Chapter 12
Using Grammar Rules to Analyse English Sentences

Chapter Aims

After reading this chapter you should be able to:

- Understand and use the special syntax provided in Prolog for analyzing grammar rules.
- Define a simple grammar able to deal with basic sentences of English.
- Define predicates to enable the validity of sentences presented as lists of words to be established and to extract important information such as the type of each noun phrase from valid sentences.
- Define predicates to convert sentences in standard English into the 'list of words' form required by Prolog grammar rules.

12.1 Introduction

In this and the next chapter we will end the book by illustrating some of the uses to which Prolog can be put. We will focus on applications from the world of Artificial Intelligence (AI) as that was the field in which Prolog was originally developed.

As this is not a textbook on AI, the examples chosen may seem quite rudimentary, but do not be fooled by this. Prolog is a powerful language for AI projects in which some very substantial systems have been written.

12.2 Parsing English Sentences

Processing natural language (particularly sentences in English) was one of the earliest application areas for Prolog and no doubt largely because of this there is a special syntax available to support it in most versions of the language.

M. Bramer, *Logic Programming with Prolog*, DOI 10.1007/978-1-4471-5487-7_12,

© Springer-Verlag London 2013

We start by looking at how Prolog can be used to break down sentences of English into their component parts (nouns, verbs etc.), which is known as *parsing*. Do not worry – this is not going to turn into a book on the grammar of English so all the sentences we use for illustration will be very basic ones.

We can think of a simple English sentence having the form 'a noun, followed by a verb, followed by a noun', or even simpler: 'a noun followed by a verb'. In Prolog we can define a sentence this way by the two clauses:

```
sentence-->noun,verb,noun.
sentence-->noun,verb.
```

The –>**/2** operator can be read as 'is a' or 'comprises' or 'is made up of'. So the first clause indicates that a sentence can comprise a noun, followed by a verb, followed by a noun. As usual in Prolog we place the more specific clause before the more general one. (Note that the –> operator is three keystrokes: two hyphens followed by a 'greater than' symbol.)

This would be a possible way of defining sentences, but is very limited. It would allow 'man saw' and 'man saw dog' but not 'the man saw a dog'. Our first improvement to the above prototype program will be to change *noun* to '*noun_phrase*' which we will define as an optional determiner followed by a noun. (The words 'the', 'a' and 'an' are called *determiners*.)

This change brings us closer to a usable definition of a sentence, but we also need to define some nouns and verbs. Putting all these definitions together gives us a first version of a Prolog program defining the grammar of a very restricted version of English.

```
sentence-->noun_phrase,verb,noun_phrase.
sentence-->noun_phrase,verb.

noun_phrase-->determiner,noun.
noun_phrase-->noun.

verb-->[sat].
verb-->[saw].
verb-->[hears].
verb-->[took].
verb-->[sees].

determiner-->[the].
determiner-->[a].
determiner-->[an].
```

```
noun-->[cat].
noun-->[mat].
noun-->[man].
noun-->[boy].
noun-->[dog].
```

Terms such as *sentence*, *noun_phrase* and *verb* are called *syntactic terms*, to indicate that they are part of the structure of the English language. The list brackets around [mat] etc. are used to indicate that they are actual words in the language, not syntactic terms. Such words are often called *terminals*. Terminals can also be characters such as ',' or '?'.

The usual rules for Prolog atoms apply to syntactic terms such as *sentence* and *noun_phrase* and terminals such as *sat* and *took*, i.e. if they comprise only lower case letters, underscores and digits starting with a lower case letter they can be written without surrounding quotes, but any that begin with a capital letter, a digit or a symbol such as a question mark must be enclosed in quotes. This is why we wrote '?' above. In practice it is easiest to keep to lower case words.

We can now check whether a sentence such as 'the cat saw the mat' is valid in our restricted language. Prolog provides a special predicate **phrase/2** to do this. It takes two arguments: the first is a syntactic term such as *sentence* or *noun_phrase* (the left hand side of one of the –> operators) and the second is a list of words. So to check whether 'the cat saw the mat' is a valid sentence we can simply enter the query:

?- phrase(sentence,[the,cat,saw,the,mat]).
true .

The sequence of words 'the cat mat' is not a valid sentence, however.

?- phrase(sentence,[the,cat,mat]).
false.

