Declarative Programming

5: natural language processing using DCGs

Definite clause grammars:
context-free grammars in Prolog

context-sensitive example:

```
noun,singular-->[turtle],singular.
singular,intransitive_verb-->[sleep]
```

one non-terminal on left-hand side

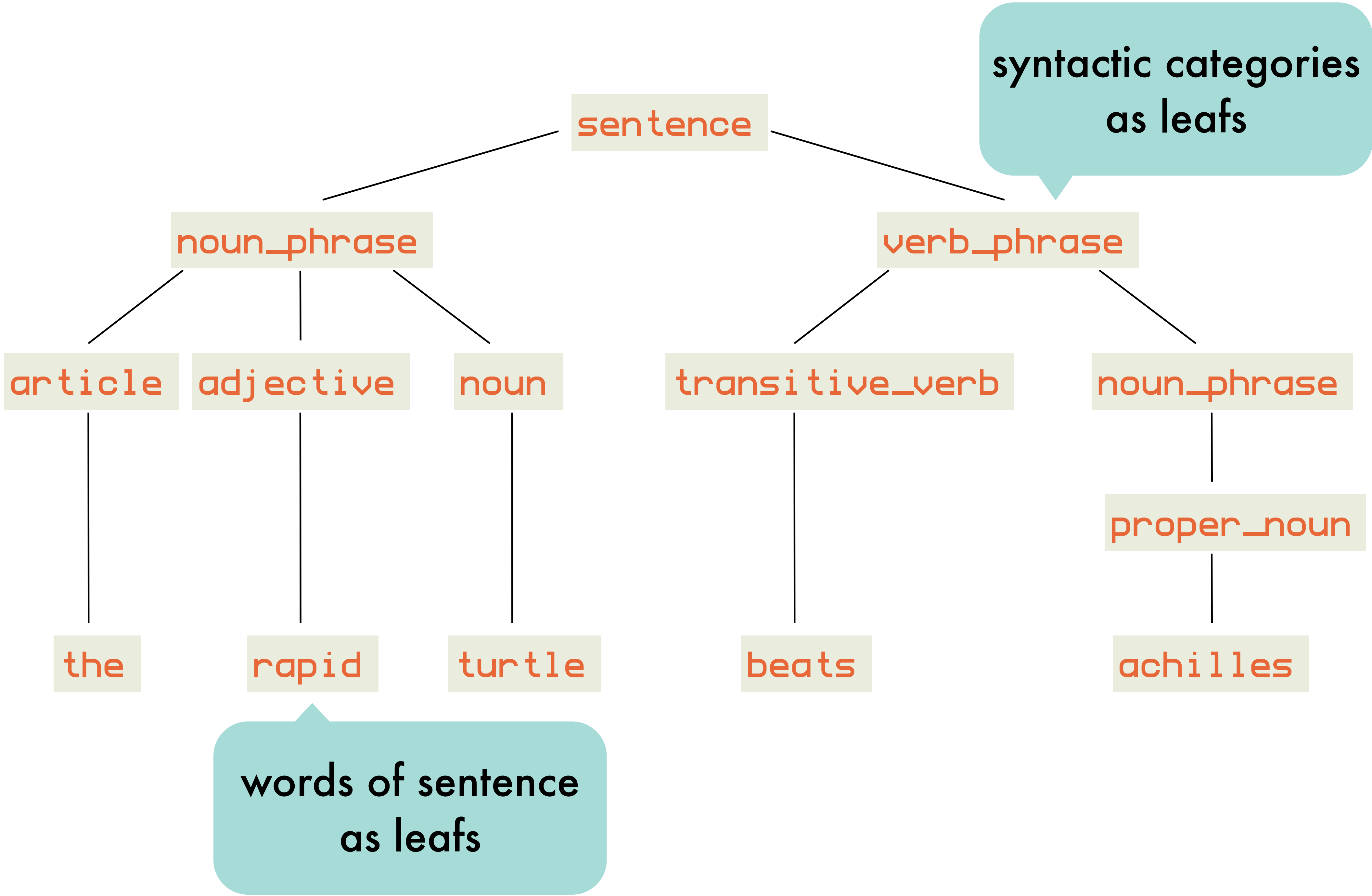
non-terminal defined by rule produces syntactic category

```
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].
```

terminal: word in language

sentences generated by grammar are lists of terminals:
the lazy turtle sleeps, Achilles beats the turtle, the rapid turtle beats Achilles

Definite clause grammars:
parse trees for generated sentences



syntactic categories as leafs

sentence

noun_phrase
verb_phrase

article
adjective
noun
transitive_verb
noun_phrase

proper_noun

the
rapid
turtle
beats
achilles

words of sentence as leafs

Definite clause grammars: *top-down construction of parse trees*

similar to SLD-resolution!

```
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]
```

start with NT and repeatedly replace NTS on right-hand side of an applicable rule until sentence is obtained as a list of terminals

