Logik für Informatiker: PROLOG
Part 8: Difference Lists & Grammars

Andreas Karwath

&

Wolfram Burgard

(original slides by Peter Flach)

- Today‘s lecture:

- Difference lists/incomplete data structures
- Definite clause grammars
- Eliza

Outline

Difference Lists and Incomplete Data Structures

- Every list can be represented as the difference of two lists.

  `[1,2,3]` as difference between `[1,2,3,4,5]` and `[4,5]`
  or as difference between `[1,2,3|Xs]` and `Xs`

- Notation: `As\Bs.`

- Such a structure is called *difference list*

- **Why difference lists?**

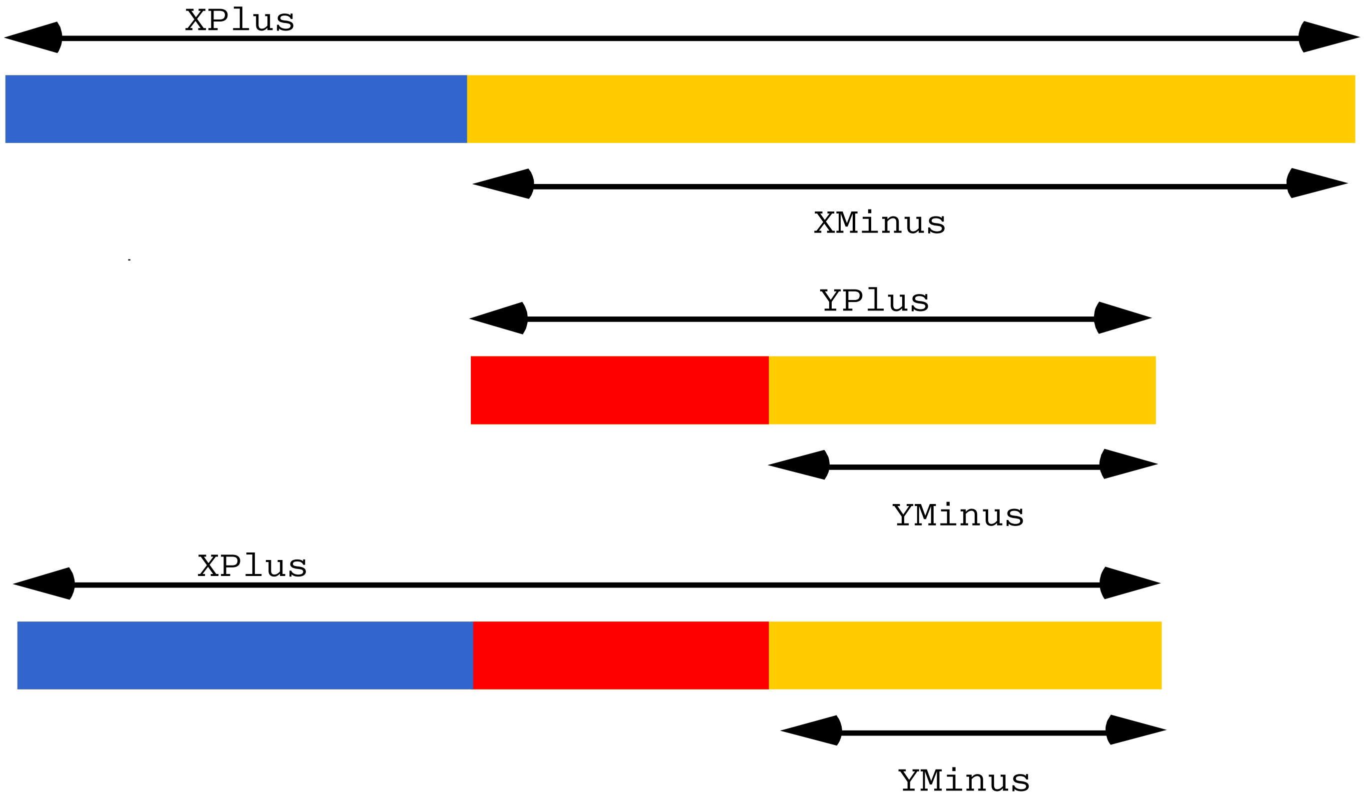
- **Efficiency:** e.g., `append` in constant time

Difference lists

```
/*

     append_dl(As,Bs,Cs) :-

     The difference-list Cs is the result of
     appending Bs to As, where As and Bs are
     compatible difference-lists.

*/

append_dl(Xs\Ys,Ys\Zs,Xs\Zs).
```

