

Grammars in Prolog

The nature of Prolog is such that it lends itself very well to grammar development

A *grammar* is the set of rules governing how a language (any language) is used...

...rules you say? Sounds like we're speaking in Prolog's terms already!

Every programming language has a grammar (think of compilers)...as does every natural language

NLP in Prolog

NLP stands for **Natural Language Processing**...

Natural in the sense of English, Japanese, Spanish, Latin...etc

I.e. not programming languages

NLP is the use of computers to understand human languages...

...that is can respond in appropriate ways to language queries, conversation, etc.

NLP in Prolog

Why do we want to do that?

- English as a command language
- Natural-language query interfaces (HCI/UI)
- Machine translation
- Information extraction
- Speech recognition

Early NLP

SHRDLU

“blocks world”

ELIZA

...led many to believe the problem was far simpler than it in fact is...

Why is it hard?

“I took the bus because it was raining”

“I took the bus because it was waiting”

Why is it hard?

“Time flies like an arrow”

You mean...

1. Time moves quickly, like an arrow?
2. Measure the speed of flies, as you would measure the speed of an arrow? (As in “to time” something)
3. Measure the speed of flies in the same fashion as an arrow (i.e. time them in the same way that an arrow would do the job)
4. Measure the speed of insects that are like arrows
5. A special type of insect, time flies, like arrows (fruit flies like a banana)

You should be thinking... UGH

Why is it hard?

Speech segmentation

“Speechisreallyjustlikethis”

Text segmentation

Chinese, Thai, e.g. do not have single-word boundaries

Word sense disambiguation

“Bank”?

Syntactic ambiguity

Imperfect or irregular input

Speech acts/pragmatics

“mean what you say/ say what you mean”

Two Kinds...

Statistical NLP (aka Stochastic NLP)

Uses statistical and probabilistic techniques to process and understand language

Rule-based NLP (aka non-stochastic)

Uses generally hand-coded rules to process and understand language

Two Kinds...

Rule-based NLP (aka non-stochastic)

Uses generally hand-coded rules to process and understand language

We're just going to concern ourselves with the latter (though there are arguments for and against both)

This will make a nice segue back into Prolog...

NLP in Prolog

What do we need?

- A complete description of the grammar

The rest is easy! (Well sorta)

NLP in Prolog

What do we need?

- *A complete description of the grammar*

But there is no “off-the-shelf” copy of English grammar, e.g., available. Or any other language for that matter.

The attempt to find such an entity is a large part of the discipline of *linguistics*...

...it also looks at language as a way to understand the brain (thus NLP can also help gather insight)

NLP in Prolog

As an aside, NLP is often equated with CL (computational linguistics)... in fact there is a difference in the motivations.

NLP seeks to describe languages via computers for the purposes of better computer applications

CL seeks to describe languages via computers for the purposes of a deeper understanding of linguistic theories, and theories of the brain

NLP in Prolog

Sadly, there is no time to delve further into these details (though if you catch me in the halls, I love to chat about this stuff)

For the purposes of this class, I'm just going to get you started with some basic NLP

We'll start with a whirlwind tour of linguistics, then delve right into NLP... hopefully enough to get you started with a basic grammar/parser.

NLP in Prolog

So if a grammar is the rules that determine what is legit in a language, a parser is the tool whereby we apply or enforce that grammar.

You're going to learn to write a parser in Prolog...

Are you excited?

You should be!

Levels of Linguistic Analysis

Phonology

how sounds are used

Morphology/Lexical

word formation and words

Syntax

sentence construction

Semantics

meaning -- connecting to the real world

Pragmatics

language in context... fuzzy boundary with semantics sometimes

Why Prolog?

One of the most appropriate for NLP.. Why?

1. Large, complex data structures are easy to build and modify e.g. syntactic and semantic structures
2. The program can examine and modify itself... can be very abstract in programming
3. Designed for knowledge representation (KR) and is build around FOL
4. Depth-first search is built in, and easily used in parsers (actually, Prolog *has* a built in parser!)
5. Unification can be used to build data structures step by step

On the Agenda

Templates and keywords

DCGs

Top-down vs bottom-up parsing

English phrase structures

A (very) brief look at semantics

Templates and Keywords

ELIZA is a good example of using templates and keywords

Demo...