DCG rules and Prolog clauses: *equivalence*

sentence

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

grammar rule

```
sentence --> noun_phrase,
             verb_phrase
```

```
verb--> [sleeps]
```

equivalent Prolog clause

```
sentence(S) :-
  noun_phrase(NP),
  verb_phrase(VP),
  append(NP,VP,S).
```

```
verb([sleeps]).
```

S is a sentence if some first part belongs to the noun_phrase category and some second part to the verb_phrase category

parsing

```
?- sentence([the,rapid,turtle,beats,achilles])
```

DCG rules and Prolog clauses:
built-in equivalence without append/3

meta-level

grammar rule

```
sentence --> noun_phrase,
             verb_phrase
```

object-level

equivalent Prolog clause

```
sentence(L,L0) :-
  noun_phrase(L,L1),
  verb_phrase(L1,L0).
```

L consists of a sentence followed by L0

parsing

```
?- phrase(sentence, L)
```

built-in meta-predicate calling sentence(L,[])

starting non-terminal

DCG rules and Prolog clauses:
summary and expressivity

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

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

non-terminals can have arguments
goals can be put into the rules
no need for deterministic grammars
a single formalism for specifying syntax, semantics
parsing and generating

Expressivity of DCG rules:

non-terminals with arguments - plurality

```
sentence  --> noun_phrase(N),verb_phrase(N).
noun_phrase(N)  --> article(N),noun(N).
verb_phrase(N)  --> 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].
```

arguments unify to express plurality agreement

```
phrase(sentence,[a,turtle,sleeps]). % yes
phrase(sentence,[the,turtles,sleep]). % yes
phrase(sentence,[the,turtles,sleeps]). % no
```

Expressivity of DCG rules:

non-terminals with arguments - 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))))
```

Expressivity of DCG rules:
goals in rule bodies

X_Y(N) if N is a number in [X..Y].

```
numeral(N) --> n1_999(N).
numeralN)  --> 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].
    …
```

regular goal enclosed by braces

```
n1_99(N,L,L0) :-
  n20_90(N1,L,L1),
  n1_9(N2,L1,L0),
  N is N1 + N2.
```

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

Interpretation of natural language:

syntax and semantics

syntax

```
sentence --> determiner, noun, verb_phrase
sentence --> proper_noun, verb_phrase
verb_phrase --> [is], property
property --> [a], noun
property --> [mortal]
determiner --> [every]
proper_noun --> [socrates]
noun --> [human]
```

semantics

```
[every, human, is, mortal]
```

interpret a sentence: assign a clause to it

```
mortal(X):- human(X)
```

represents meaning of sentence

Interpretation of natural language:

interpreting sentences as clauses (I)

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

the meaning of the proper noun ‘Socrates’ is the term socrates

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

operator X=>L: term X is mapped to literal L

the meaning of the property ‘mortal’ is a mapping from terms to literals containing the unary predicate mortal

```
verb_phrase(M) --> [is], property(M).
sentence([(L:-true)]) --> proper_noun(X),
                          verb_phrase(X=>L).
```

singleton clause list, cf. determiner ‘some’

the meaning of a phrase (proper noun - verb) is a clause with empty body and of which the head is obtained by applying the meaning of the verb phrase to the meaning of the proper noun

```
?-phrase(sentence(C),[socrates,is,mortal]).
C = [(mortal(socrates):- true)]
```

Interpretation of natural language:

interpreting sentences as clauses (II)

```
sentence(C) --> determiner(M1,M2,C),
                noun(M1),
                verb_phrase(M2).
noun(X=>human(X)) --> [human].
```

```
determiner(X=>B, X=>H, [(H:- B)]) --> [every].
```

```
?-phrase(sentence(C),[every,human,is,mortal])
C = [(mortal(X):- human(X))]
```

the meaning of a determined sentence with determiner ‘every’ is a clause with the same variable in head and body

Interpretation of natural language:

interpreting sentences as clauses (III)

```
determiner(sk=>H1,sk=>H2,
           [(H1:-true),(H1:-true)] --> [some].
```

```
?-phrase(sentence(C),[some,humans,are,mortal])
C = [(human(sk):-true),(mortal(sk):-true)]
```

the meaning of a determined sentence with determiner ‘some’ are two clauses about the same individual (i.e., skolem constant)

Interpretation of natural language:

relational nature illustrated

```
?-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];
```

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

Interpretation of natural language:

complete grammar with plurality agreement

