

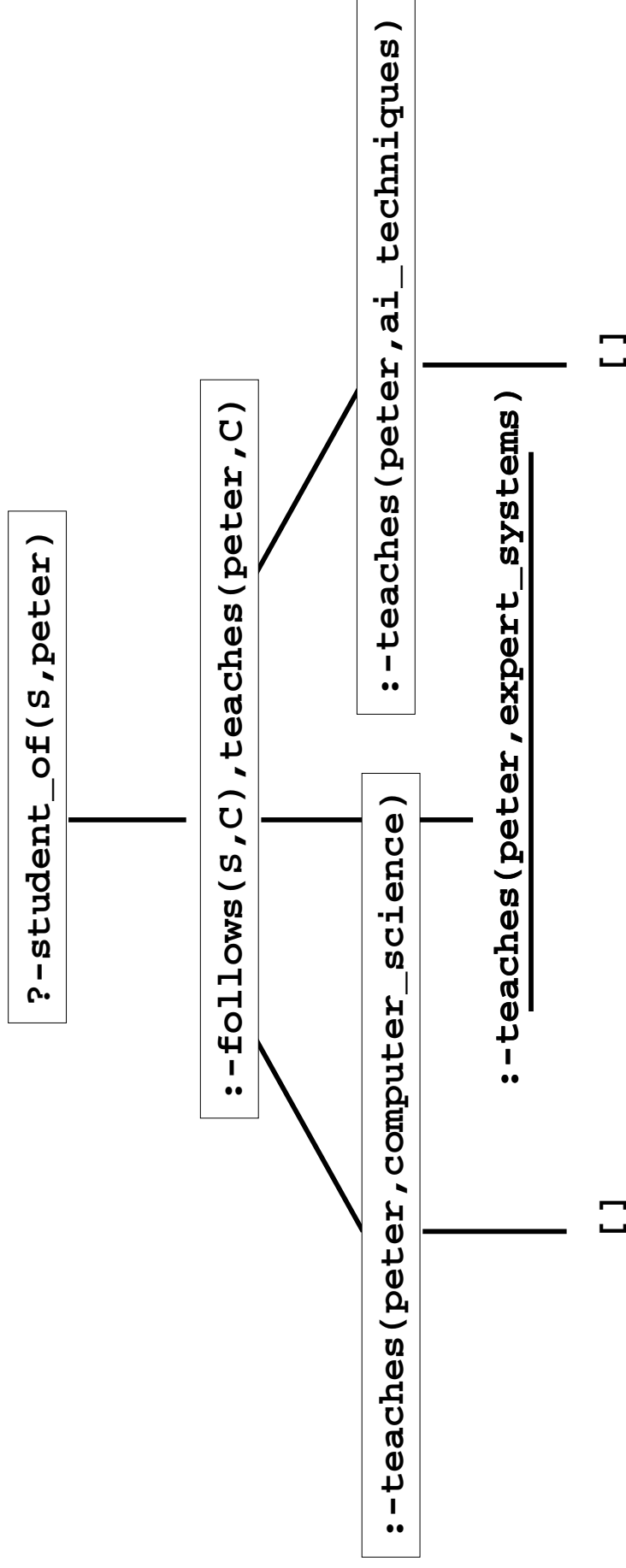
Prolog Programming

We follow Peter Flach's Book
Simply Logical, John Wiley.

```

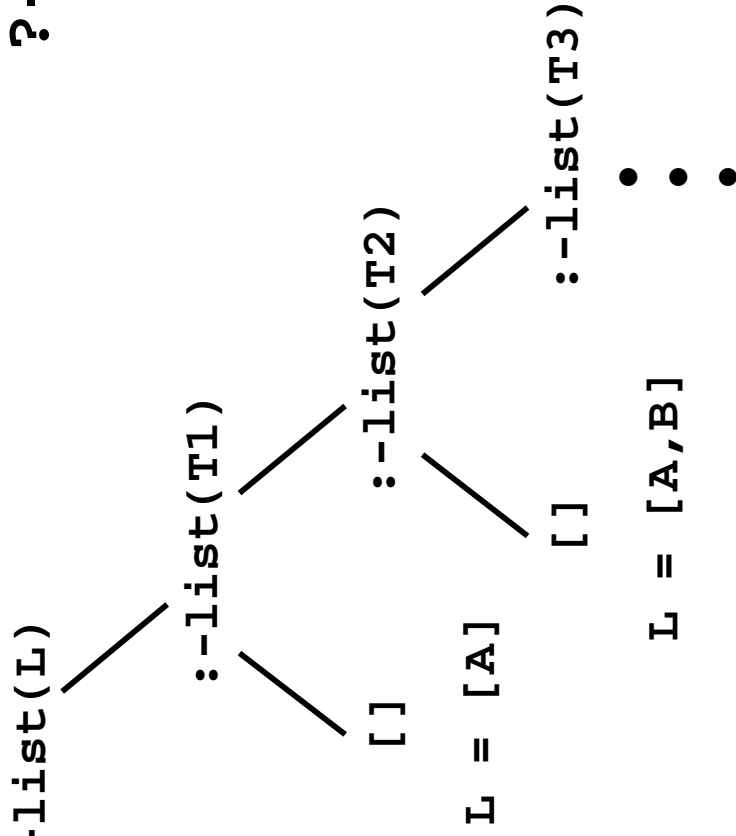
student_of(X,T):-follows(X,C),teaches(T,C).
follows(paul,computer_science).
follows(paul,expert_systems).