A grammar such as the one above is called a 'definite clause grammar'. It is a 'context free' grammar, in the sense that it takes no account of the meaning of words, just whether they match the structure of the language (its syntax). Thus 'the mat saw the cat' is also a valid sentence.

?- phrase(sentence,[the,mat,saw,the,cat]).
true .

We can also test whether a sequence of words is a valid form of another syntactic term, such as a *noun_phrase*, e.g.

?- phrase(noun_phrase,[a,cat]).
true .

As this is Prolog we can also enter more complex queries, such as which words can validly come at the end of a sentence beginning 'the cat saw the':

?- phrase(sentence,[the,cat,saw,the,X]).
X = cat;
X = mat;
X = man;
X = boy;
X = dog;
false.

or which single word can end a sentence starting 'the cat saw'.

?- phrase(sentence,[the,cat,saw,X]).
X = cat;
X = mat;
X = man;
X = boy;
X = dog;
false.

or which two words can end a sentence beginning 'the cat saw':

?- phrase(sentence,[the,cat,saw,X,Y]).
X = the,
Y = cat;
X = the,
Y = mat;
X = the,
Y = man;
[etc.]

We can represent a compound verb such as 'will see' by a single word with an embedded underscore.

```
verb-->[will_see].
```

?- phrase(sentence,[the,man,will_see,the,cat]).
true .

We can now elaborate our language by introducing adjectives between the determiner and the noun in a *noun_phrase*, e.g. a large brown cat.

To allow for one adjective we can add an extra clause to the definition of *noun_phrase* giving

```
noun_phrase-->determiner,adjective,noun.
noun_phrase-->determiner,noun.
noun_phrase-->noun.
```

and define some adjectives, e.g.

```
adjective-->[large].
adjective-->[small].
adjective-->[brown].
adjective-->[orange].
adjective-->[green].
adjective-->[blue].
```

With this improved grammar we can verify some more complex sentences, e.g.:

?- phrase(sentence,[the,blue,cat,saw,the,large,man]).
true .

If we want to allow for a sequence of adjectives rather than just one, we can define an *adjective_sequence* which is either an adjective or an adjective followed by an *adjective_sequence*.

To do this change the first line in the definition of *noun_phrase* to

```
noun_phrase-->determiner,adjective_sequence,noun.
```

and add the definition of adjective_sequence

```
adjective_sequence-->adjective,adjective_sequence.
adjective_sequence-->adjective.
```

We can then verify quite lengthy sentences such as

?- phrase(sentence,[the,large,orange,man,saw,the,small,brown,orange,green, dog]).
true .

Before going any further, we need to step back and consider the clauses shown in this section so far. Although they certainly have a resemblance to them, they are not Prolog clauses (rules and facts) as defined and used elsewhere in this book. Clauses using the –> operator may be mixed freely with 'regular' Prolog clauses in a Prolog program and are essentially regular clauses 'in disguise'. For example the clause

```
verb-->[took].
```

is a 'disguised' form of the 'regular' Prolog clause (fact)

```
verb([took|A], A).
```

The latest version of the program developed in this section is as follows:

```
sentence-->noun_phrase,verb,noun_phrase.
sentence-->noun_phrase,verb.

noun_phrase-->determiner,adjective_sequence,noun.
noun_phrase-->determiner,noun.
noun_phrase-->noun.

verb-->[sat].
verb-->[saw].
verb-->[hears].
verb-->[took].
verb-->[sees].
verb-->[will_see].

adjective_sequence-->adjective,adjective_sequence.
adjective_sequence-->adjective.

determiner-->[the].
determiner-->[a].
determiner-->[an].

noun-->[cat].
noun-->[mat].
noun-->[man].
noun-->[boy].
noun-->[dog].

adjective-->[large].
adjective-->[small].
adjective-->[brown].
adjective-->[orange].
adjective-->[green].
adjective-->[blue].
```