Difference lists: `append`

```
append_dl(XPlus\XMinus,YPlus\YMinus,XPlus\YMinus):-XMinus=YPlus.
```



```
?-append_dl([a,b|X]\X,[c,d|Y]\Y,Z).
X = [c,d|Y], Z = [a,b,c,d|Y]\Y
```

Difference lists

```
queue(S) :- queue(S, Q\Q).

queue([], Q).

queue([enqueue(X)|Xs], Q) :-
  enqueue(X, Q, Q1), queue(Xs, Q1).

queue([dequeue(X)|Xs], Q) :-
  dequeue(X, Q, Q1), queue(Xs, Q1).



enqueue(X, Qh\[X|Qt], Qh\Qt).

dequeue(X, [X|Qh]\Qt, Qh\Qt).
```

`first in first out list`

Example: `queue`

```
reverse(Xs,Ys) :- reverse_dl(Xs,Ys\[]).


reverse_dl([],Xs\Xs).

reverse_dl([X|Xs],Ys\Zs) :-
 reverse_dl(Xs,Ys\[X|Zs]).
```

Example: `reverse`

Eliza

from Sterling and Shapiro, The art of Prolog

„I am unhappy.“

>> „How long have you been unhappy?“

„Six months. Can you help me?“

>> „What makes you think I help you?“

„You remind me of my sister.“

>> „Can you tell me more about your sister?“

„I like teasing my sister.“

>> „Does anyone else in your family like teasing your sister?“

„No. Only me.“

>> „Please go on.“

Eliza

- Stimulus-response pair:
  - `I am <statement>.`
  - `How long have you been <statement>?`

- Read the input
  - While the input is not „bye“
    - Choose a stimulus-response pair
    - Match the input to the stimulus
    - Generate the reply from the response and the above match
    - Output the response
    - Read the next input

Eliza

```
/*

   lookup(Key,Dictionary,Value) :-
      Dictionary contains Value indexed
      under key. Dictionary is
      represented as an incomplete list
      of pairs of the form (Key,Value).

*/

lookup(Key,[(Key,Value)|Dict],Value).

lookup(Key,[(Key1,Value1)|Dict],Value)
  :-
      Key \= Key1,
      lookup(Key,Dict,Value).
```

Example: dictionary lookup

```
/*
  lookup(Key,Dictionary,Value) :-
   Dictionary contains the value indexed under
   Key. Dictionary is represented as an ordered
   binary tree.
*/

lookup(Key,dict(Key,X,Left,Right),Value) :-
  !,
  X = Value.

lookup(Key,dict(Key1,X,Left,Right),Value) :-
  Key < Key1,
  lookup(Key,Left,Value).

lookup(Key,dict(Key1,X,Left,Right),Value) :-
  Key > Key1,
  lookup(Key,Right,Value).
```

Example: dictionary lookup

```
/*
  pattern(Stimulus,Response) :-
  Response is an applicable response pattern to the
  pattern Stimulus.
*/

pattern([i,am,1],['How',long,have,you,been,1,?]).
pattern([1,you,2,me],['What',makes,you,think,'I',2,you,?]).
pattern([i,like,1],['Does',anyone,else,in,your,family,like,1,?]).
pattern([i,feel,1],['Do',you,often,feel,that,way,?]).
pattern([1,X,2],['Please',you,tell,me,more,about,X]) :-
  important(X).
pattern([1],['Please',go,on,'.']).

important(father).
important(mother).
important(sister).
important(brother).
important(son).
important(daughter).
```

Eliza (1)

Eliza (2)

```
reply([]) :- nl.

reply([Head|Tail]) :-
  write(Head),
  write(' '),
  reply(Tail).
```

```
lookup(X,[(X,V)|XVs],V).

lookup(X,[(X1,V1)|XVs],V) :-
  X \= X1,
  lookup(X,XVs,V).
```

```
/*

    eliza :- Simulates a conversation via side
   effects.

*/

eliza :- read(Input), eliza(Input), !.

eliza([bye]) :-
   writeln(['Goodbye. I hope I have helped you']).

eliza(Input) :-
   pattern(Stimulus,Response),
   match(Stimulus,Table,Input),
   match(Response,Table,Output),
   reply(Output),
   read(Input1),
   !,
   eliza(Input1).
```

Eliza (3)

```
/*
  match(Patterm,Dictionary,Words) :-
     Pattern matches the list of words Words, and
     matchings are recorded in the Dictionary.
*/

match([N|Pattern],Table,Target) :-
  integer(N),
  lookup(N,Table,LeftTarget),
  append(LeftTarget,RightTarget,Target),
  match(Pattern,Table,RightTarget).

match([Word|Pattern],Table,[Word|Target]) :-
  atom(Word),
  match(Pattern,Table,Target).

match([],Table,[]).
```