Templates and Keywords

A template is a pattern such as

I like X

What ELIZA showed was that templates were powerful enough to convincingly simulate realistic natural-language dialogue

Templates, however, are highly *restrictive* and require much simplification.

Instead we are going to learn about DCGs...

Questions?

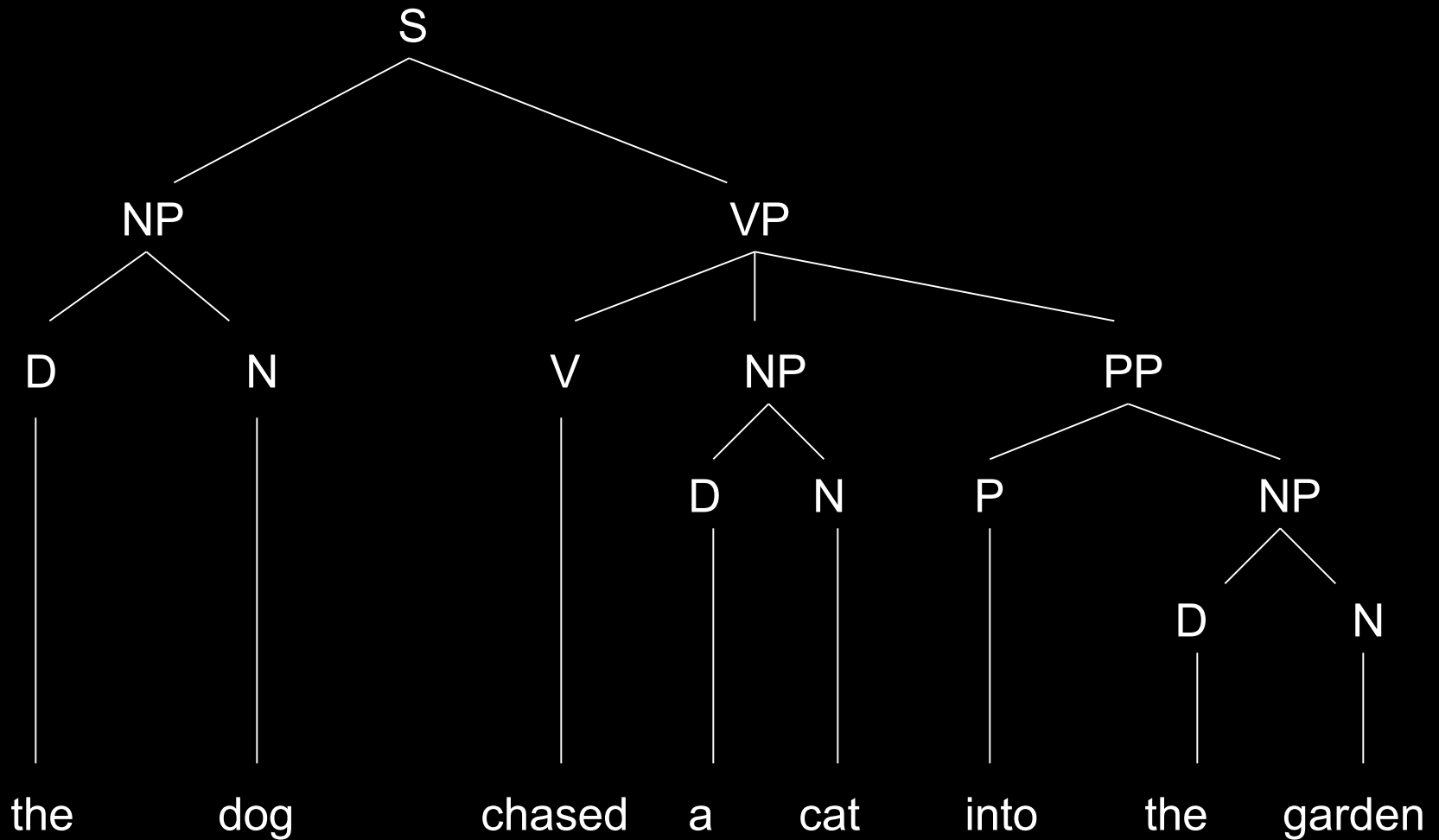
Definite Clause Grammars

Templates and keywords ignore the *constituent structure* of natural language

A sentence is not just a string of unrelated words, but rather consists of words which can be grouped into meaningful units: *phrases*