If we use **the listing/1** predicate to see which rules and facts are in the database for predicates *sentence*, *noun_phrase*, *verb* etc. in turn, we can see that the above program is in fact stored as the 'regular' Prolog clauses:

```
sentence(A, D) :-
        noun_phrase(A, B),
        verb(B, C),
        noun_phrase(C, D).
sentence(A, C) :-
        noun_phrase(A, B),
        verb(B, C).

noun_phrase(A, D) :-
        determiner(A, B),
        adjective_sequence(B, C),
        noun(C, D).
noun_phrase(A, C) :-
        determiner(A, B),
        noun(B, C).
noun_phrase(A, B) :-
        noun(A, B).

verb([sat|A], A).
verb([saw|A], A).
verb([hears|A], A).
verb([took|A], A).
verb([sees|A], A).
verb([will_see|A], A).

adjective_sequence(A, C) :-
        adjective(A, B),
        adjective_sequence(B, C).
adjective_sequence(A, B) :-
        adjective(A, B).

determiner([the|A], A).
determiner([a|A], A).
determiner([an|A], A).

noun([cat|A], A).
noun([mat|A], A).
```

```
noun([man|A], A).
noun([boy|A], A).
noun([dog|A], A).

adjective([large|A], A).
adjective([small|A], A).
adjective([brown|A], A).
adjective([orange|A], A).
adjective([green|A], A).
adjective([blue|A], A).
```

This is considerably different! For example the syntactic term *sentence* has become the predicate **sentence** with two arguments. This is what was meant by saying that a special syntax is available for language parsing in Section 12.2. The –> operator is not merely an infix operator, using it enables a different syntax for Prolog to be used that is automatically converted to 'regular' Prolog clauses.

Special language syntax of this form is sometimes known by the slightly derogatory term *syntactic sugar*. The implication is presumably that 'real' programmers do not need such sweetening on top of their favourite language. While this may be true, there is no doubt that most people will find it much easier to work with Prolog clauses for language processing written using the –> notation than with the 'raw' Prolog clauses generated from them.

We will next show how the grammar rule notation can be used to ensure that a singular noun is followed by a singular verb and a plural noun is followed by a plural verb, i.e. that a noun and the following verb have the same *plurality*.

We first change the definition of our five nouns to

```
noun(singular)-->[cat].
noun(singular)-->[mat].
noun(singular)-->[man].
noun(singular)-->[boy].
noun(singular)-->[dog].
```

and then add the plural forms

```
noun(plural)-->[cats].
noun(plural)-->[mats].
noun(plural)-->[men].
noun(plural)-->[boys].
noun(plural)-->[dogs].
```

We next do the same for the verbs

```
verb(both)-->[sat].
verb(both)-->[saw].
verb(both)-->[took].
verb(both)-->[will_see].

verb(singular)-->[hears].
verb(singular)-->[sees].

verb(plural)-->[hear].
verb(plural)-->[see].
```

The first four verbs are labelled 'both', indicating that they are the same with either a singular or a plural noun ('the man sat', 'the men sat' etc.). Verbs 'hears' and 'sees' are labelled as 'singular' and we have added the plural forms 'hear' and 'see'.

We can now change the definitions of *sentence* and *noun_phrase* to incorporate information about the plurality of the first noun and the corresponding verb.

```
sentence-->noun_phrase(_),verb(both),noun_phrase(_).
sentence-->noun_phrase(_),verb(both).

sentence-->noun_phrase(Plurality),verb(Plurality),
   noun_phrase(_).
sentence-->noun_phrase(Plurality),verb(Plurality).

noun_phrase(Plurality)-->determiner,
   adjective_sequence,noun(Plurality).
noun_phrase(Plurality)-->determiner,noun(Plurality).
noun_phrase(Plurality)-->noun(Plurality).
```

The definition of *sentence* indicates that if the plurality of the verb is 'both' we do not care about the plurality of the preceding noun. However if the verb has a plurality that is 'singular' or 'plural' we require the first noun in the sentence to have the same plurality as the verb. In all cases we are unconcerned about the plurality of the second noun (if there is one).

?- phrase(sentence,[the,small,green,man,sees,a,large,dog]).
true .

?- phrase(sentence,[the,small,green,men,sees,a,large,dog]).
false.

?- phrase(sentence,[the,small,green,men,see,a,large,dog]).
true .

?- phrase(sentence,[the,small,green,man,sees,the,large,dogs]).
true .

?- phrase(sentence,[the,small,green,man,took,the,large,dogs]).
true .

?- phrase(sentence,[the,small,green,men,took,the,large,dogs]).
true .

