

Definite Clause Grammars

Contents

- Definite Clause Grammars
- Grammar rules in Prolog
- How Prolog uses grammar rules
- Adding arguments and Prolog goals
- Explanation of Difference Lists
- DCG Recognisers
- Extracting meaning using a DCG
- A Grammar for Extracting Meaning.

Definite Clause Grammars

- DCGs can be used in a variety of applications: besides the analysis and description of language, whether natural or formal, general operations on lists, which are essential in a multitude of programs, can also be described by them.
- The notation of DCGs offers the possibility to express grammars with simple rules. This makes them a good choice for getting into declarative and logical programming.
- If knowledge exists, however, the functionality can be extended by nesting normal Prolog code.

Definite Clause Grammars

In order to parse sentences like:

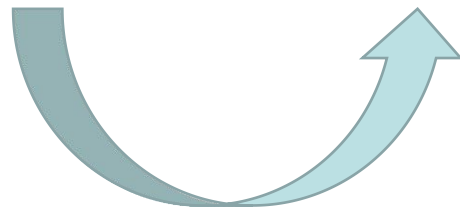
- a cat eats the fish.
- the cat eats a fish.
- a fish eats the cat.
- the fish eat a fish.

Definite Clause Grammars

```
sentence(S1, S3) :-  
    noun_phrase(S1, S2),  
    verb_phrase(S2, S3).  
noun_phrase(S1, S3) :-  
    det(S1, S2),  
    noun(S2, S3).  
verb_phrase(S1, S3) :-  
    verb(S1, S2),  
    noun_phrase(S2, S3).
```

```
det([the|X], X).  
det([a|X], X).  
noun([cat|X], X).  
noun([fish|X], X).  
verb([eats|X], X).
```

```
sentence --> noun_phrase, verb_phrase.  
noun_phrase --> det, noun.  
verb_phrase --> verb, noun_phrase.  
det --> [the].  
det --> [a].  
noun --> [cat].  
noun --> [fish].  
verb --> [eats].
```



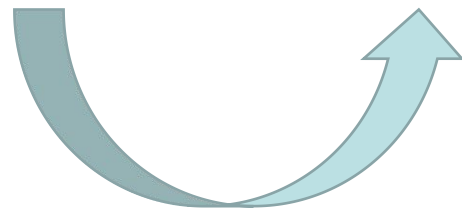
DCG

```
sentence(Sentence) :-
    append(NounPhrase, VerbPhrase, Sentence),
    nounPhrase(NounPhrase),
    verbPhrase(VerbPhrase).

nounPhrase(Phrase) :-
    append(Det, Noun, Phrase),
    det(Det),
    noun(Noun).

verbPhrase(Phrase) :-
    append(Verb, NounPhrase, Phrase),
    verb(Verb),
    nounPhrase(NounPhrase).

det([the]).
det([a]).
noun([cat]).
noun([fish]).
verb([eats]).
```



```
sentence(S1, S3) :-
    noun_phrase(S1, S2),
    verb_phrase(S2, S3).
noun_phrase(S1, S3) :-
    det(S1, S2),
    noun(S2, S3).
verb_phrase(S1, S3) :-
    verb(S1, S2),
    noun_phrase(S2, S3).
det([the|X], X).
det([a|X], X).
noun([cat|X], X).
noun([fish|X], X).
verb([eats|X], X).
```

Difference Lists

Definite Clause Grammars

- A *grammar* is a precise definition of which sequences of words or symbols belong to some *language*.
- Grammars are particularly useful for natural language processing: the computational processing of human languages, like English.
- But they can be used to process any precisely defined 'language', such as the commands allowed in some human-computer interface.
- Prolog provides a notational extension called *DCG (definite Clause Grammar)* that allows the direct implementation of formal grammars.

Grammar rules

- In general, a grammar is defined as a collection of *grammar rules*. These are sometimes called *rewrite rules*, since they show how we can rewrite one thing as something else.
- In linguistics, a typical grammar rule for English might look like this:

sentence → noun_phrase, verb_phrase

e.g. “The man ran.”

