tower* is ambiguous (he either looked up the tower in a book, or looked upward along the tower), and that this ambiguity is structural; Figure 4.7 on pages 94 and 95 shows the two structures.

Exercise 4.4.2.1

Extend your parser to handle particles and to parse the sentences in Figures 4.6 and 4.7, giving both structures for *Joe looked up the tower.* You need not provide for all the combinations of particles with other parts of the VP; just add enough VP rules to parse the sentences needed for this exercise.

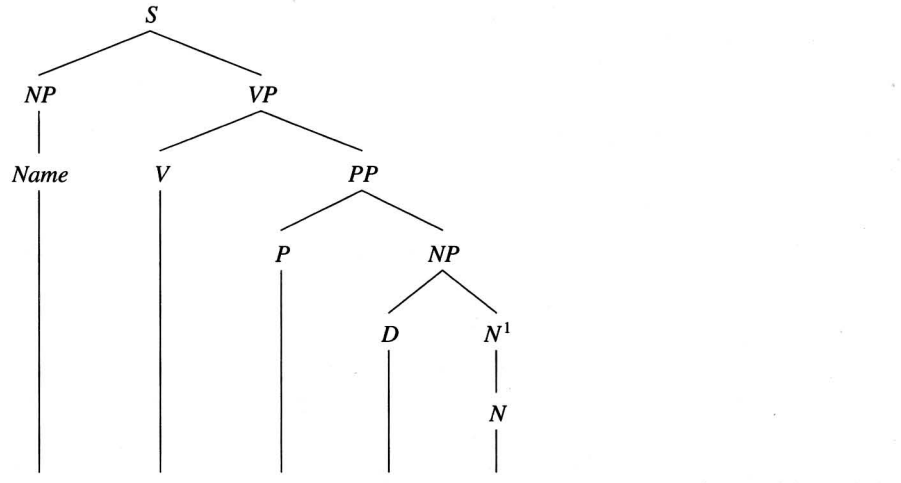
4.4.3 The Copula

The COPULA, or verb of being (*is, are,* etc.), takes an NP or AdjP as complement, as shown in Figure 4.8 on page 95.

Note (added 2008): In Fig. 4.8 we treat Copula as a separate syntactic category. There is a good case for treating the copula as a V with a particular subcategorization.

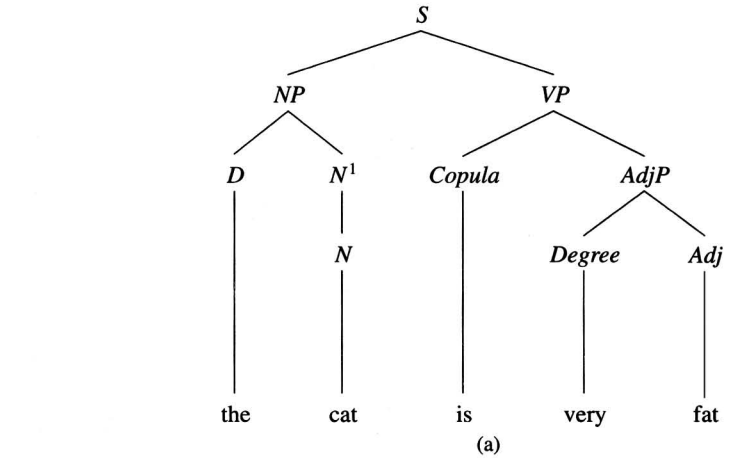
Exercise 4.4.3.1

Extend your parser to parse the sentences in Figure 4.8.

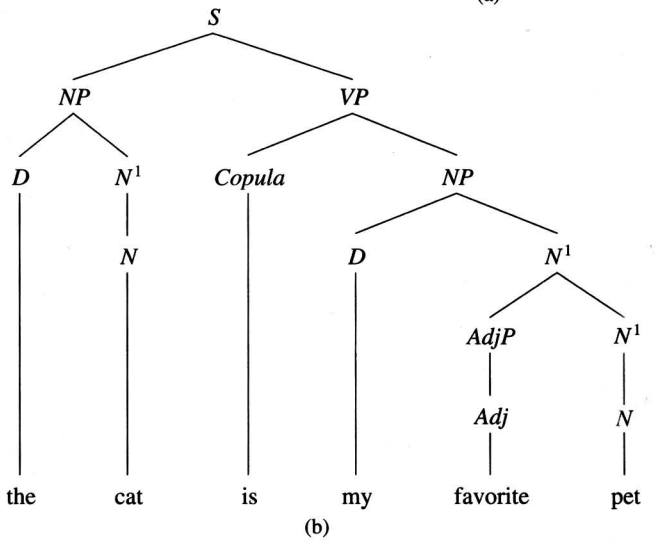


(b)

Figure 4.7 cont.