It would be straightforward to associate a tense (past, present or future) with a verb in the same way that we have associated a plurality with nouns and verbs, but we will not pursue this possibility here.

Instead we will demonstrate that it is possible to include 'regular' Prolog in a grammar rule clause. To do this we make use of another piece of special syntax and enclose the 'regular' Prolog in {braces}.

Instead of the six grammar rules currently defining adjectives we can write:

```
adjective-->[X],{adjective_is(X)}.

adjective_is(large).
adjective_is(small).

adjective-->[brown].
adjective-->[orange].
adjective-->[green].
adjective-->[blue].
```

The first clause indicates that an *adjective* is any X such that *adjective_is(X)*. The other definitions of *adjective* (green etc.) could also have been converted into *adjective_is* form, of course, but they have been left as they were to indicate that a mixed notation is possible.

?- phrase(sentence,[the,man,saw,the,large,dog]).
true .

?- phrase(sentence,[the,man,saw,the,large,green,dog]).
true .

?- phrase(sentence,[the,green,man,saw,the,small,blue,large,boy]).
true .

Now we have the {brace} notation available, we can simplify the whole definition of *adjective* to the much more compact form:

```
adjective-->[X],
    {member(X,[large,small,brown,orange,green,blue])}.
```

?- phrase(sentence,[the,small,green,man,saw,the,large,orange,green,dog]).
true .

We can also simplify the definition of singular and plural nouns to just:

```
noun(singular)-->[X],{member(X,[cat,mat,man,
    boy,dog])}.
noun(plural)-->[X],{member(X,[cats,mats,men,
    boys,dogs])}.
```

?- phrase(sentence,[the,small,green,man,sees,a,large,dog]).
true .

?- phrase(sentence,[the,small,green,men,sees,a,large,dog]).
false.

?- phrase(sentence,[the,small,green,men,see,a,large,dog]).
true .

?- phrase(sentence,[the,small,green,man,sees,the,large,dogs]).
true .

?- phrase(sentence,[the,small,green,man,took,the,large,dogs]).
true .

?- phrase(sentence,[the,small,green,men,took,the,large,dogs]).
true .

Similarly the definition of *verb* can be simplified to:

```
verb(both)-->[X],{member(X,[sat,saw,took,will_see])}.
verb(singular)-->[X],{member(X,[hears,sees])}.
verb(plural)-->[X],{member(X,[hear,see])}.
```

As well as wishing to verify that a sentence is syntactically valid we may wish to show which of the four types of valid sentence we have defined applies in a particular case. We can do this by adding an argument to each of the *sentence* clauses.

```
sentence(s1)-->noun_phrase(_),verb(both),
    noun_phrase(_).
sentence(s2)-->noun_phrase(Plurality),verb(Plurality),
    noun_phrase(_).
sentence(s3)-->noun_phrase(_),verb(both).
sentence(s4)-->noun_phrase(Plurality),verb(Plurality).
```

?- phrase(sentence(Stype),[the,blue,man,saw,a,large,green,boy]).
Stype = s1 .

?- phrase(sentence(Stype),[the,small,green,men,hear]).
Stype = s4 .

We may also want to know which of the definitions of *noun_phrase* have been used. We can give them labels too:

```
noun_phrase(np1,Plurality)-->determiner,
    adjective_sequence,noun(Plurality).
noun_phrase(np2,Plurality)-->determiner,
    noun(Plurality).
noun_phrase(np3,Plurality)-->noun(Plurality).
```

There may be either one or two *noun_phrases* in a valid sentence. To allow for this, we could define two versions of *sentence*, one with two arguments (the *sentence* type, followed by the single *noun_phrase* type) and the other with three arguments (the *sentence* type and then two *noun_phrase* types).

However it is probably preferable for *sentence* to have a single argument which is a list. The first element of the list is always the *sentence* type (s1, s2 etc.). The rest of the list comprises the type of the one or two *noun_phrases* (np1,np2 etc.) that appear in the sentence, as appropriate.

```
sentence([s1,NP1,NP2])-->noun_phrase(NP1,_),
    verb(both),noun_phrase(NP2,_).
sentence([s2,NP1,NP2])-->noun_phrase(NP1,Plurality),
    verb(Plurality),noun_phrase(NP2,_).
sentence([s3,NP1])-->noun_phrase(NP1,_),verb(both).
sentence([s4,NP1])-->noun_phrase(NP1,Plurality),
    verb(Plurality).
```

?-phrase(sentence(S),[the,large,green,men,see,a,small,blue,dog]).
S = [s2, np1, np1] .