follows(maria,ai_techniques).
teaches(adrian,expert_systems).
teaches(peter,ai_techniques).
teaches(peter,computer_science).

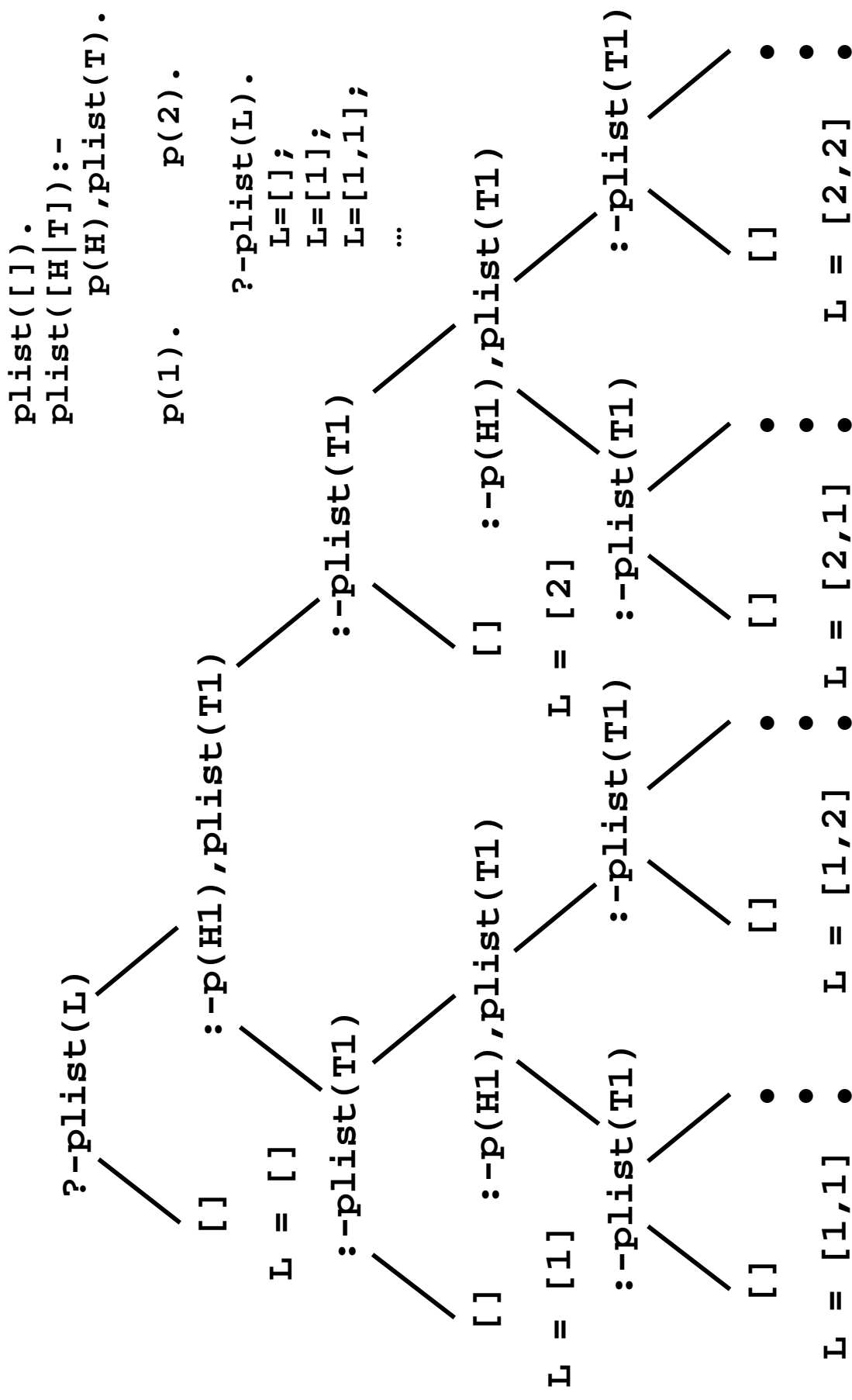
```




```
list([]).
list([H|T]):-list(T).
```

```
?-list(L).
L = [];
L = [A];
L = [A,B];
...
```