(a)



(b)

Figure 4.8 The copula takes an NP or AdjP as complement.

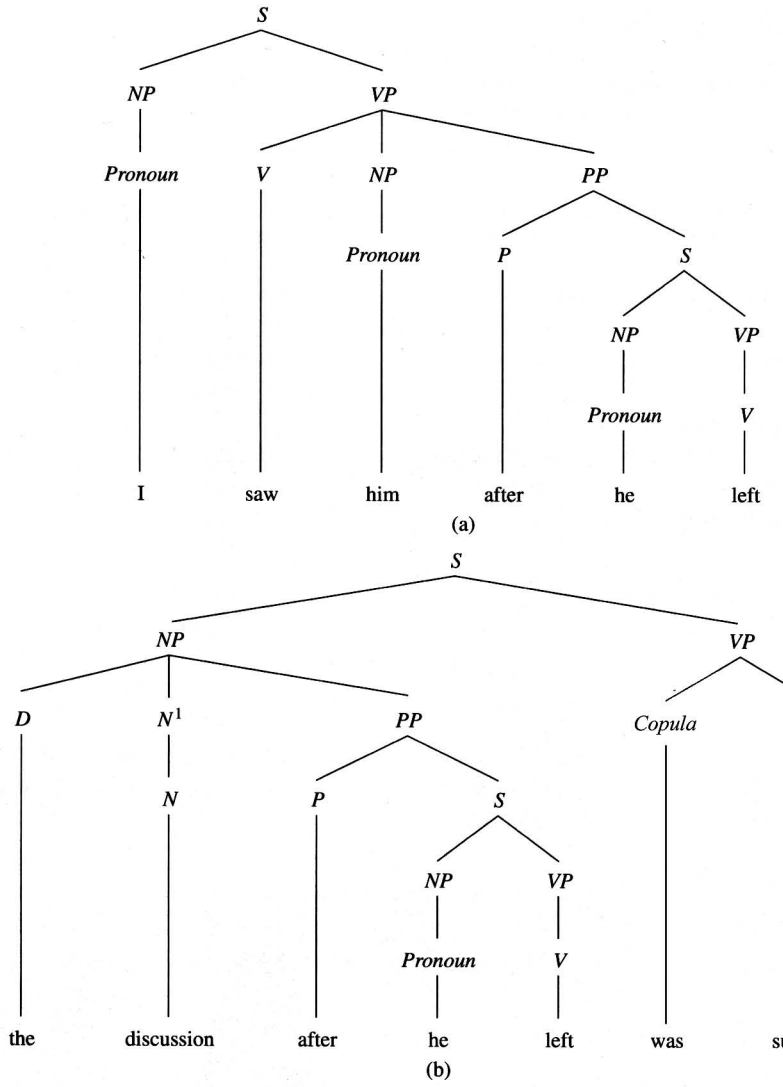


Figure 4.10 Examples of PP containing S.

Exercise 4.5.2.1

The sentence *I heard about the discussion after the meeting* is structurally ambiguous; *after the meeting* modifies either *discussion* or *heard.* Draw trees for its two structures.

Exercise 4.5.2.2

Extend your parser to handle the sentences in Figure 4.10, as well as:

an ID (IMMEDIATE DOMINANCE) rule. It doesn't say in what order the V, NP, PP, and AdvP occur. The order is established by one or more LP (LINEAR PRECEDENCE) rules such as:

$$V < NP$$
$$V < PP$$
$$NP < S^1$$

which say that V precedes NP, V precedes PP, NP precedes S^1, and so on (when they hang from the same node). This is only a partial specification of the ordering. Constituents can occur anywhere as long as they don't violate any LP rules. So if no position is specified for AdvP, AdvP can occur anywhere. ID/LP parsers have been developed (Kilbury 1984, Shieber 1984, Barton 1985, Leiss 1990).