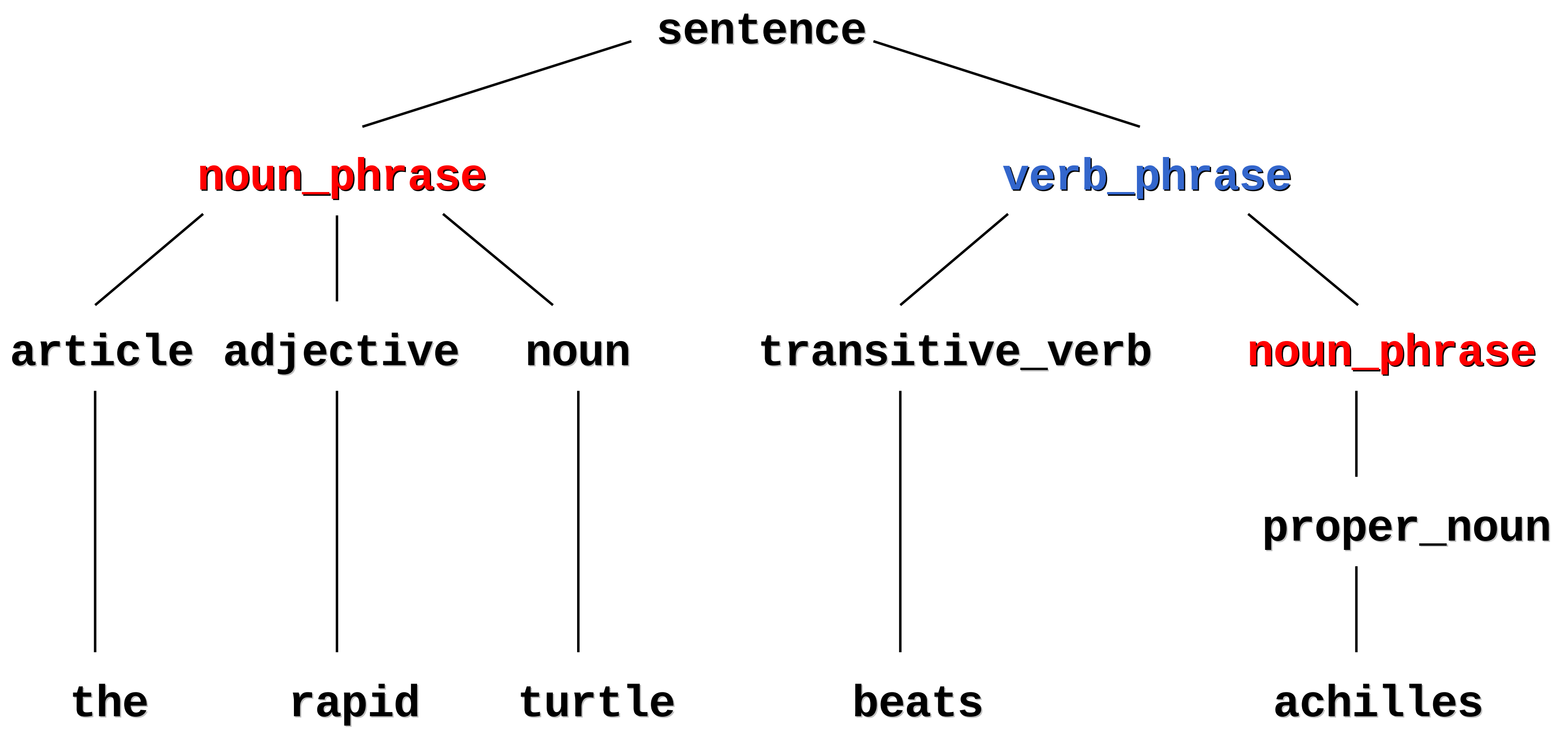
Eliza (4)