- This would show that, in English, a *sentence* could be constructed as a *noun phrase*, followed by a *verb phrase*.
- Other rules would then define how a noun phrase, and a verb phrase, might be constructed. For example:

noun_phrase → noun

noun_phrase → determiner, noun

verb_phrase → intransitive_verb

verb_phrase → transitive_verb, noun_phrase

Terminals and non-terminals

- In these rules, symbols like *sentence*, *noun*, *verb*, etc., are used to show the structure of the language, but they don't go as far down as individual 'words' in the language.
- Such symbols are called *non-terminal symbols*, because we can't stop there.
- In defining grammar rules for *noun*, though, we can say:
 - noun → [ball]
 - noun → [dog]
 - noun → [stick]
 - noun → ['Edinburgh']
- Here, 'ball', 'dog', 'stick' and 'Edinburgh' are words in the language itself.
- These are called the *terminal symbols*, because we can't go any further. They can't be expanded any more.

Grammar rules in Prolog


- Prolog allows us to directly implement grammars of this form.
- In place of the \rightarrow arrow, we have a special operator: `-->`.
- So, we can write the same rules as:

```
sentence    --> noun_phrase, verb_phrase.  
noun_phrase --> noun.  
noun_phrase --> determiner, noun.  
verb_phrase --> intransitive_verb.  
verb_phrase --> transitive_verb, noun_phrase.
```

- Here, each non-terminal symbol is like a predicate with no arguments.
- Terminal symbols are represented as lists containing one atom

```
noun --> [ball].  
noun --> [dog].  
noun --> [stick].  
noun --> ['Edinburgh'].
```

Proper nouns must be written
as strings otherwise they are
interpreted as variables.



How Prolog uses grammar rules

- Prolog converts DCG rules into an internal representation which makes them conventional Prolog clauses.
 - This can be seen by 'listing' the consulted code.
- Non-terminals are given two extra arguments, so:
`sentence --> noun_phrase, verb_phrase.`
becomes: `sentence(In, Out) :-
 noun_phrase(In, Temp),
 verb_phrase(Temp, Out) .`
- This means: some sequence of symbols **In**, can be recognised as a sentence, leaving **Out** as a remainder, if
 - a noun phrase can be found at the start of **In**, leaving **Temp** as a remainder,
 - and a verb phrase can be found at the start of **Temp**, leaving **Out** as a remainder.

How Prolog uses grammar rules (2)

- Terminal symbols are represented using the special predicate 'C', which has three arguments. So:

```
noun --> [ball].
```

becomes: `noun(In, Out) :-`

```
'C'(In, ball, Out).
```

- This means: some sequence of symbols `In` can be recognised as a noun, leaving `Out` as a remainder, if the atom `ball` can be found at the start of that sequence, leaving `Out` as a remainder.
- The built-in predicate 'C' is very simply defined:

```
'C' ( [Term|List], Term, List ).
```

where it succeeds if its second argument is the head of its first argument, and the third argument is the remainder.

A very simple grammar

- Here's a very simple little grammar, which defines a very small subset of English:

```
sentence --> noun, verb_phrase.
```

```
verb_phrase --> verb, noun.
```

```
noun --> [bob].
```

```
noun --> [david].
```

```
noun --> [annie].
```

```
verb --> [likes].
```

```
verb --> [hates].
```

```
verb --> [runs].
```

- We can now use the grammar to test whether some sequence of symbols *belongs to* the language:

```
| ?- sentence([bob, likes, annie], []).
```

```
yes
```

```
| ?- sentence([bob, runs], []).
```

```
no
```

Need to write an extra rule for intransitive verbs.

A very simple grammar (2)

- By specifying that the remainder is an empty list we can use the grammar to generate all of the possible sentences in the language:

```
| ?- sentence(X, []).  
X = [bob,likes,bob] ? ;  
X = [bob,likes,david] ? ;  
X = [bob,likes,annie] ? ;  
X = [bob,hates,bob] ? ;  
X = [bob,hates,david] ? ;
```

This is a *recogniser*. It will tell us whether some sequence of symbols is in a language or not. This has limited usefulness.

- It would be much more useful if we could *do* something with the sequence of symbols, such as converting it into some internal form for processing, or translating it into another language.
- We can do this very powerfully with DCGs, by building a *parser*, rather than a recogniser.