This is often noted through a *tree*

Definite Clause Grammars



Definite Clause Grammars

Based upon this tree, we can establish the following rules:


S --> NP VP
NP --> D N
VP --> V NP
VP --> V NP PP
PP --> P NP
D --> the
D --> a
N --> dog
N --> cat
N --> garden
V --> chased
V --> saw
P --> into

These are known
as **Phrase Structure
Rules** (or PS rules for
short)

Definite Clause Grammars

Based upon this tree, we can establish the following rules:

These
symbols
on the
LHS are
called
*non-
terminals*



- S --> NP VP
- NP --> D N
- VP --> V NP
- VP --> V NP PP
- PP --> P NP
- D --> the
- D --> a
- N --> dog
- N --> cat
- N --> garden
- V --> chased
- V --> saw
- P --> into

Definite Clause Grammars

Based upon this tree, we can establish the following rules:

S --> NP VP

NP --> D N

VP --> V NP

VP --> V NP PP

PP --> P NP

D --> the

D --> a

N --> dog

N --> cat

N --> garden

V --> chased

V --> saw

P --> into

These
symbols
on the
RHS are
called
terminals

Definite Clause Grammars

A symbol can expand to nothing or null:

S --> NP VP

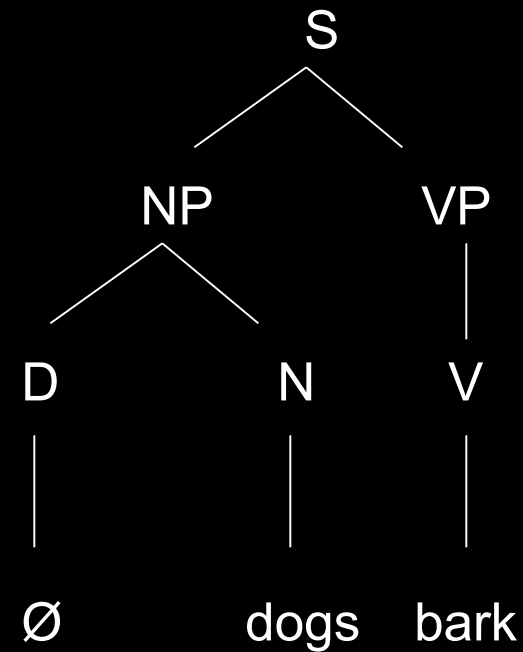
NP --> D N

VP --> V NP

D --> \emptyset

N --> dogs

V --> bark



Definite Clause Grammars

These rules are known as *context-free*

That is, there is no restriction on what rule can apply on the basis of context

Thus, the rule $NP \rightarrow D N$ says that *any* NP can consist of a D and a N, regardless of where it occurs.

Definite Clause Grammars

A context-sensitive rule might say that a that
“NP expands to a D and a N, but only
when preceded by a P”

Context sensitivity adds a great deal of
complexity in NLP and is not used often
(we will focus on context-free rules)

Definite Clause Grammars

By writing rules as constituent rules, we also can make use of recursion:

The dog chased the cat.

The girl thought the dog chased the cat.

The butler said the girl thought the dog chased the cat.

The gardener claimed the butler said the girl thought the dog chased the cat.

Definite Clause Grammars

We just need to add the following to our existing grammar:

VP \rightarrow V S

N \rightarrow gardener

N \rightarrow girl

N \rightarrow butler

V \rightarrow claimed

V \rightarrow thought

V \rightarrow said

Definite Clause Grammars

We just need to add the following to our existing grammar:

S --> NP VP

NP --> D N

VP --> V NP

VP --> V NP PP

PP --> P NP

D --> the

D --> a

N --> dog

N --> cat

N --> garden

V --> chased

V --> saw

P --> into

VP --> V S

N --> gardener

N --> girl

N --> butler

V --> claimed

V --> thought

V --> said

A Parsing Algorithm

A parser is an algorithm process for determining if a sentence is generated by a particular grammar

As a side-effect, parsers can also report the structure assigned to a sentence by a grammar (we're not going to worry about that yet)

A Parsing Algorithm

There are 2 major ways to parse...