$p(x, y) :- q(x, y).$

$p(x, y) :- r(x, y).$

$q(x, y) :- s(x), !, t(y).$

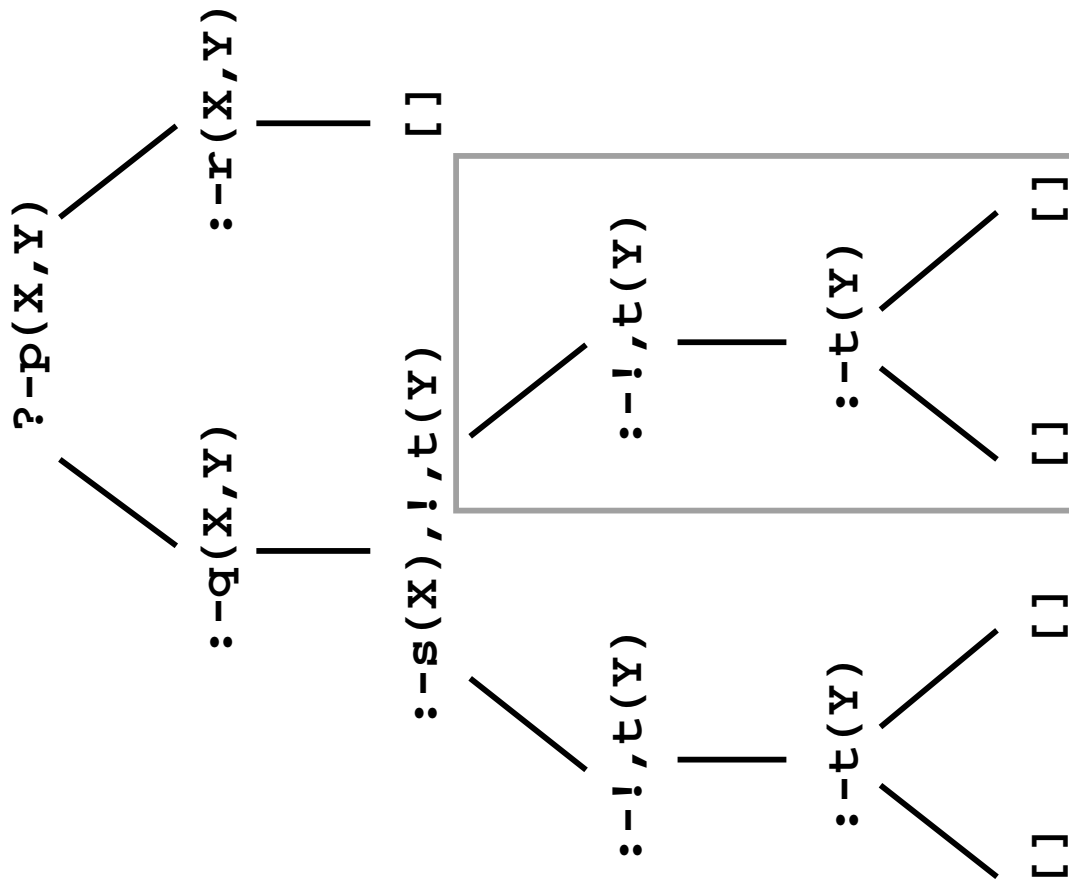
$r(c, d).$

$s(a).$

$s(b).$

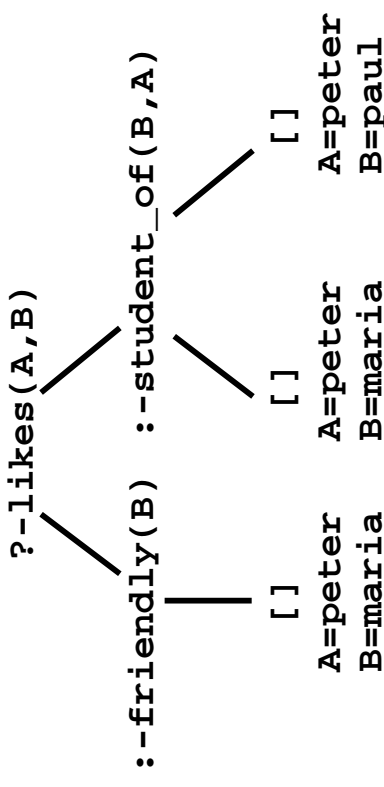
$t(a).$

$t(b).$

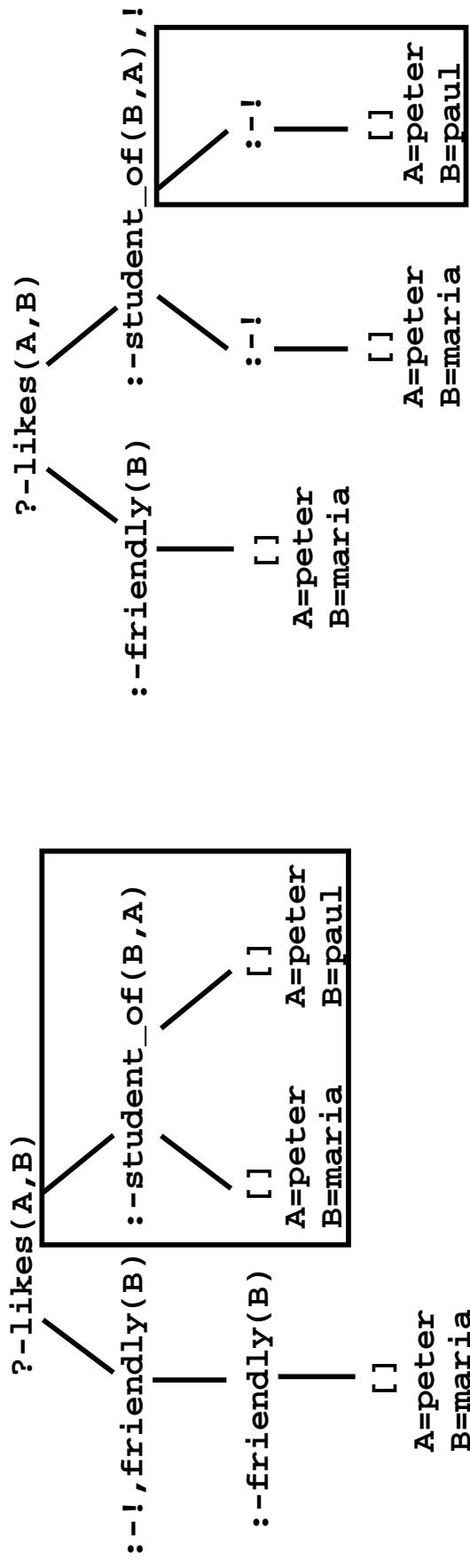


The effect of cut


```
likes(peter, Y) :- friendly(Y).
likes(T, S) :- student_of(S, T).
student_of(maria, peter).
student_of(paul, peter).
friendly(maria).
```

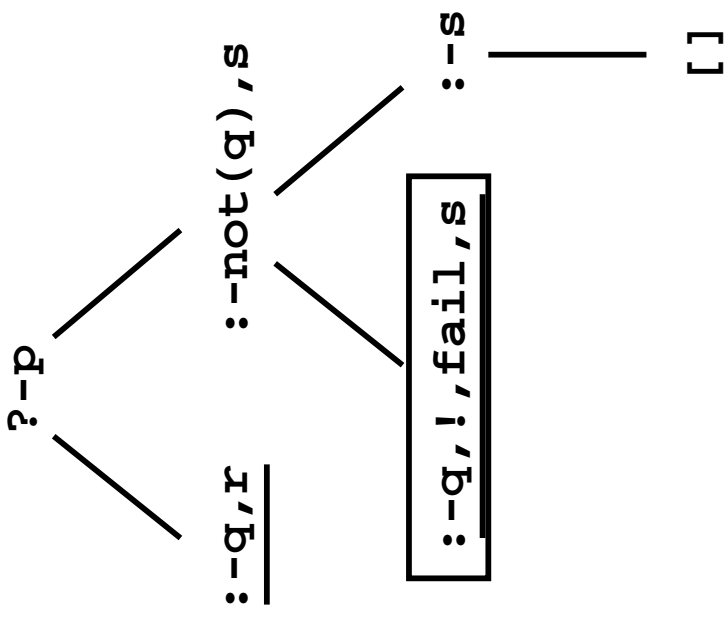
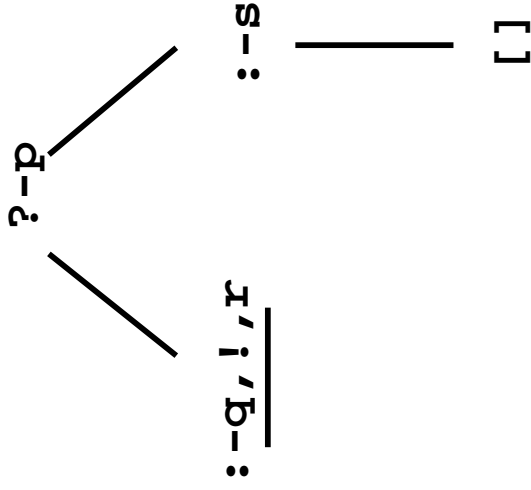


```
likes(peter, Y) :- !, friendly(Y). likes(T, S) :- student_of(S, T), !.
```



p:-q,r.
p:-not(q),s.
s.

not(Goal):-Goal,! ,fail.
not(Goal).