Definite Clause Grammars

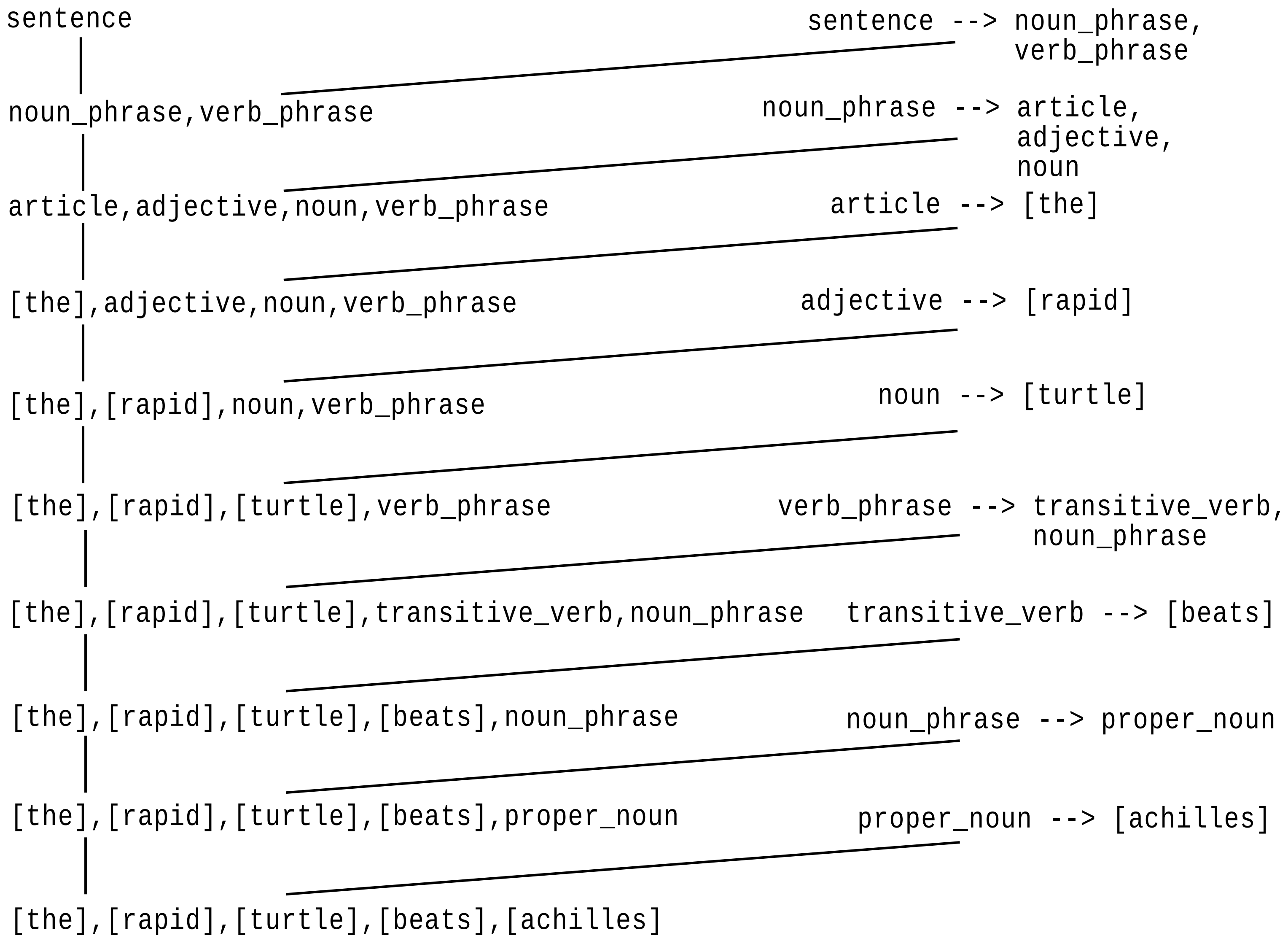
from Flach, Chapter 7

Context-free grammar

```
sentence            --> noun_phrase,verb_phrase.
noun_phrase         --> proper_noun.
noun_phrase         --> article,adjective,noun.
noun_phrase         --> article,noun.
verb_phrase         --> intransitive_verb.
verb_phrase         --> transitive_verb,noun_phrase.
article             --> [the].
adjective           --> [lazy].
adjective           --> [rapid].
proper_noun         --> [achilles].
noun                --> [turtle].
intransitive_verb   --> [sleeps].
transitive_verb     --> [beats].
```


sentence
noun_phrase
verb_phrase
article
adjective
noun
transitive_verb
noun_phrase
proper_noun
the
rapid
turtle
beats
achilles

Parse tree



sentence

sentence --> noun_phrase,
             verb_phrase

noun_phrase,verb_phrase

noun_phrase --> article,
                adjective,
                noun

article,adjective,noun,verb_phrase

article --> [the]

[the],adjective,noun,verb_phrase

adjective --> [rapid]

[the],[rapid],noun,verb_phrase

noun --> [turtle]

[the],[rapid],[turtle],verb_phrase

verb_phrase --> transitive_verb,
                noun_phrase

[the],[rapid],[turtle],transitive_verb,noun_phrase

transitive_verb --> [beats]