Bottom Up Parsing (BUP)

looks at the words (terminals) and tries to combine them to form larger constituents

E.g. *The* is a D
 Dog is an N
 D and N together make an NP
 ...

A Parsing Algorithm

There are 2 major ways to parse...

Top Down Parsing

looks for a specific constituent, such as an S,
and uses the rules to find the sub-parts

E.g. S consists of NP and VP

 NP consists of D and N

The is a D

Dog is an N...

A Parsing Algorithm

Both BUP and Top-Down Parsers need to backtrack

Such a parser is called a “non-deterministic” parser (we’ve had the deterministic/non-deterministic chat already)

We’re going to focus on top-down for now (an example of a BUP parser would be a shift-reduce parser)

A Parsing Algorithm

Parsing TD in Prolog is just like satisfying Prolog queries...

We know that S is true, if NP and VP are true, etc.

All we need to do is rewrite our grammar as a set of Prolog clauses...

...such a grammar is called a “definite clause grammar” or DCG

(Definite clauses are nothing fancy -- just Prolog rules and facts.)

A Parsing Algorithm

Sounds good..

So how do we do that?

A Parsing Algorithm

Let's recall just one grammar rule for now:

$$S \rightarrow NP VP$$

A Parsing Algorithm

Let's recall just one grammar rule for now:

$$S \rightarrow NP VP$$

In a DCG this is translated into:

$$s(L1,L):- np(L1,L2), vp(L2,L)$$

A Parsing Algorithm

In English...

$s(L1, L) :- np(L1, L2), vp(L2, L)$

...means:

“To parse an S starting with input string L1 and ending with L, first parse an NP starting with L1 and ending with L2, then parse a VP starting at L2 (where the NP ended) and ending at L”

A Parsing Algorithm

Non-terminals then look like the following:

$n([\text{dog}|L], L).$

A Parsing Algorithm

Non-terminals then look like the following:

$n([dog|L],L).$

In English:

“To parse a noun, remove *dog* from the beginning of the input string”

A Parsing Algorithm

s(L1,L):- np(L1,L2), vp(L2,L).

np(L1,L):- d(L1,L2), n(L2,L).

vp(L1,L):- v(L1,L2), np(L2,L).

d([the|L],L).

d([a|L],L).

n([dog|L],L).

n([cat|L],L).

n([gardener|L],L).

n([butler|L],L).

v([chased|L],L).

v([saw|L],L).

Here's our
grammar in
Prolog

A Parsing Algorithm

It just so happens that this sort of parsing is so common that Prolog has a built-in facility for it

That is, if you write your rules in DCG notation, Prolog will automatically convert them into PS rules

A Parsing Algorithm

DCG rules look like this:

```
s --> np, vp.  
np --> d, n.  
n --> [dog].
```

Upon being read-in via consult/reconsult, they are automatically converted into:

```
s(L1,L) :- np(L1,L2), vp(L2,L).  
np(L1,L) :- d(L1,L2), n(L2,L).  
n([dog|L],L).
```

A Parsing Algorithm

In other words, Prolog will track those pesky lists for you.

Once read into memory, Prolog forgets that the rules were ever expressed in DCG notation...

...thus if you type `listing np/2`, the rule `np` will appear as the translated rule, not the DCG version that you supplied

A Parsing Algorithm

Every DCG rule takes the form:

nonterminal symbol --> expansion

Where *expansion* is one of

- A nonterminal symbol such as np;
- A list of terminal symbols such as [dog] or [the,dogs];
- A null constituent represented by [];
- A plain Prolog goal enclosed in braces, such as { write('Found NP') };
- A series of any of these expressions joined by commas

A Parsing Algorithm

Note that Prolog's built in translator works by adding two arguments to every symbol that is not in brackets or braces.

It also takes care of the order, and chaining them together

i.e. $\text{np}(L, L2) :- \text{d}(L, L1), \text{n}(L1, L2)$

A Parsing Algorithm

These internal variables are **not** accessible to you, however, when using DCG notation:

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

In fact, you cannot access L2... at this point in time the indexing variables do not exist, and even upon translation, Prolog will not know what you are referring to as it will use its own internal names for those variables.