p:-q,! ,r.
p:-s.
s.

```

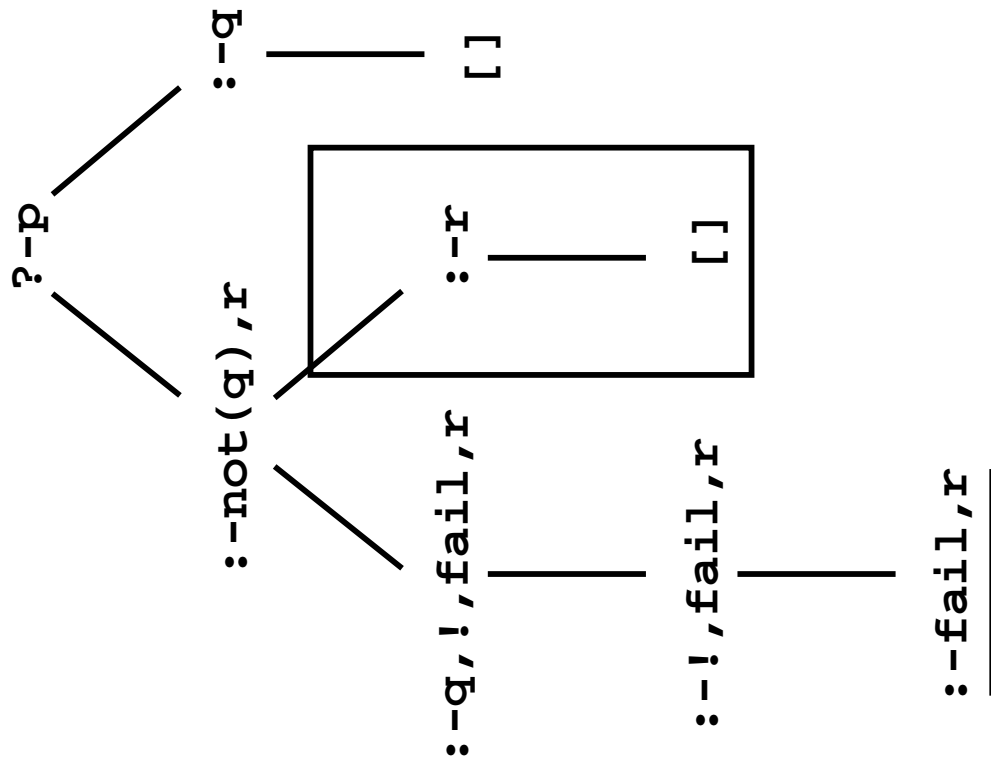
p:-not(q),r.
p:-q.
q.
r.

```

```

not(Goal):-Goal,!,fail.
not(Goal).

```

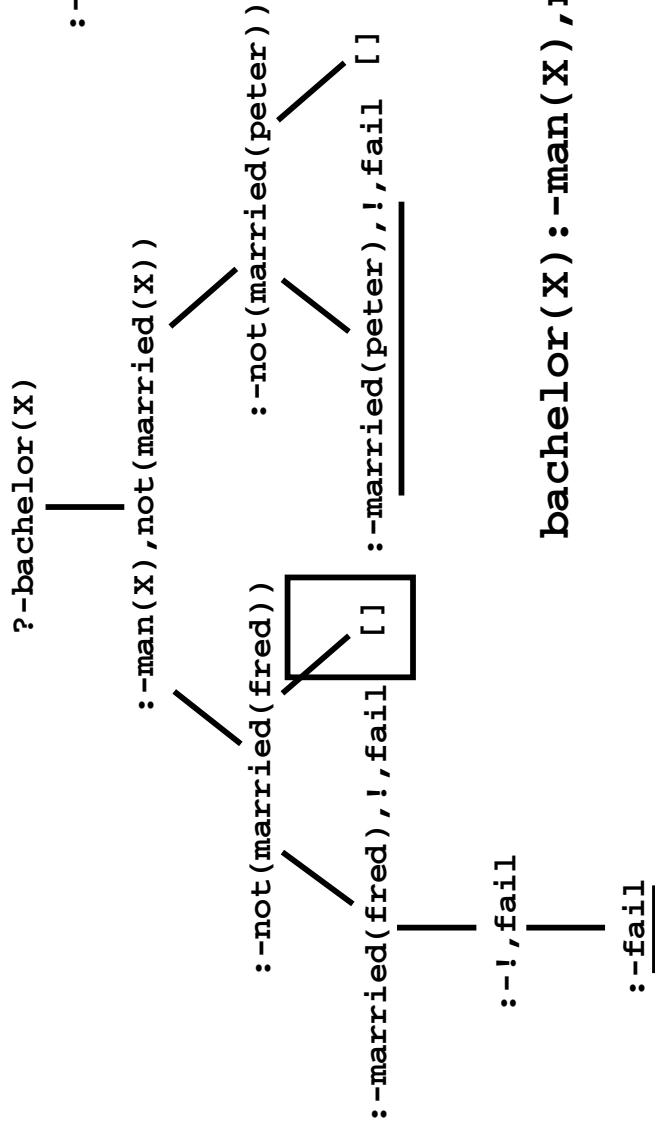
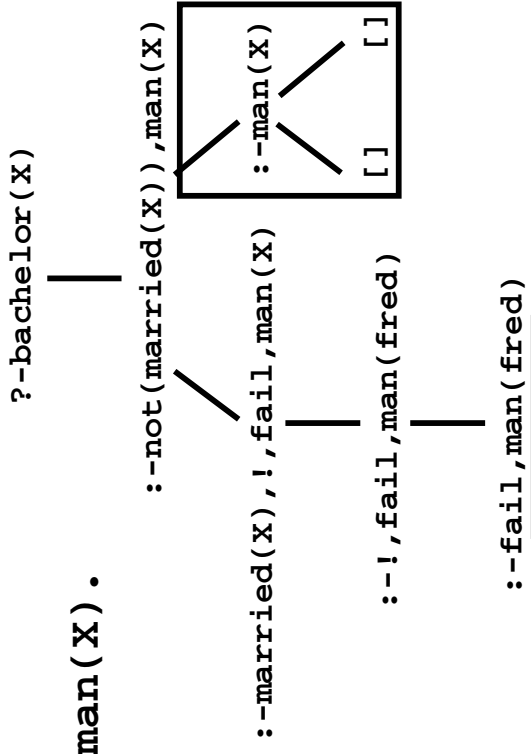


:-not(q) fails

```

bachelor(X) :- not(married(X)), man(X).
man(fred).
man(peter).
married(fred).