[the],[rapid],[turtle],[beats],noun_phrase

noun_phrase --> proper_noun

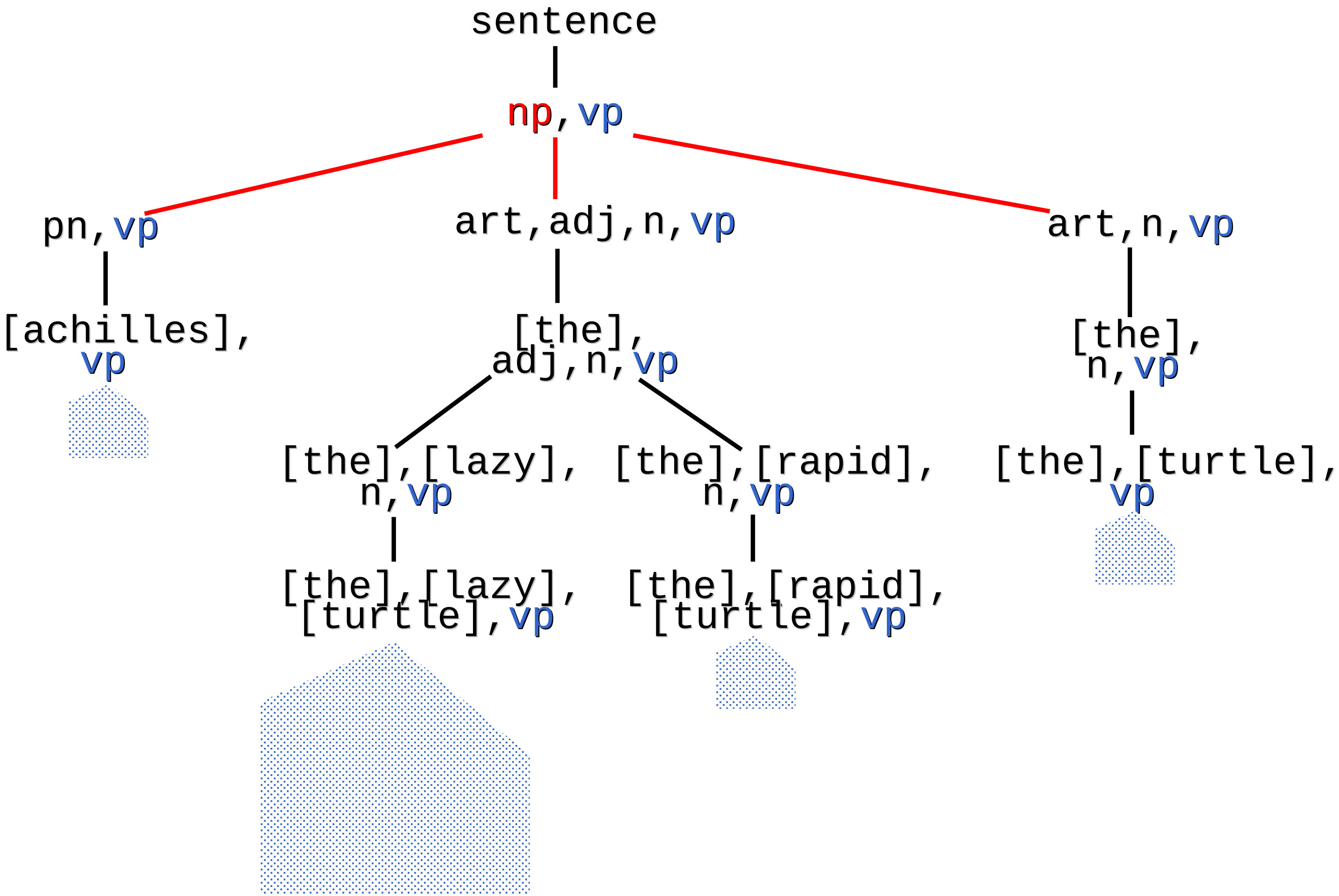
[the],[rapid],[turtle],[beats],proper_noun

proper_noun --> [achilles]

[the],[rapid],[turtle],[beats],[achilles]