Adding Arguments

- We can add our own arguments to the non-terminals in DCG rules, for whatever reasons we choose.
- As an example, we can very simply add *number* agreement (singular or plural) between the subject of an English sentence and the main verb.

```
sentence --> noun(Num) , verb_phrase(Num) .
verb_phrase(Num) --> verb(Num) , noun(_).
noun(singular) --> [bob] .
noun(plural) --> [students] .
verb(singular) --> [likes] .
verb(plural) --> [like] .
```

- So now:

```
| ?- sentence([bob, likes, students], []).
yes
| ?- sentence([students, likes, bob], []).
no
```

Adding Prolog goals

- If we need to, we can add Prolog goals to any DCG rule.
- They need to be put inside `{ }` brackets, so that Prolog knows they're to be processed as Prolog, and not as part of the DCG itself.
- Let's say that within some grammar, we wanted to be able to say that some symbol had to be an integer between 1 and 100 inclusive. We *could* write a separate rule for each number:

```
num1to100 --> [1].
```

```
num1to100 --> [2].
```

```
num1to100 --> [3].
```

```
num1to100 --> [4].
```

```
...
```

```
num1to100 --> [100].
```

- But using a Prolog goal, there's a much easier way:

```
num1to100 --> [X], {integer(X), X >= 1, X =< 100}.
```


Difference Lists

- We call our grammar with a list of *terminal symbols* and an *empty list* as we are checking that the first list conforms to the grammar with nothing left over.
 - `sentence([the,man,ran],[])`.
- We do this as the Prolog interpreter uses *difference lists* to convert the DCG rules into conventional code.
- The difference list representation is a way of expressing how two lists intersect.
- Any list can be represented as the difference between two lists:

[the,little,blue,man] can be represented as the difference between:

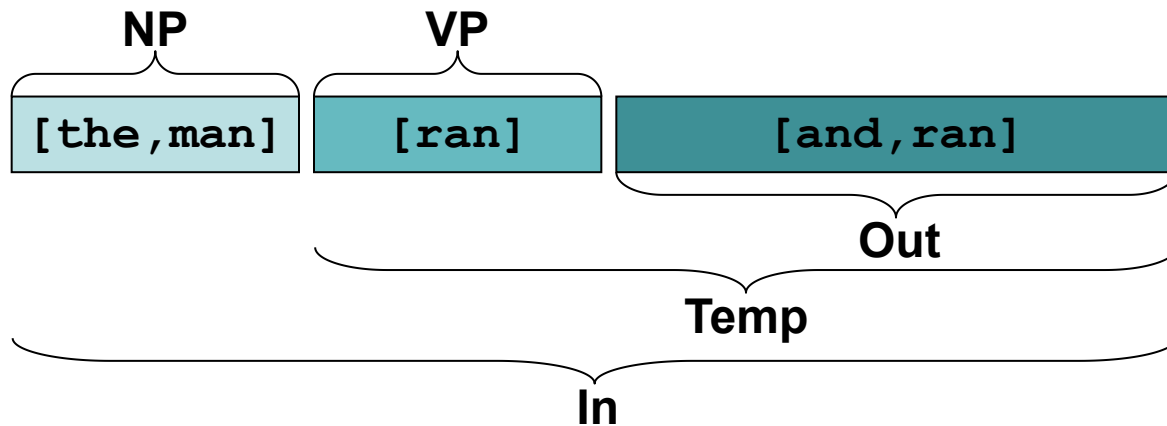
`[the,little,blue,man]-[]`

`[the,little,blue,man,who,swam]-[who,swam]`

`[the,little,blue,man,called,bob]-[called,bob]`

Difference Lists (2)

- The Prolog interpreter converts `sentence --> noun_phrase, verb_phrase.`
- into conventional Prolog code using difference lists that can be read as:
The difference of lists **In** and **Out** is a sentence if the difference between **In** and **Temp** is a noun phrase and the difference between **Temp** and **Out** is a verb phrase.



```
sentence ([the, man, ran, and, ran], [and, ran]) :-  
    noun_phrase ([the, man], [ran, and, ran]),  
    verb_phrase ([ran], [and, ran]).
```

Diff. Lists: An Efficient Append

```
append([], L2, L2) .
```

```
append([H|T], L2, [H|Out]) :-  
    append(T, L2, Out) .
```

- append/2 is a highly inefficient way of combining two lists.

```
?- append([a,b,c], [d], X) .
```

```
append([b,c], [d], X1)      where X1 = [a|X2]  
append([c], [d], X2)       where X2 = [b|X3]  
append([], [d], X3)        where X3 = [c|X4]  
true.                       where X4 = [d]
```

- It must first recurse through the whole of the first list before adding its elements to the front of the second list.
- As the first list increases in length as does the number of recursions needed.
- If we represent the lists as *difference lists* we can append the second list directly to the end of the first list.