```



```

bachelor(X) :- man(X), not(married(X)).

```

Prolog's not is unsound

?-X is 5+7-3.
X = 9

?-X = 5+7-3.
X = 5+7-3

?-9 is 5+7-3.
Yes

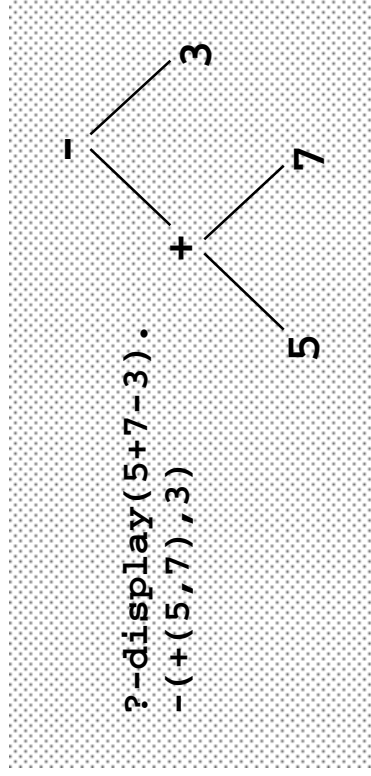
?-9 = 5+7-3.
No

?-9 is X+7-3.
Error in arithmetic expression

?-9 = X+7-3.
No

?-X is 5*3+7/2.
X = 18.5

?-X = Y+7-3.
X = _947+7-3
Y = _947




```

?-length([a,b,c],N)  length([H|T],N1):-length(T,M1),
                    N1 is M1+1
                    {H->a, T->[b,c], N1->N}

:-length([b,c],M1),  length([H|T],N2):-length(T,M2),
N is M1+1           N2 is M2+1
                    {H->b, T->[c], N2->M1}

:-length([c],M2),   length([H|T],N3):-length(T,M3),
M1 is M2+1,         N3 is M3+1
N is M1+1           {H->c, T->[], N3->M2}

:-length([],M3),    length([],0)
M2 is M3+1,
M1 is M2+1,
N is M1+1           {M3->0}

:-M2 is 0+1,
M1 is M2+1,
N is M1+1           {M2->1}

:-M1 is 1+1,
N is M1+1           {M1->2}

:-N is 2+1
                    {N->3}

                    []

```

```

length([],0).
length([H|T],N):-
length(T,M),
N is M+1.

```

```

?-length_acc([a,b,c],0,N)   length_acc([H|T],N10,N1):-N11 is N10+1,
                           length_acc(T,N11,N1)
                           {H->a, T->[b,c], N10->0, N1->N}

:-N11 is 0+1,
 length_acc([b,c],N11,N)
 {N11->1}

:-length_acc([b,c],1,N)   length_acc([H|T],N20,N2):-N21 is N20+1,
                           length_acc(T,N21,N2)
                           {H->b, T->[c], N20->1, N2->N}

:-N21 is 1+1,
 length_acc([c],N21,N)
 {N21->2}

:-length_acc([c],2,N)   length_acc([H|T],N30,N3):-N31 is N30+1,
                           length_acc(T,N31,N3)
                           {H->c, T->[], N30->2, N3->N}

:-N31 is 2+1,
 length_acc([],N31,N)
 {N31->3}

:-length_acc([],3,N)   length_acc([],N,N)
 {N->3}
 []

```

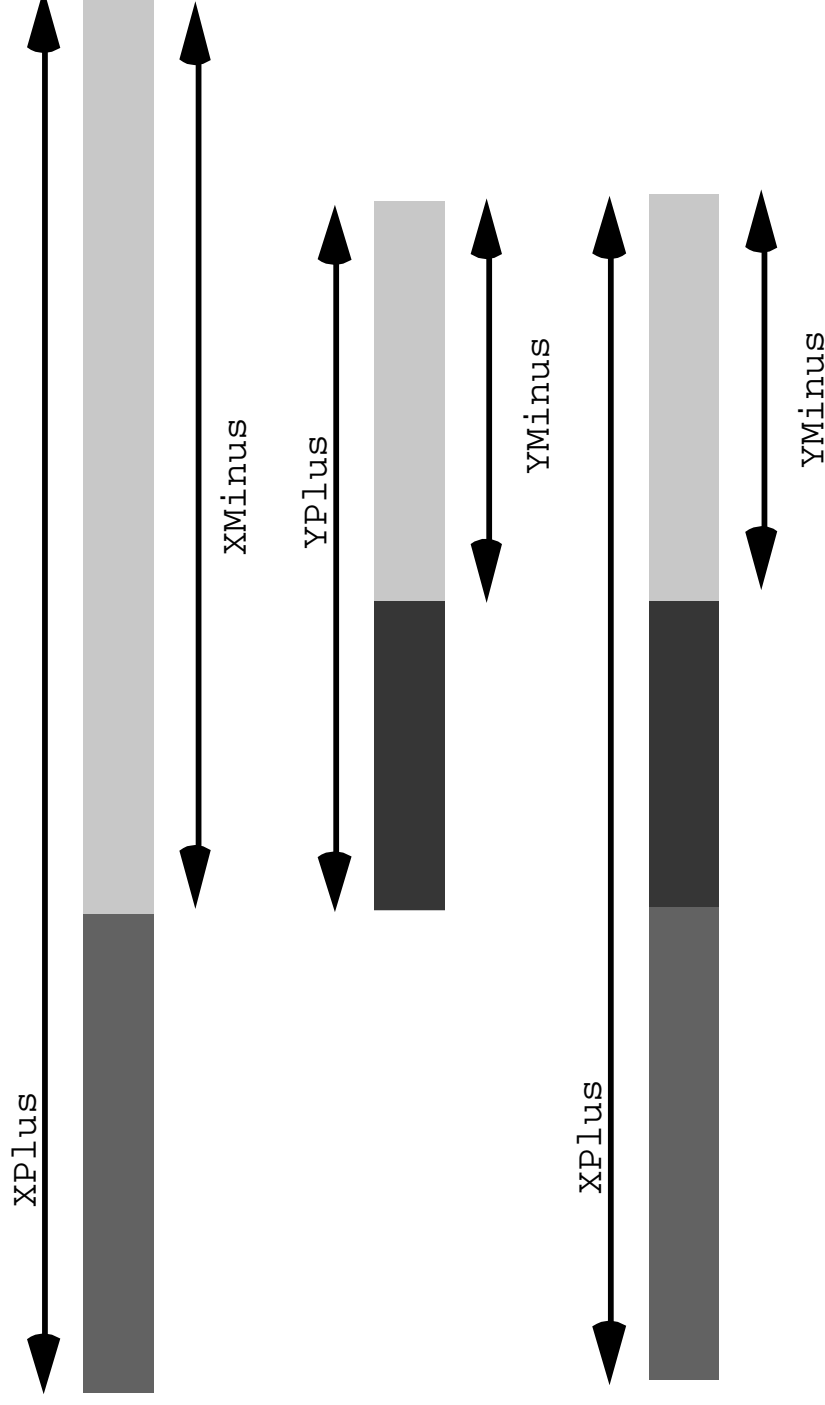
```

length_acc([],N,N).
length_acc([H|T],N0,N):-
    N1 is N0+1,
    length_acc(T,N1,N).