Exercise 7.1

Exercise 7.2 (1)



sentence

np,vp

pn,vp

[achilles],vp

art,adj,n,vp

[the],adj,n,vp

[the],[lazy],n,vp

[the],[lazy],[turtle],vp

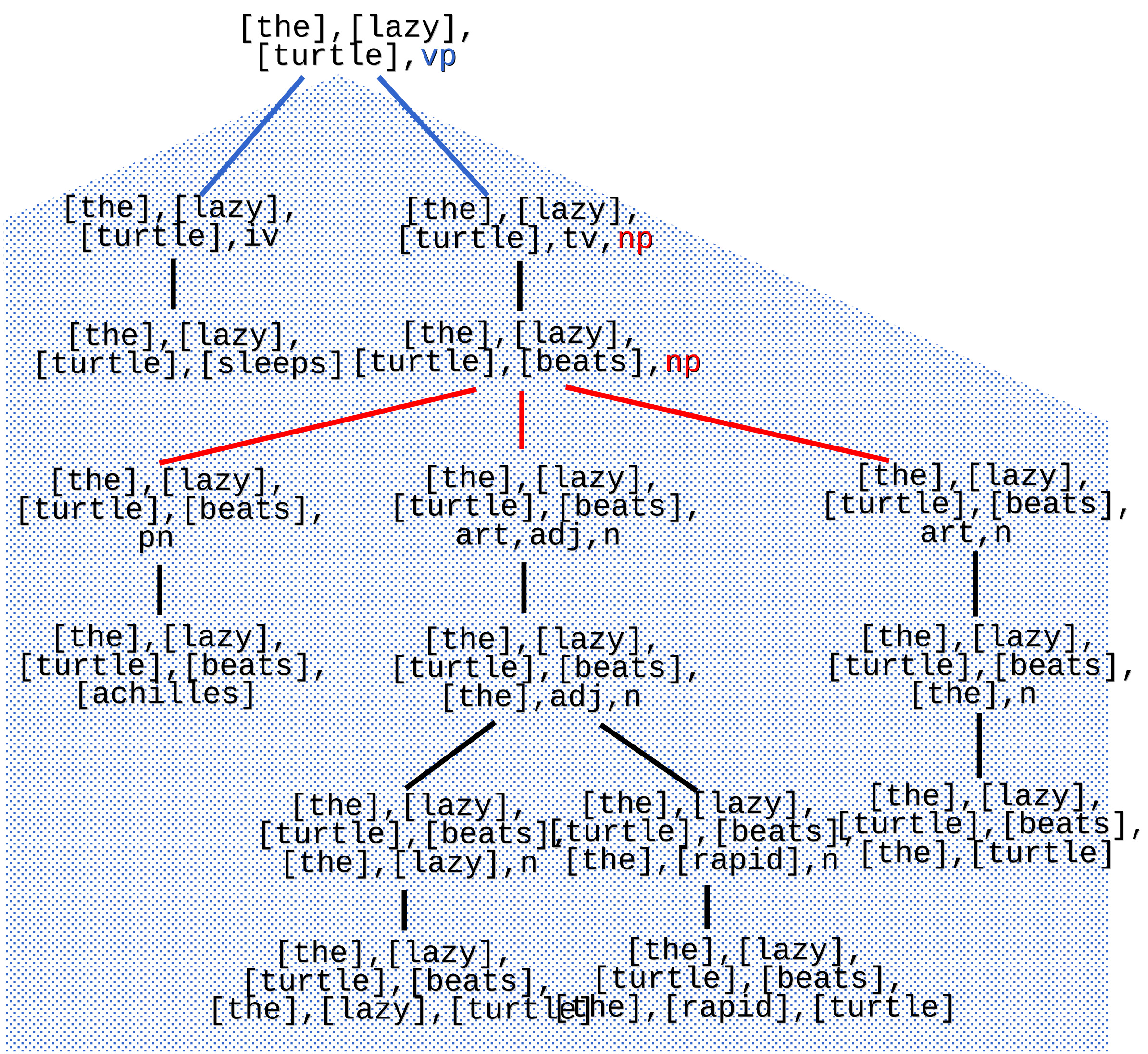
[the],[rapid],n,vp

[the],[rapid],[turtle],vp

art,n,vp

[the],n,vp

[the],[turtle],vp



[the],[lazy],[turtle],vp

[the],[lazy],[turtle],iv

[the],[lazy],[turtle],tv,np

[the],[lazy],[turtle],[sleeps]

[the],[lazy],[turtle],[beats],np

[the],[lazy],[turtle],[beats],pn

[the],[lazy],[turtle],[beats],art,adj,n

[the],[lazy],[turtle],[beats],art,n

[the],[lazy],[turtle],[beats],[achilles]