Diff. Lists: An Efficient Append (2)

- We can represent any list as the difference between two lists:

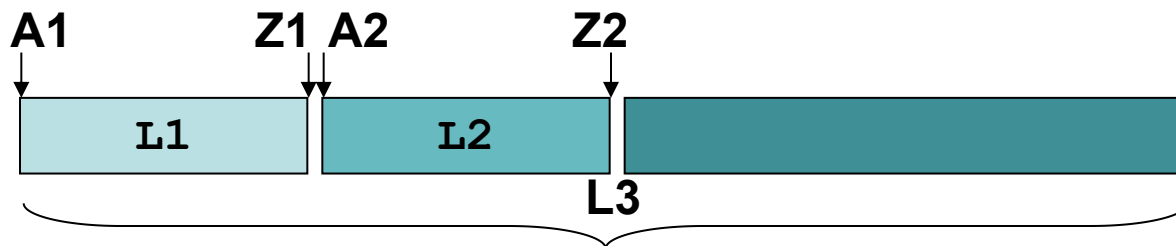
[a,b,c] can be represented as

[a,b,c]-[] or [a,b,c,d,e]-[d,e] or [a,b,c|T]-T

Where 'T' can be any list of symbols.

- As the second member of the pair refers to the end of the list it can be directly accessed.
- This allows us to define a version of append that just uses unification to append two lists L1 and L2 to make L3.

`append(A1-Z1, Z1-Z2, A1-Z2) .`



- When L1 is represented by A1-Z1, and L2 by A2-Z2 the result L3 is A1-Z2 if Z1=A2.

Diff. Lists: An Efficient Append (3)

- If we replace our usual append definition by this one line we can append without recursion.

```
append(A1-Z1, Z1-Z2, A1-Z2) .
```

```
?- append([a,b,c|Z1]-Z1, [d,e|Z2]-Z2, L) .
```

```
L = [a,b,c,d,e|Z2]-Z2,
```

```
Z1 = [d,e|Z2] ?
```

- A clean append can then be achieved by specifying that Z2 is an empty list.

```
| ?- append([a,b,c|Z1]-Z1, [d,e]-[], A1-[]).
```

```
1 Call: append([a,b,c|_506]-_506, [d,e]-[], _608-[]) ?
```

```
1 Exit: append([a,b,c,d,e]-[d,e], [d,e] [], [a,b,c,d,e]-[]) ?
```

```
A1 = [a,b,c,d,e],
```

```
Z1 = [d,e] ?
```

```
yes
```

Definite Clause Grammars Summary

- We can use the `-->` DCG operator in Prolog to define grammars for any language.
- The grammar rules consist of *non-terminal symbols* (e.g. NP, VP) which define the structure of the language and *terminal symbols* (e.g. Noun, Verb) which are the words in our language.
- The Prolog interpreter converts the DCG notation into conventional Prolog code using *difference lists*.
- We can add *arguments* to non-terminal symbols in our grammar for any reason (e.g. number agreement).
- We can also add pure Prolog code to the right-hand side of a DCG rule by enclosing it in `{ }`.

DCG Recognisers

- We can write simple grammars using the DCG notation that *recognise* if a string of words (represented as a list of atoms) belongs to the language.

```
sentence --> noun, verb_phrase.
```

```
verb_phrase --> verb, noun.
```

```
noun --> [bob].
```

```
noun --> [david].
```

```
noun --> [annie].
```

```
verb --> [likes].
```

```
verb --> [hates].
```

```
verb --> [runs].
```