Left Recursion Again

Remember this example?

$$np \rightarrow d, n$$
$$np \rightarrow np, conj, np$$

Recall that this will loop infinitely. However, we can come up with a contrived definition that will prevent this:

$$np \rightarrow np_x, conj, np$$
$$np \rightarrow np_x$$
$$np_x \rightarrow d, n$$

Questions?

Next...

We're going to explore some of things you
can do with DCGs...

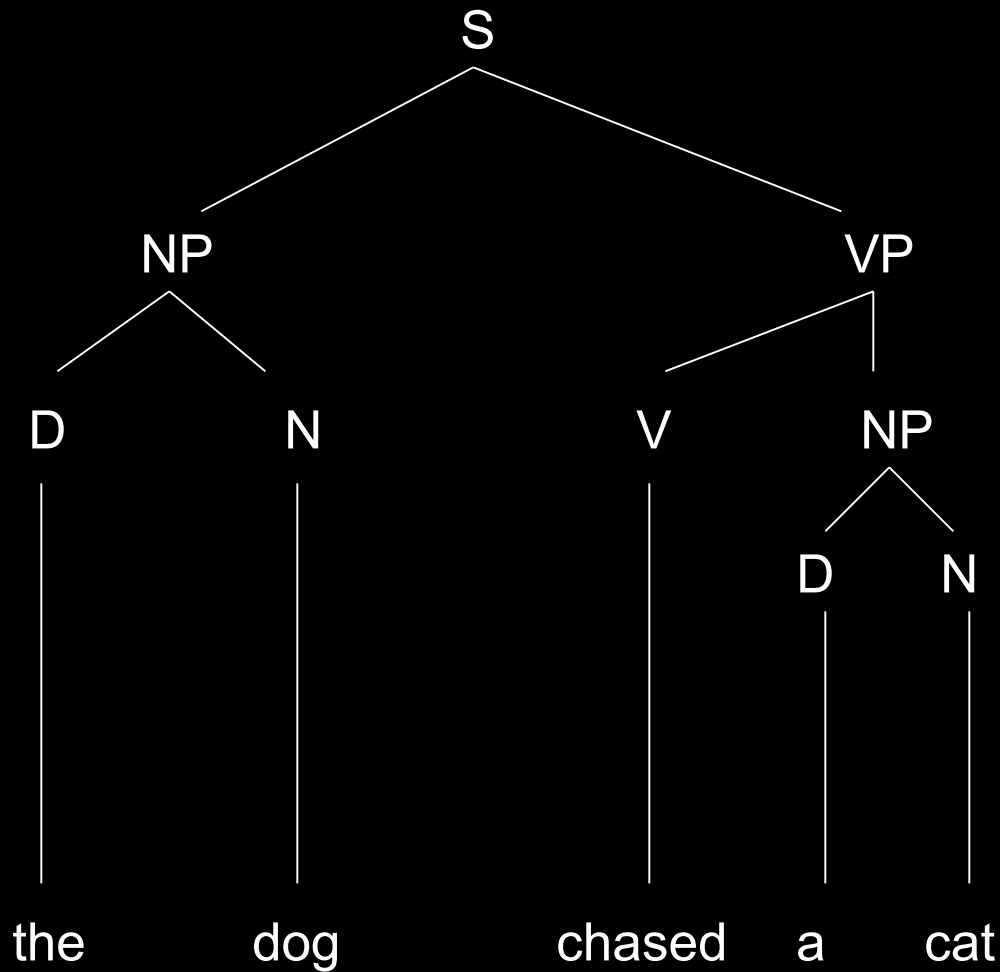
Building Syntactic Trees

It turns out that we can build syntactic trees as a side-effect of the parsing process.

These trees will give us information about the structure of the sentence/phrase which is accepted by our grammar.

We've already seen one example of a tree...

Building Syntactic Trees



Building Syntactic Trees

Such a tree can be succinctly represented as the following:

```
s(np(d(the),n(dog)),vp(v(chased),np(d(a),n(cat))))
```

We can get Prolog to build this representation for us as it parses...

Each rule is responsible for its own part of the structure...