```

Exercise 3.11

```
append_d1(XPlus-XMinus, YPlus-YMinus, XPlus-YMinus) :- XMinus=YPlus.
```



```
?-append_d1([a,b|X]-X,[c,d|Y]-Y,Z).
X = [c,d|Y], Z = [a,b,c,d|Y]-Y
```

☞ **Very often employ**

Grammar formalisms
Logic
Unification

☞ **We : very brief introduction to**

Definite clause grammars

Just to give an idea / to illustrate

Computer science point of view !

Parsing

Analyzing the syntactic structure of sentences
Construct the parse tree

Interpretation

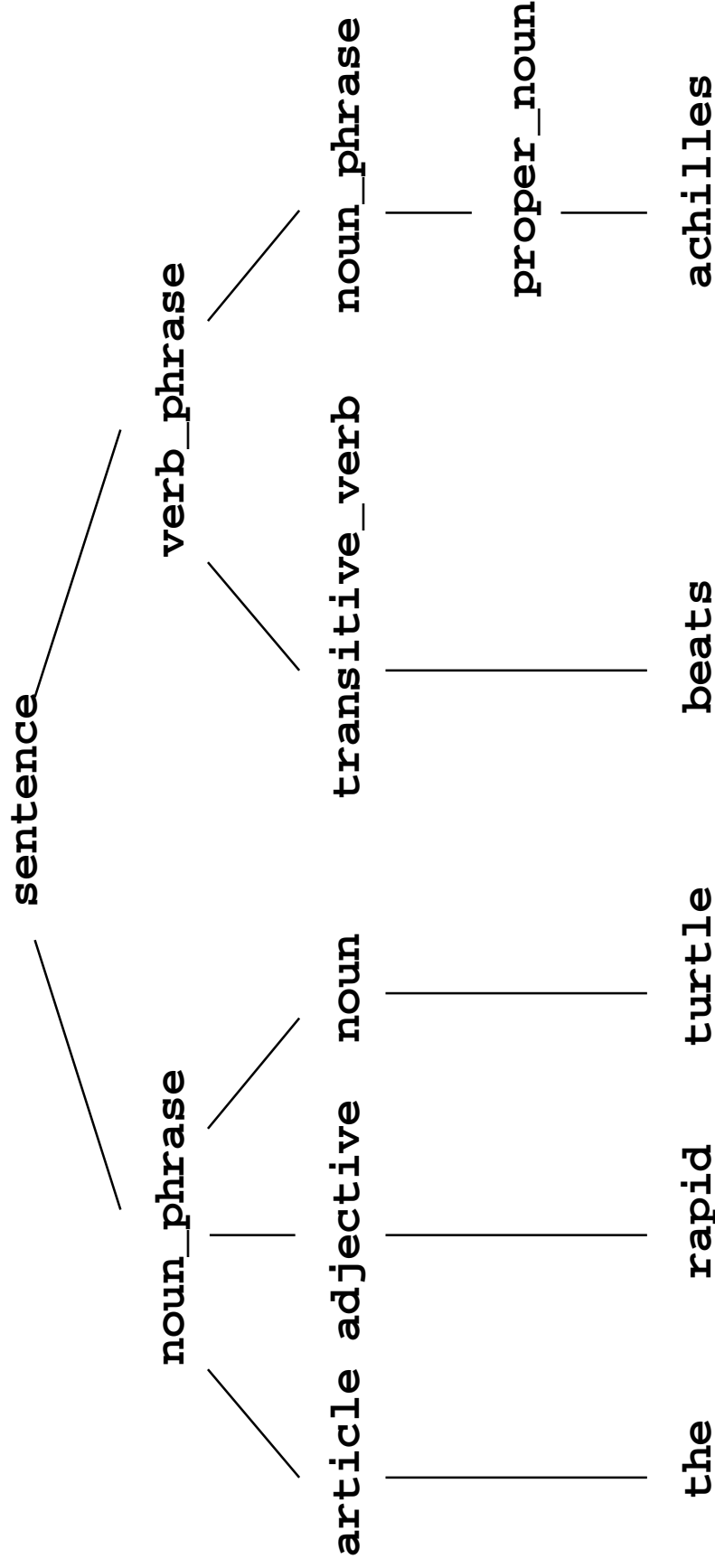
Determine the meaning (internal representation thereof) of the sentence
Internal representation can then be used to reason