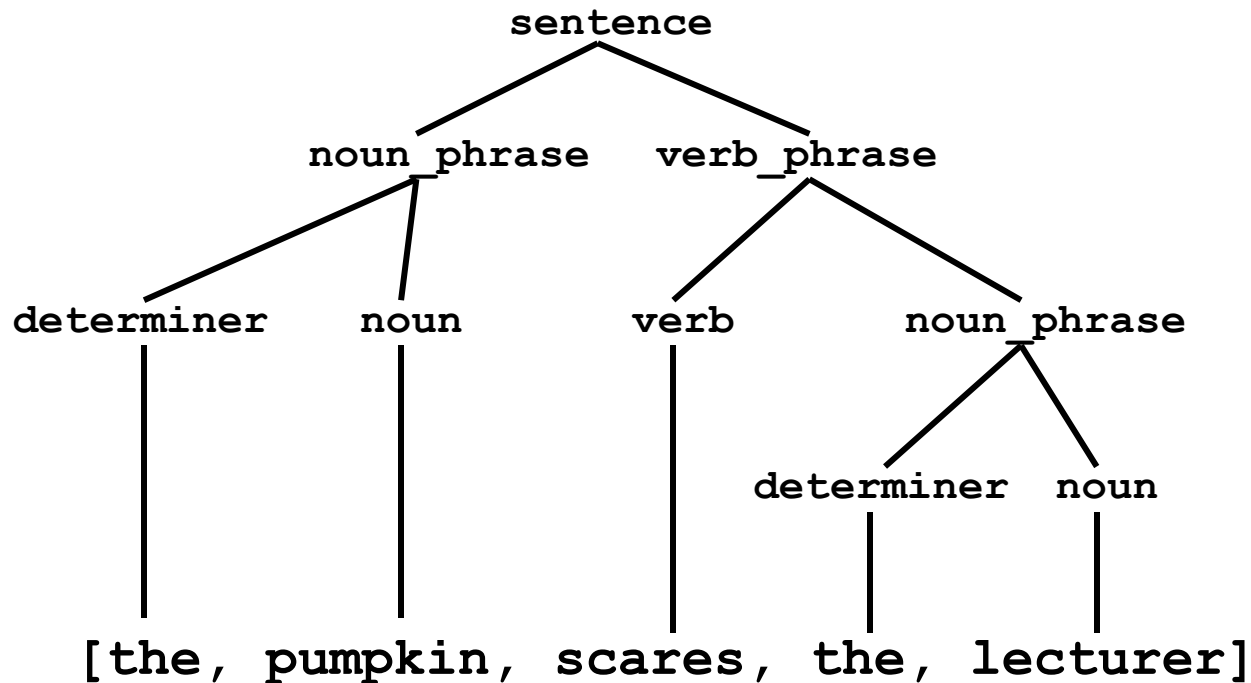
```
|?- sentence([annie, hates, david], []).
```

```
yes
```

However, this is of limited usefulness. Ideally we would like to *interpret* the input in some way: to understand it, parse it, or convert it into some other more useful form.

DCG: Parsers

- A parser *represents* a string as some kind of structure that can be used to understand the role of each of its elements.
- A common representation is a *parse tree* which shows how input breaks down into its grammatical constituents.



Two parsing techniques

There are generally two ways of using DCGs to build a structural representation of the input.

1. Computing the structure once the constituents of the input have been identified.

- Partial results can be passed via extra arguments in non-terminal symbols and computed to create a suitably representative result.
- For example, we might want our DCG to represent a number expressed as a string as an integer.

```
number(N) --> digit(D), [hundred], {N is (D * 100)}.
```

```
digit(1) --> [one].
```

```
|?- number(X, [one, hundred], []).
```

```
X = 100?
```

```
yes
```

This is only good for summary representations; it doesn't tell us anything about the internal structure of our input.

Two parsing techniques (2)

2. The more popular method is to use *unification* to identify the grammatical role of each word and show how they combine into larger grammatical structures.

- This creates a representation similar to a parse tree.

sentence (s (NP , VP)) -->

noun_phrase (NP) , verb_phrase (VP) .

- Which can be read as:

The parsed structure of a **sentence** must be **s(NP,VP)**,

where **NP** is the parsed structure of the **noun phrase**, and

VP is the parsed structure of the **verb phrase**.

- The rules for NPs and VPs would then need to be augmented so that they also represent a parse of their constituents in the head of the rule.

Example: parsing English

- So lets take a small grammar which defines a tiny fragment of the English language and add arguments so that it can produce a parse of the input.