[the],[lazy],[turtle],[beats],[the],adj,n

[the],[lazy],[turtle],[beats],[the],n

[the],[lazy],[turtle],[beats],[the],[lazy],n

[the],[lazy],[turtle],[beats],[the],[rapid],n

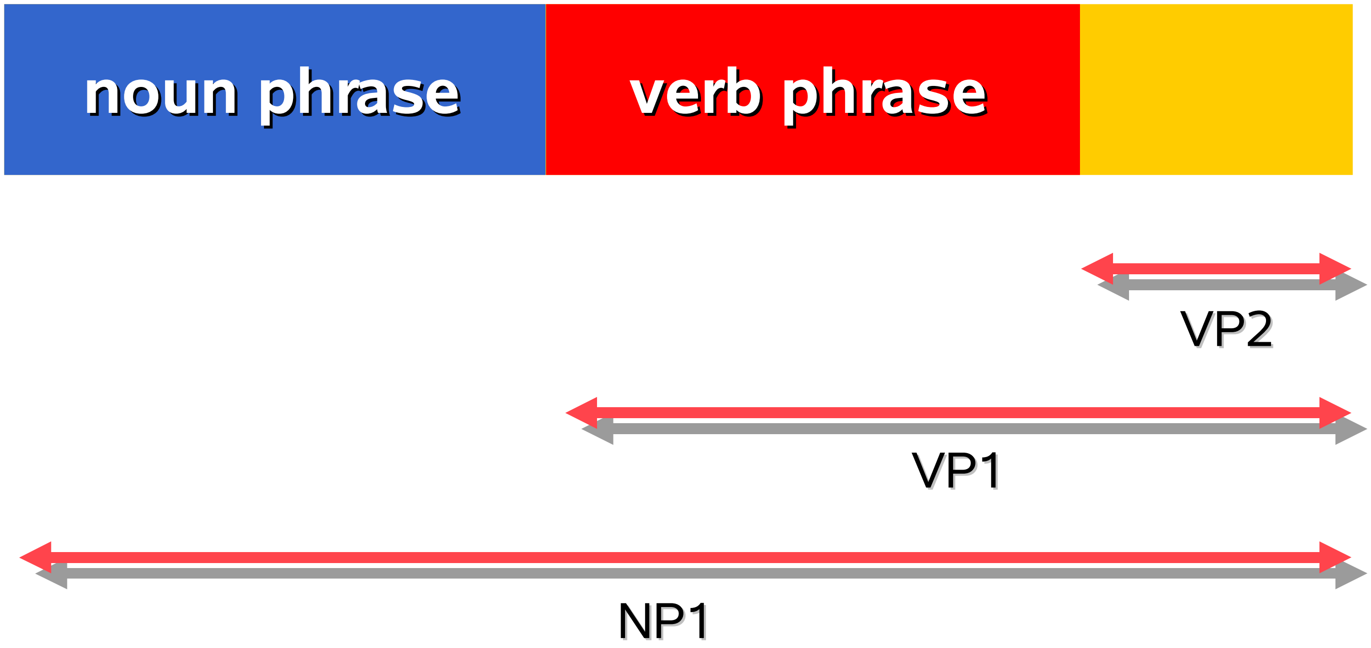
[the],[lazy],[turtle],[beats],[the],[turtle]

[the],[lazy],[turtle],[beats],[the],[lazy],[turtle]

[the],[lazy],[turtle],[beats],[the],[rapid],[turtle]

Exercise 7.2 (2)

Difference lists in grammar rules



```
sentence(NP1-VP2):-
      noun_phrase(NP1-VP1),
      verb_phrase(VP1-VP2)
```

Meta-level vs. object-level

	GRAMMAR	**PARSING**
META-LEVEL	`s --> np,vp`	`?-phrase(s,L)`

OBJECT-LEVEL	`s(L,L0):-`
	`np(L,L1),`
	`vp(L1,L0)`
		`?-s(L,[])`

```
sentence                --> noun_phrase(N),verb_phrase(N).
noun_phrase             --> article(N),noun(N).
verb_phrase             --> intransitive_verb(N).
article(singular)       --> [a].
article(singular)       --> [the].
article(plural)         --> [the].
noun(singular)          --> [turtle].
noun(plural)            --> [turtles].
intransitive_verb(singular)  --> [sleeps].
intransitive_verb(pluralr)   --> [sleep].
```

Non-terminals with arguments

Constructing parse trees

```
sentence(s(NP,VP))         --> noun_phrase(NP),verb_phrase(VP)
noun_phrase(np(N))         --> proper_noun(N).
noun_phrase(np(Art,Adj,N))  --> article(Art),adjective(Adj),
                                 noun(N).
noun_phrase(np(Art,N))     --> article(Art),noun(N).
verb_phrase(vp(IV))        --> intransitive_verb(IV).
verb_phrase(vp(TV,NP))     --> transitive_verb(TV),
                               noun_phrase(NP).
article(art(the))          --> [the].
adjective(adj(lazy))       --> [lazy].
adjective(adj(rapid))      --> [rapid].
proper_noun(pn(achilles))   --> [achilles].
noun(n(turtle))            --> [turtle].
intransitive_verb(iv(sleeps))  --> [sleeps].
transitive_verb(tv(beats))  --> [beats].
```

```
?-phrase(sentence(T),[achilles,beats,the,lazy,turtle])
  T = s(np(pn(achilles)),
        vp(tv(beats),
           np(art(the),
              adj(lazy),
              n(turtle))))
```