Example Application

NLP interface to a database

```
sentence
noun_phrase
noun_phrase
noun_phrase
verb_phrase
verb_phrase
article
adjective
adjective
proper_noun
noun
intransitive_verb
transitive_verb

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



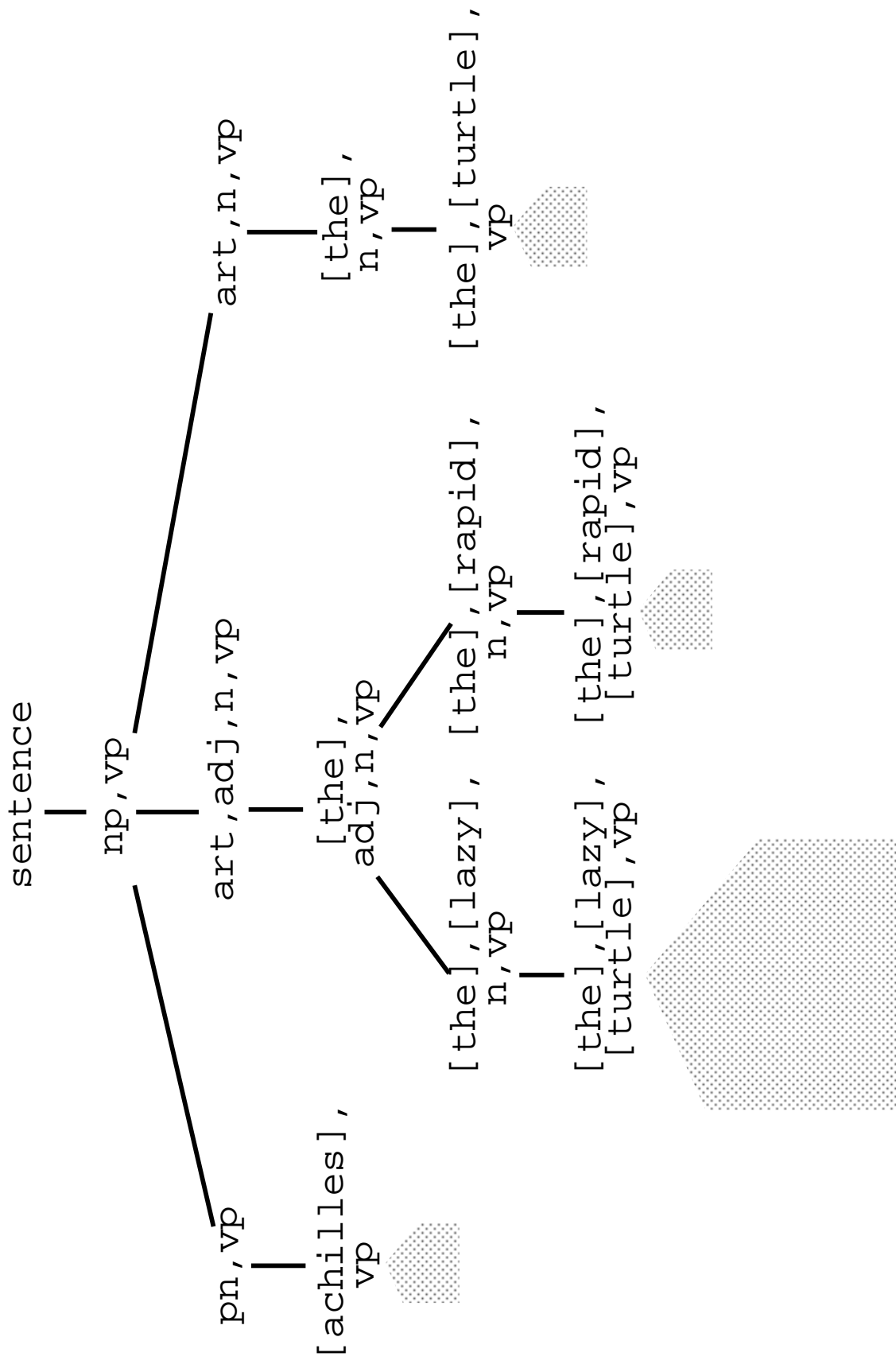
Parse tree

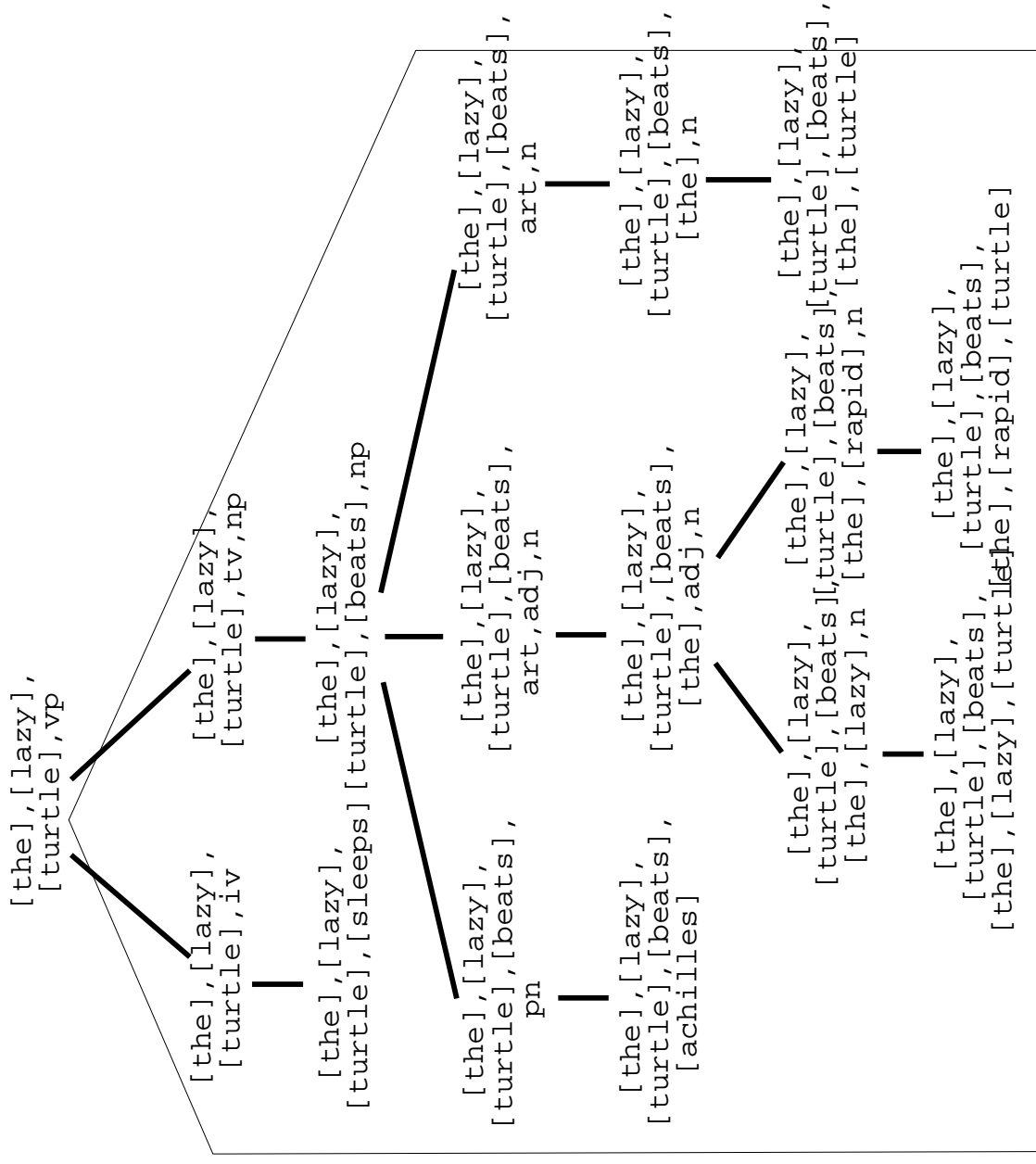
```