Exercise 4.6.1.1

Extend your parser to handle all the sentences in Figure 4.11, plus the same sentences with *very quickly* in place of *quickly*. You need not add rules for expansions of VP that do not occur in these sentences.

Exercise 4.6.1.2

Convert the ID/LP rules

$$VP \rightarrow \quad V, NP, PP, AdvP$$
$$V < NP$$
$$V < PP$$
$$NP < PP$$

into the complete set of equivalent PS rules.

4.6.2 Postposing of Long Constituents

There is a general tendency in English for long constituents to be POSTPOSED (placed at the very end of the sentence). This is obviously a practical thing to do; it lets the hearer parse as many constituents as possible, thereby obtaining context, before tackling the longest one.

Here's an example. One reason our VP rule is so complicated is that we must parse both

Max [VP *revealed* [NP *the fact*] [PP *at the meeting*]].

and

Max [VP *revealed* [PP *at the meeting*] [NP *the amazing fact that birds fly*]].

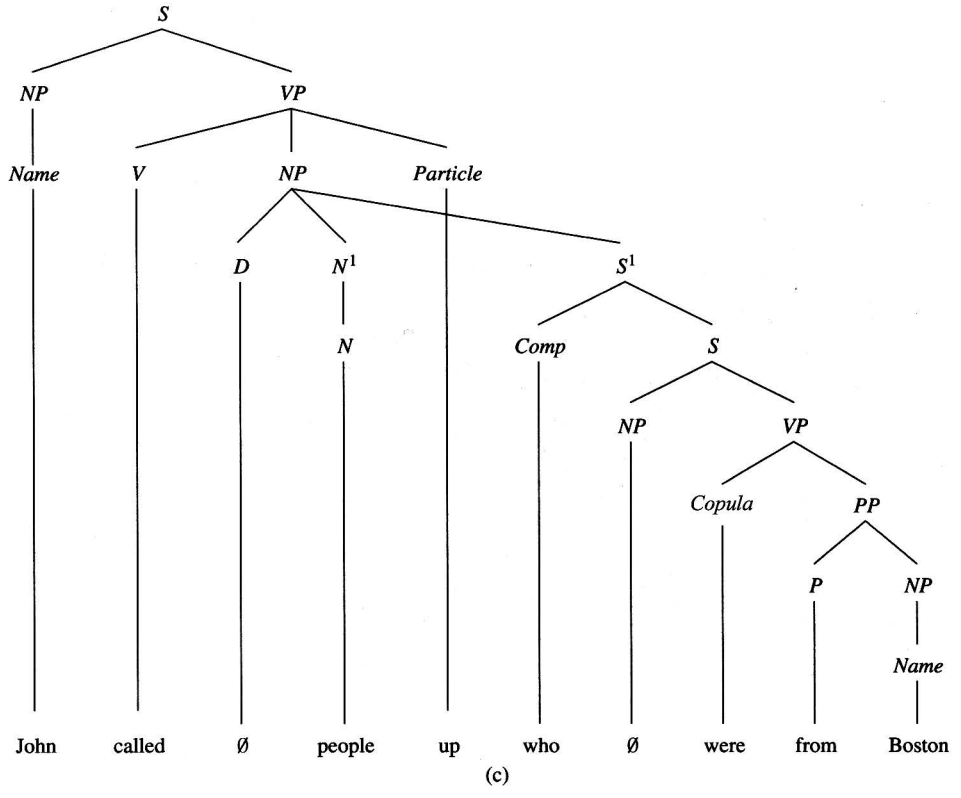


Figure 4.12 cont.

been moved from its original site to the beginning. Examples:

Who said Bill thought Joe believed Fido barked? (Max.)
Who did Max say ␣ thought Joe believed Fido barked? (Bill.)
Who did Max say Bill thought ␣ believed Fido barked? (Joe.)
Who did Max say Bill thought Joe believed ␣ barked? (Fido.)

Here ␣ represents the missing NP.

This phenomenon, called *wh*-movement, occurs not only in questions, but also in exclamations such as

What a noise Max said Fido made ␣!

and in relative clauses (sentences modifying NPs) as in:

the boy who(m) Fido chased ␣ into the garden

to tell the Prolog system that queries to nonexistent predicates should simply fail, rather than raising error conditions. It is not sufficient to declare `chart` as `dynamic`, because when `clear_chart` abolishes `chart`, it will abolish the dynamic declaration too.