There is a valid sentence of type s2, with two noun_phrases, both of type np1.

?- phrase(sentence(S),[the,men,saw,dogs]).
S = [s1, np2, np3] .

?- phrase(sentence(S),[the,green,men,took]).
S = [s3, np1] .

We can also include the plurality of the verb as the second element of the list by changing the definition of *sentence* as follows:

```
sentence([s1,both,NP1,NP2])-->noun_phrase(NP1,_),
    verb(both),noun_phrase(NP2,_).
sentence([s2,Plurality,NP1,NP2])-->noun_phrase
   (NP1,Plurality),verb(Plurality),noun_phrase(NP2,_).
sentence([s3,both,NP1])-->noun_phrase(NP1,_)
   ,verb(both).
sentence([s4,Plurality,NP1])-->noun_phrase
   (NP1,Plurality),verb(Plurality).
```

?- phrase(sentence(S),[the,large,green,men,see,a,small,blue,dog]).
S = [s2, plural, np1, np1] .

?- phrase(sentence(S),[the,men,saw,dogs]).
S = [s1, both, np2, np3] .

?- phrase(sentence(S),[the,green,men,took]).
S = [s3, both, np1] .

This is clearly moving away from merely verifying that a sentence is syntactically valid to something closer to a linguistic analysis of the structure of the sentence.

We can add the verb itself as the third element of the list by changing the definitions of *sentence* and *verb*.

```
sentence([s1,both,V,NP1,NP2])-->noun_phrase(NP1,_),
   verb(both,V),noun_phrase(NP2,_).
sentence([s2,Plurality,V,NP1,NP2])
-->noun_phrase(NP1,Plurality),
   verb(Plurality,V),noun_phrase(NP2,_).
sentence([s3,both,V,NP1])-->noun_phrase(NP1,_),
verb(both,V).
sentence([s4,Plurality,V,NP1])
-->noun_phrase(NP1,Plurality),verb(Plurality,V).

verb(both,X)-->[X],{member(X,[sat,saw,
   took,will_see])}.
verb(singular,X)-->[X],{member(X,[hears,sees])}.
verb(plural,X)-->[X],{member(X,[hear,see])}.
```

?- phrase(sentence(S),[the,large,green,men,see,a,small,blue,dog]).
S = [s2, plural, see, np1, np1] .

?- phrase(sentence(S),[the,large,green,men,will_see,a,small,blue,dog]).
S = [s1, both, will_see, np1, np1] .

?- phrase(sentence(S),[the,green,small,men,see,a,blue,dog]).

S = [s2, plural, see, np1, np1] .

?- phrase(sentence(S),[the,green,small,men,took,a,blue,dog]).
S = [s1, both, took, np1, np1] .

?- phrase(sentence(S),[the,green,small,man,sees]).
S = [s4, singular, sees, np1] .

If we also want to get the noun or nouns into the list we can do so, starting by changing the definition of a *noun* to include as the second argument the word itself. Then we change the definition of *noun_phrase* to have a third argument, the noun word itself. Finally, we change the definition of *sentence* to put the noun or nouns into the list that is the argument of *sentence*.

```
sentence([s1,both,V,NP1,Noun1,NP2,Noun2])
-->noun_phrase(NP1,_,Noun1),
   verb(both,V),noun_phrase(NP2,_,Noun2).
sentence([s2,Plurality,V,NP1,Noun1,NP2,Noun2]
-->noun_phrase(NP1,Plurality,Noun1),
   verb(Plurality,V),noun_phrase(NP2,_,Noun2).
sentence([s3,both,V,NP1,Noun1])
-->noun_phrase(NP1,_,Noun1),verb(both,V).
sentence([s4,Plurality,V,NP1,Noun1])
-->noun_phrase(NP1,Plurality,Noun1),
   verb(Plurality,V).

noun_phrase(np1,Plurality,N)-->determiner,
   adjective_sequence,noun(Plurality,N).

noun_phrase(np2,Plurality,N)-->determiner,
   noun(Plurality,N).
noun_phrase(np3,Plurality,N)-->noun(Plurality,N).

noun(singular,X)-->[X],{member(X,[cat,mat,man,
   boy,dog])}.
noun(plural,X)-->[X],{member(X,[cats,mats,men,
   boys,dogs])}.
```

?- phrase(sentence(S),[the,green,small,men,see,a,blue,dog]).
S = [s2, plural, see, np1, men, np1, dog] .