Original grammar rules:

```
sentence --> noun_phrase (Num) , verb_phrase (Num) .
```

```
noun_phrase (Num) --> determiner (Num) , noun_phrase2 (Num) .
```

```
noun_phrase (Num) --> noun_phrase2 (Num) .
```

```
noun_phrase2 (Num) --> adjective , noun_phrase2 (Num) .
```

```
noun_phrase2 (Num) --> noun (Num) .
```

```
verb_phrase (Num) --> verb (Num) .
```

```
verb_phrase (Num) --> verb (Num) , noun_phrase ( _ ) .
```

- Note the use of an argument to enforce number agreement between noun phrases and verb phrases.

Example: parsing English (2)

- Now we can add a new argument to each non-terminal to represent its structure.

```
sentence (s (NP, VP)) -->
```

```
    noun_phrase (NP, Num), verb_phrase (VP, Num) .
```

```
noun_phrase (np (DET, NP2), Num) -->
```

```
    determiner (DET, Num), noun_phrase2 (NP2, Num) .
```

```
noun_phrase (np (NP2), Num) -->
```

```
    noun_phrase2 (NP2, Num) .
```

```
noun_phrase2 (np2 (N), Num) --> noun (N, Num) .
```

```
noun_phrase2 (np2 (ADJ, NP2), Num) -->
```

```
    adjective (ADJ), noun_phrase2 (NP2, Num) .
```

```
verb_phrase (vp (V), Num) --> verb (V, Num) .
```

```
verb_phrase (vp (V, NP), Num) -->
```

```
    verb (V, Num), noun_phrase (NP, _) .
```

Example: parsing English (3)

- We also need to add extra arguments to the terminal symbols i.e. the lexicon.

```
determiner(det(the), _) --> [the].
```

```
determiner(det(a), singular) --> [a].
```

```
noun(n(pumpkin), singular) --> [pumpkin].
```

```
noun(n(pumpkins), plural) --> [pumpkins].
```

```
noun(n(lecturer), singular) --> [lecturer].
```

```
noun(n(lecturers), plural) --> [lecturers].
```

```
adjective(adj(possessed)) --> [possessed].
```

```
verb(v(scared), singular) --> [scared].
```

```
verb(v(scare), plural) --> [scare].
```

- We represent the terminal symbols as the actual word from the language and its grammatical role. The rest of the grammatical structure is then built around these terminal symbols.

Using the parser.

- Now as a consequence of recognising the input, the grammar constructs a term representing the constituent structure of the sentence.
- This term is the 1st argument of `sentence/3` with the 2nd argument the input list and the 3rd the remainder list (usually []).

```
|?-sentence(Struct,[the, pumpkin, scares, the, lecturer],[ ]).  
Struct = s( np( det(the), np2(n(pumpkin)) ),  
            vp( v(scares), np(det(the), np2(n(lecturer))) ) ) ?
```

- We can now generate all valid sentences and their structures by making the 2nd argument a variable.

```
|?-sentence(X,Y,[ ]).  
X = s(np(det(the), np2(adj(possessed), np2(n(lecturer)))) ,  
      vp(v(scares), np(det(the), np2(n(pumpkin)))) ) ,  
Y = [the, possessed, lecturer, scares, the, pumpkin] ? ;  
.....etc
```

Extracting meaning using a DCG

- Representing the structure of a sentence allows us to see the beginnings of semantic relationships between words.
- Ideally we would like to take these relationships and represent them in a way that could be used computationally.
- A common use of meaning extraction is as a natural language interface for a database. The database can then be questioned directly and the question converted into the appropriate internal representation.
- One widely used representation is Logic as it can express subtle semantic distinctions:
 - e.g. “Every man loves a woman.” vs. “A man loves every woman.”
- Therefore, the logical structures of Prolog can also be used to represent the meaning of a natural language sentence.

Logical Relationships

- Whenever we are programming in Prolog we are representing meaning as logical relationships:
 - e.g. “John paints.” = `paints(john)`.
 - e.g. “John likes Annie” = `likes(john,annie)`.
- It is usually our job to make the conversion between natural language and Prolog but it would be very useful if a DCG could do it for us.
- To do this we need to add Prolog representations of meaning (e.g. `paints(john)`) to the non-terminal heads of our grammar.
 - Just as we added parse structures to our previous grammar,
e.g. `sentence(s(NP,VP)) --> noun_phrase(NP,Num),
verb_phrase(VP,Num) .`
 - We can construct predicates that represent the relationship between the terminal symbols of our language:
e.g. `intrans_verb(Actor,paints(Actor)) --> [paints]`