Exercise 6.5.2.1

Get the chart parser working and use it to parse *The dog chases the cat near the elephant.* What information is in the chart at the end of the parse?

Exercise 6.5.2.2

Modify your chart parser to display a message whenever the second clause of *parse* succeeds. This will let you know when the chart is actually saving the parser some work. Parse *The dog chases the cat near the elephant* again. What output do you get?

Exercise 6.5.2.3 (small project)

Chart parsers need not be top-down. Implement a bottom-up (shift-reduce) chart parser.

6.5.3 Representing Positions Numerically

We noted that storing whole lists in the chart, as in

```
chart(np,[the,cat,into,the,garden],[into,the,garden]).
```

is inefficient. The lists represent positions in the input string; they can be replaced by word counts (0 for the beginning of the string, 1 for the position after the first word, and so on), so that chart entries look like this:

```
chart(np,0,2).
```

That is: "The NP begins when 0 words have been accepted, and ends when 2 words have been accepted." Numbers are more efficient than lists because they are smaller, can be compared more quickly, and can be distinguished by first-argument indexing.

If we do this, we can get rid of the input "string" altogether, and replace it by a set of facts about what words are in what positions. For example:

```
c(the,0,1).
c(dog,1,2).
c(sees,2,3).
c(the,3,4).
c(cat,4,5).
c(near,5,6).
c(the,6,7).
c(elephant,7,8).
```

It's quicker to look these up than to repeatedly pick a list apart.

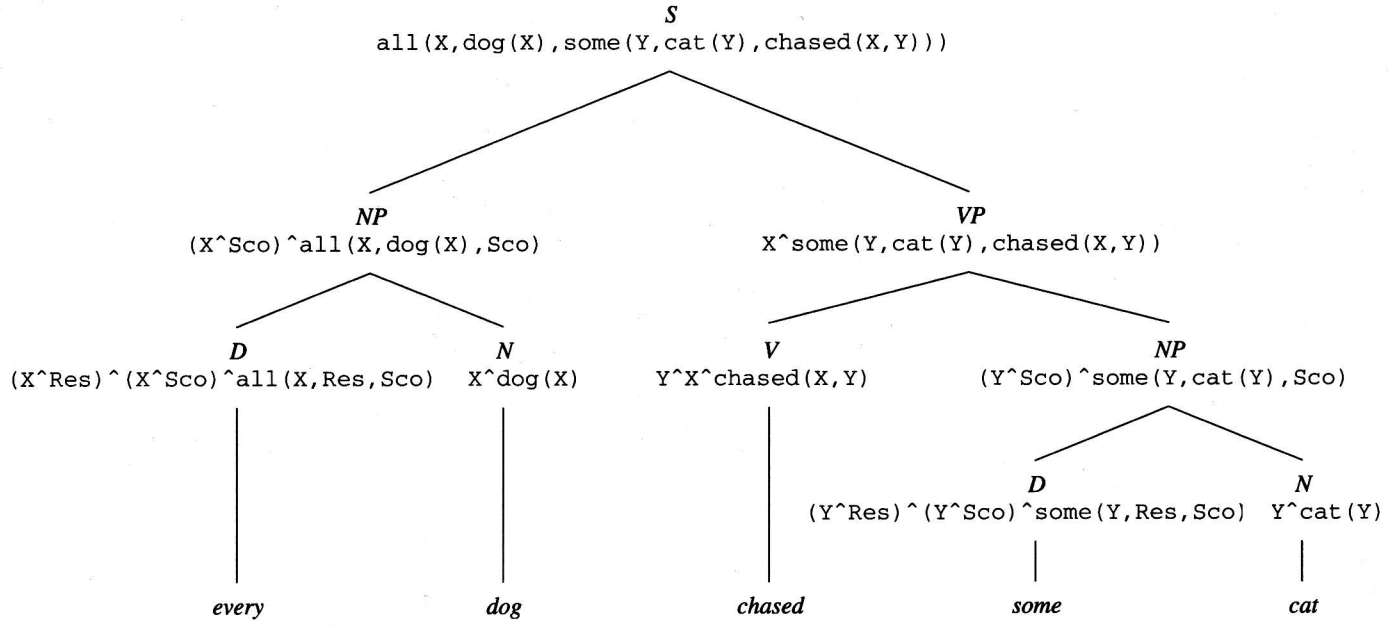


Figure 7.3 Building semantic structure, in detail.