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

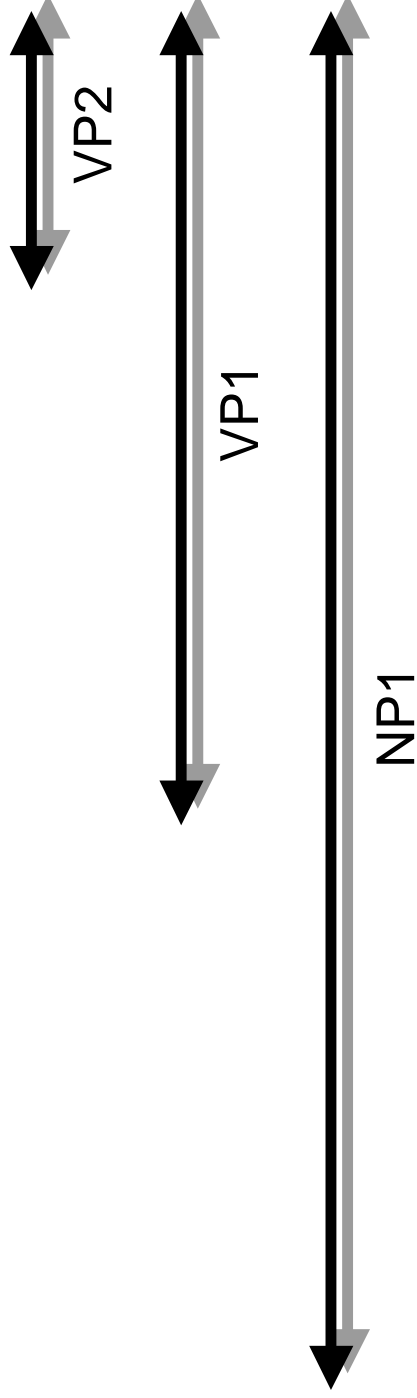
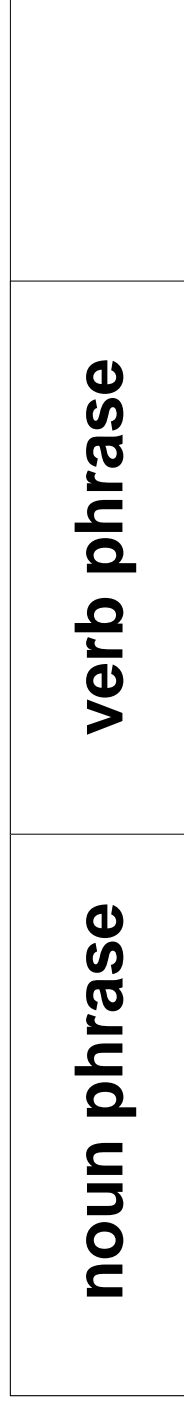




```

sentence --> noun_phrase, verb_phrase
translate as
sentence(S) :-
    noun_phrase(Np),
    verb_phrase(Vp),
    append(Np, Vp, S).
verb -- \sleep].
Translate as
verb([sleep]).
Parsing (answer following query)
:- sentence([the,rapid,turtle,beats,achilles]).

```



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

```

sentence --> noun_phrase, verb_phrase
translate as
sentence(Np1, Vp2) –
    noun_phrase(Np1, Vp1),
    verb_phrase(Vp1, Vp2).
Translate as
verb --> [sleep].
Parsing (answer following query)
:- sentence([the,rapid,turtle,beats,achilles],[ ]).

```

	GRAMMAR	PARSING
META-LEVEL	$s \rightarrow np, vp$?-phrase(s, L)
OBJECT-LEVEL	$s(L, L0) :-$ $np(L, L1),$ $vp(L1, L0)$?-s(L, [])

Meta-level vs. object-level

```
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(plural)   --> [sleep] .
```

Non-terminals with arguments

```

sentence(s(NP, VP))    --> noun_phrase(NP), verb_phrase(
noun_phrase(np(N))    --> proper_noun(N).
noun_phrase(np(Art, Adj, N)) --> article(Art), adjective(Ad
    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)))

```

```

numeral(N)
numeral(N)
n1_999(N)
n1_999(N)
n1_99(N)
n1_99(N)
n0_9(0)
n0_9(N)
n1_9(1)
n1_9(2)
n10_19(10)
n10_19(11)
n20_90(20)
n20_90(30)
...

```

```

--> n1_999(N).
--> n1_9(N1), [thousand], n1_999(N2),
    {N is N1*1000+N2}.
--> n1_99(N).
--> n1_9(N1), [hundred], n1_99(N2),
    {N is N1*100+N2}.
--> n0_9(N).
--> n10_19(N).
--> n20_90(N).
--> n20_90(N1), n1_9(N2), {N is N1+N2}.
--> [].
--> n1_9(N).
--> [one].
--> [two].
...
--> [ten].
--> [eleven].
...
--> [twenty].
--> [thirty].
...

```

?-phrase(numeral(2211),N).

N = [two,thousand,two,hundred,eleven]

Prolog goals in grammar rules

☞ 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)
```


?-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] ;

☞ ‘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)]


```
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 ).
```

☞ Real Grammars

Are much more complicated
Account for detailed syntactic and semantic aspects of language
Take ages to develop
...

☞ Extension for NLP database interface of present grammar is possible

Assert interpreted facts/clauses in Database
Logical reasoning possible
Extend with queries, e.g. What – Who – Where queries
Who is mortal ?
What is Socrates ?
...

Good books on Prolog

Peter Flach, *Simply Logical*, Wiley

Ivan Bratko, *Prolog programming for AI*, Addison Wesley.
Sterling and Shapiro, *The art of Prolog*, MIT Press