Prolog goals in grammar rules

```
numeral(N)    --> n1_999(N).
numeral(N)    --> n1_9(N1),[thousand],n1_999(N2),
                  {N is N1*1000+N2}.
n1_999(N)     --> n1_99(N).
n1_999(N)     --> n1_9(N1),[hundred],n1_99(N2),
                  {N is N1*100+N2}.
n1_99(N)      --> n0_9(N).
n1_99(N)      --> n10_19(N).
n1_99(N)      --> n20_90(N).
n1_99(N)      --> n20_90(N1),n1_9(N2),{N is N1+N2}.
n0_9(0)       --> [].
n0_9(N)       --> n1_9(N).
n1_9(1)       --> [one].
n1_9(2)       --> [two].
              …
n10_19(10)    --> [ten].
n10_19(11)    --> [eleven].
              …
n20_90(20)    --> [twenty].
n20_90(30)    --> [thirty].
              …
```

```
?-phrase(numeral(2211),N).
N = [two,thousand,two,hundred,eleven]
```

- ☞ The meaning of the **proper noun ‘Socrates’** is **the term `socrates`**

```
proper_noun(socrates) --> [socrates].
```

- ☞ The meaning of the **property ‘mortal’** is **a mapping from terms to literals containing the unary predicate `mortal`**

```
property(X=>mortal(X))--> [mortal].
```

- ☞ The meaning of a **proper noun - verb phrase sentence** is **a clause with empty body and head obtained by applying the meaning of the verb phrase to the meaning of the proper noun**

```
sentence((L:-true))   --> proper_noun(X),verb_phrase(X=>L).
?-phrase(sentence(C),[socrates,is,mortal]).
C = (mortal(socrates):-true)
```

Interpretation

Exercise 7.4

- ☞ A transitive verb is a binary mapping from a pair of terms to literals

  -
    ```
    transitive_verb(Y=>X=>likes(X,Y)) --> [likes].
    ```
- ☞ A proper noun instantiates one of the arguments, returning a unary mapping

  -
    ```
    verb_phrase(M) -->
    transitive_verb(Y=>M),proper_noun(Y).
    ```

```
sentence((L:-true))  -->
proper_noun(X),verb_phrase(X=>L).
sentence((H:-B)) -->
[every],noun(X=>B),verb_phrase(X=>H).
% NB. separate ‘determiner’ rule removed, see later

verb_phrase(M)               --> [is],property(M).

property(M)                  --> [a],noun(M).
property(X=>mortal(X))       --> [mortal].

proper_noun(socrates)        --> [socrates].

noun(X=>human(X))            --> [human].
```

Interpretation (2)

```
?-phrase(sentence(C),S).

C = human(X):-human(X)
S = [every,human,is,a,human];

C = mortal(X):-human(X)
S = [every,human,is,mortal];

C = human(socrates):-true
S = [socrates,is,a,human];

C = mortal(socrates):-true
S = [socrates,is,mortal];
```

Interpretation (3)

Determiners

- ‘Determiner’ sentences have the form ‘every/some [noun] [verb-phrase]’ (NB. meanings of ‘some’ sentences require 2 clauses)
  - `sentence(Cs) --> determiner(M1,M2,Cs),noun(M1),verb_phrase(M2).`
  - `determiner(X=>B, X=>H,[(H:-B)]) --> [every].`
  - `determiner(sk=>H1,sk=>H2,[(H1:-true),(H1:-true)] --> [some].`

  - `?-phrase(sentence(Cs),[D,human,is,mortal]).`
  - `D = every, Cs = [(mortal(X):-human(X))];`
  - `D = some, Cs = [(human(sk):-true),(mortal(sk):-true)]`

Questions

```
question(Q)        --> [who],[is],property(X=>Q).


question(Q)        -->
[is],proper_noun(X),property(X=>Q).


question((Q1,Q2)) --> [is],[some],noun(sk=>Q1),
                          property(sk=>Q2).
```

```
handle_input(Question,Rulebase):-
phrase(question(Query),Question),      % question
prove_rb(Query,Rulebase),!,            % it can be
solved
transform(Query,Clauses),              % transform to

phrase(sentence(Clauses),Answer),      % answer
show_answer(Answer),
nl_shell(Rulebase).
```

Querying a rulebase