Adding meaning to a simple grammar

- Here is a simple DCG to recognise these sentences:

```
sentence --> noun_phrase, verb_phrase
noun_phrase --> proper_noun.
verb_phrase --> intrans_verb.
verb_phrase --> trans_verb, noun_phrase.
intrans_verb --> [paints].
trans_verb --> [likes].
proper_noun --> [john].
proper_noun --> [annie].
```

```
| ?- sentence([john,likes,annie], []).
```

yes

- To encode meaning we first need to represent nouns as atoms. Prolog atoms are existential statements

e.g. `john` = “There exists an entity ‘john’ ”.

```
proper_noun(john) --> [john].
proper_noun(annie) --> [annie].
```

Adding meaning to a simple grammar (2)

- Now we need to represent the meaning of verbs.
- This is more difficult as their meaning is defined by their context i.e. a noun phrase.
- We can represent this in Prolog as a property with a variable entity. For example, the intransitive verb 'paints' needs an NP as its actor: "Somebody paints" = `paints(Somebody)`.
- We now need to ensure that this variable 'somebody' is matched with the NP that precedes the VP.
- To do this we need to make the argument of the Prolog term ('somebody') *visible* from outside of the term.
- We do this by adding another argument to the head of the rule.
e.g `intrans_verb(Somebody,paints(Somebody)) --> [paints]`.

Adding meaning to a simple grammar (3)

- Now we need to ensure that this variable gets matched to the NP at the sentence level.
- First the variable needs to be passed to the parent VP:
`verb_phrase(Actor,VP) --> intrans_verb(Actor,VP) .`
- The Actor variable must then be linked to the NP at the sentence level:
`sentence(VP) --> noun_phrase(Actor) , verb_phrase(Actor,VP) .`
- It now relates directly to the meaning derived from the NP.
- The logical structure of the VP is then passed back to the user as an extra argument in `sentence`.
- If the grammar is more complex then the structure returned to the user might be the product of more than just the VP.

Adding meaning to a simple grammar (4)

- Lastly, we need to define the transitive verb.
- This needs two arguments, a Subject and an Object.

```
trans_verb(Subject, Object, likes(Subject, Object)) --> [likes].
```

- The `subject` needs to be bound to the initial NP and the `object` to the NP that is part of the VP.

```
verb_phrase(Subject, VP) --> trans_verb(Subject, Object, VP),  
                                noun_phrase(Object).
```

- This binds the `subject` to the initial NP at the sentence level as it appears in the right position the `verb_phrase` head.

A Grammar for Extracting Meaning.

- Now we have a grammar that can extract the meaning of a sentence.

```
sentence (VP) --> noun_phrase (Actor) ,  
verb_phrase (Actor, VP) .
```

```
noun_phrase (NP) --> proper_noun (NP) .
```

```
verb_phrase (Actor, VP) --> intrans_verb (Actor, VP) .
```

```
verb_phrase (Actor, VP) --> trans_verb (Actor, Y, VP) ,  
                                noun_phrase (Y) .
```

```
intrans_verb (Actor, paints (Actor)) --> [paints] .
```

```
trans_verb (X, Y, likes (X, Y)) --> [likes] .
```

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

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

A Grammar for Extracting Meaning.

- The meaning of specific sentences can be extracted:

```
| ?- sentence(X, [john, likes, annie], []).  
X = likes(john, annie) ? ;  
no
```

- Or, all possible meanings can be generated:

```
| ?- sentence(X, Y, []).  
  
X = paints(john),  
Y = [john, paints] ? ;  
  
X = likes(john, john),  
Y = [john, likes, john] ? ;  
  
X = likes(john, annie),  
Y = [john, likes, annie] ? ;  
  
X = paints(annie),  
Y = [annie, paints] ? ;  
  
X = likes(annie, john),  
Y = [annie, likes, john] ? ;  
  
X = likes(annie, annie),  
Y = [annie, likes, annie] ? ;  
  
no
```

Extending the meaning

- By writing grammars that accept
 - conjunctions (e.g. ‘and’),
 - relative clauses (e.g. ‘that snores’ in ‘The man that snores’),
 - and conditionals (e.g. ‘I am blue *if I am a dolphin*’)
- all forms of logical relationship in Prolog can be extracted from strings of natural language.