Building Syntactic Trees

This rule:

$$s \rightarrow np, vp$$

Is responsible for:

$$s (... , ...)$$

(where “...” are np and vp structures, respectively)

Building Syntactic Trees

This rule:

$np \rightarrow d, n$

Is responsible for:

$np (... , ...)$

And so on, till the terminals...they are responsible
for the actual words...

Building Syntactic Trees

This rule:

$n \rightarrow [dog]; [cat]$

Is responsible for:

$n (...)$

Where “...” is an actual word from the lexicon

Building Syntactic Trees

To implement this, recall that DCGs in Prolog allow for extra arguments:

So...

```
s(a,b) --> np(c,d), vp(e,f) .
```

becomes...

```
s(a,b,L1,L) :- np(c,d,L1,L2), vp(e,f,L2,L) .
```

Building Syntactic Trees

Thus, we get the following:

`s(s(NP,VP)) --> np(NP), vp(VP).`

`np(np(D,N)) --> d(D), n(N).`

`vp(vp(V,NP)) --> v(V), np(NP).`

`d(d(the)) --> [the].`

`n(n(dog)) --> [dog].`

`n(n(cat)) --> [cat].`

`v(v(chased)) --> [chased].`

Unification and instantiation gives us a way to work with information we do not yet have... this gives Prolog much of its power!

Agreement in English

In English, a verb and its subjects must agree in *number*...

In other words, they are either both *singular* or both *plural*

The dog chases the cats (singular subject; singular verb)

The dogs chase the cats (plural subject; plural verb)

**The dog chase the cats* (singular subject; plural verb)

**The dogs chases the cats* (plural subject; singular verb)

Agreement in English

In DCGs...

```
n(singular) --> [dog];[cat];[mouse]
```

```
n(plural) --> [dogs];[cats];[mice]
```

```
v(singular) --> [chases];[sees]
```

```
v(plural) --> [chase];[see]
```

```
np(Number) --> d, n(Number) .
```

```
vp(Number) --> v(Number), np(_) .
```

```
s --> np(Number), vp(Number) .
```

Case Marking

Related to the problem of agreement... some English pronouns are marked for *case*

That is, a pronoun takes a different form before as opposed to after the verb.

He sees him.

She sees her.

They see them.

**Him sees he / *he sees he.*

**Her sees she / *her sees her.*

**Them sees they.*

Case Marking

The forms occurring *before* the verb,
happen to be called the **NOMINATIVE**
case...

He, she, they, we

The forms occurring *after* the verb, happen
to be called the **ACCUSATIVE** case...

Him, her, them, us

Case Marking

In DCGs...

```
pronoun(singular,nominative) --> [he]; [she]  
pronoun(singular,accusative) --> [him]; [her]  
pronoun(plural,nominative) --> [they]; [we]  
pronoun(plural,accusative) --> [them]; [us]
```

```
np(Number,Case) --> pronoun(Number,Case) .  
np(Number,_) --> d, n(Number)
```

```
vp(Number) --> v(Number), np(_,accusative)
```

```
s --> np(Number,nominative), vp(Number)
```

Subcategorization

The structure of an English VP depends on the verb...

Each verb has its own requirements -- these requirements are known as its *subcategorization frame*

VERB	COMPLEMENT	EXAMPLE
sleep, bark	None	(The cat) slept .
chase, see	One NP	(The dog) chased the cat.
give, sell	Two NPs	(Bob) sold Bill his car
say, claim	Sentence	Bob claimed the cat barked

Subcategorization

This means we really need >1 rule for VP.

In DCGs...

```
vp --> v(1) .
```

```
vp --> v(2), np .
```

```
vp --> v(3), np, np .
```

```
vp --> v(4), s .
```

```
v(1) --> [barked];[slept] .
```

```
v(2) --> [chased];[saw] .
```

```
v(3) --> [gave];[sold] .
```

```
v(4) --> [said];[thought] .
```

Separating the Lexicon

We can abstract the lexicon from the parser... this allows us to create more complex structures (e.g. a noun rule that adds “-ness” to a word to create a noun), as well as search more efficiently through the facts.

```
n --> [X], { noun(X) }.
```

```
noun(dog) .
```

```
noun(cat) .
```

```
noun(man) .
```

```
noun(N) :- ...
```

A Final Note

How do we use the parser we have prepared?

```
?- s([some,sentence,in,list,form],[ ]).
```