```
:- op(600,xfy,’=>’).
sentence(C) --> determiner(N,M1,M2,C), noun(N,M1),
verb_phrase(N,M2).
sentence([(L:- true)]) --> proper_noun(N,X),
verb_phrase(N,X=>L).
verb_phrase(s,M) --> [is], property(s,M).
verb_phrase(p,M) --> [are], property(p,M).
property(N,X=>mortal(X)) --> [mortal].
property(s,M) --> noun(s,M).
property(p,M) --> noun(p,M).
determiner(s, X=>B , X=>H, [(H:- B)]) --> [every].
determiner(p, sk=>H1, sk=>H2, [(H1 :- true),(H2 :- true)]) -->[some].
proper_noun(s,socrates) --> [socrates].
noun(s,X=>human(X)) --> [human].
noun(p,X=>human(X)) --> [humans].
noun(s,X=>living_being(X)) --> [living],[being].
noun(p,X=>living_being(X)) --> [living],[beings].
```

Interpretation of natural language:

shell for building up and querying rule base

grammar for queries

```
question(Q) --> [who], [is], property(s,X=>Q)
question(Q) --> [is], proper_noun(N,X), property(N,X=>Q)
question((Q1,Q2)) --> [are], [some], noun(p,sk=>Q1),
                      property(p,sk=>Q2)
```

shell

```
nl_shell(RB) :- get_input(Input), handle_input(Input,RB).

handle_input(stop,RB) :- !.
handle_input(show,RB) :- !, show_rules(RB), nl_shell(RB).
handle_input(Sentence,RB) :- phrase(sentence(Rule),Sentence),
                             nl_shell([Rule|RB]).
handle_input(Question,RB) :- phrase(question(Query),Question),
                             prove_rb(Query,RB),!
                              transform(Query,Clauses),
                             phrase(sentence(Clauses),Answer),
                             show_answer(Answer),
                             nl_shell(RB).
handle_input(Error,RB) :-    show_answer(’no’), nl_shell(RB).
```

add new rule

question that can be solved

transform instantiated query (conjuncted literals) to list of clauses with empty body

generate nl

Interpretation of natural language:

shell for building up and querying rule base - aux

convert rule to natural language sentence

```
show_rules([]).
show_rules([R|Rs]) :-
   phrase(sentence(R),Sentence),
   show_answer(Sentence),
   show_rules(Rs).
get_input(Input) :-
   write(’? ’),read(Input).
   show_answer(Answer) :-
   write(’! ’),write(Answer), nl.
```

```
show_answer(Answer) :- write(‘!’),nl.
```

```
get_input(Input) :- write(‘?’),read(Input).
```

```
transform((A,B),[(A:-true)|Rest]):-!,
  transform(B,Rest).
transform(A,[(A:-true)]).
```

convert query to list of clauses for which natural language sentences can be generated

Interpretation of natural language:

shell for building up and querying rule base - interpreter

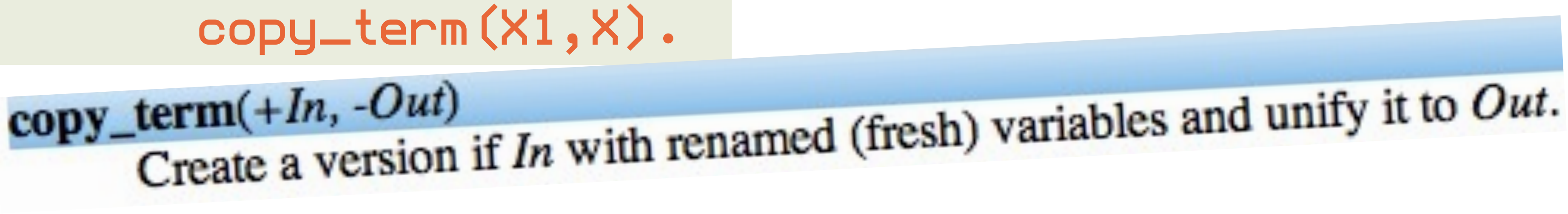
```
prove(true,RB) :- !.
prove((A,B),RB) :- !,
  prove(A,RB),prove(B,RB).
prove(A,RB) :-
  find_clause((A:-B),RB),
  prove(B,RB).

find_clause(C, [R|Rs]) :-
  copy_element(C,R).
find_clause(C, [R|Rs]) :-
  find_clause(C,Rs).

copy_element(X,Ys) :- element(X1,Ys),
                      copy_term(X1,X).
```

handy when storing rule base in list

finds a clause in the rule base, but without instantiating its variables (rule can be used multiple times, rules can share variables)



copy_term(+*In*, -*Out*)
Create a version if *In* with renamed (fresh) variables and unify it to *Out*.

Interpretation of natural language:

shell for building up and querying rule base - example

```
? [every,human,is,mortal]
? [socrates,is,a,human]
? [who,is,mortal]
! [socrates,is,mortal]
? [some,living,beings,are,humans]
? [are,some,living,beings,mortal]
! [some,living,beings,are,mortal]
```

possible improvement: apply idiom of failure-driven loop to avoid memory issues

built-in repeat/1 succeeds indefinitely

```
shell :- repeat, get_input(X), handle_input(X).
handle_input(stop) :- !.
handle_input(X) :- /* handle */, fail.
```

causes backtracking to repeat literal