?- phrase(sentence(S),[the,green,small,men,took,a,blue,dog]).
S = [s1, both, took, np1, men, np1, dog] .

?- phrase(sentence(S),[the,green,small,man,sees]).
S = [s4, singular, sees, np1, man] .

?- phrase(sentence(S),[the,green,small,men,will_see,a,blue,dog]).
S = [s1, both, will_see, np1, men, np1, dog] .

Finally, we may wish to compile a list of all the verbs that occur in a sequence of valid sentences that we analyse. This can be done by adding an **assertz** goal at the end of each of the four definitions of *sentence*.

```
sentence([s1,both,V,NP1,Noun1,NP2,Noun2])
-->noun_phrase(NP1,_,Noun1),verb(both,V),
   noun_phrase(NP2,_,Noun2),{assertz(wordlist
   (verb,both,V))}.
sentence([s2,Plurality,V,NP1,Noun1,NP2,Noun2])
-->noun_phrase(NP1,Plurality,Noun1),
   verb(Plurality,V),noun_phrase(NP2,_,Noun2),
   {assertz(wordlist(verb,Plurality,V))}.
sentence([s3,both,V,NP1,Noun1])-->noun_phrase
   (NP1,_,Noun1),
   verb(both,V),{assertz(wordlist(verb,both,V))}.
sentence([s4,Plurality,V,NP1,Noun1])
-->noun_phrase(NP1,Plurality,Noun1),
   verb(Plurality,V),{assertz(wordlist
   (verb,Plurality,V))}.
```

?- phrase(sentence(S),[the,man,sees,the,blue,green,small,dog]).
S = [s2, singular, sees, np2, man, np1, dog] .

?- phrase(sentence(S),[the,man,took,the,blue,green,small,dog]).
S = [s1, both, took, np2, man, np1, dog] .

?- phrase(sentence(S),[a,large,man,sees,the,blue,green,small,dog]).
S = [s2, singular, sees, np1, man, np1, dog] .

?- phrase(sentence(S),[the,men,hear,the,blue,green,small,dog]).
S = [s2, plural, hear, np2, men, np1, dog] .

?- listing(wordlist).

wordlist(verb, singular, sees).
wordlist(verb, both, took).
wordlist(verb, singular, sees).
wordlist(verb, plural, hear).

true.

There is clearly much more that could be done, but we will leave the parsing of English here. For reference, the complete program developed in this section is given below.

```
sentence([s1,both,V,NP1,Noun1,NP2,Noun2])
-->noun_phrase(NP1,_,Noun1),verb(both,V),
    noun_phrase(NP2,_,Noun2),{assertz(wordlist
    (verb,both,V))}.
sentence([s2,Plurality,V,NP1,Noun1,NP2,Noun2])
-->noun_phrase(NP1,Plurality,Noun1),verb(Plurality,V),
    noun_phrase(NP2,_,Noun2),
    {assertz(wordlist(verb,Plurality,V))}.
sentence([s3,both,V,NP1,Noun1])-->noun_phrase
    (NP1,_,Noun1),
    verb(both,V),{assertz(wordlist(verb,both,V))}.
sentence([s4,Plurality,V,NP1,Noun1])
-->noun_phrase(NP1,Plurality,Noun1),
    verb(Plurality,V),{assertz(wordlist
    (verb,Plurality,V))}.

noun_phrase(np1,Plurality,N)-->determiner,
    adjective_sequence,noun(Plurality,N).
noun_phrase(np2,Plurality,N)-->determiner,noun
    (Plurality,N).
noun_phrase(np3,Plurality,N)-->noun(Plurality,N).

verb(both,X)-->[X],{member(X,[sat,saw,took,
    will_see])}.
verb(singular,X)-->[X],{member(X,[hears,sees])}.
verb(plural,X)-->[X],{member(X,[hear,see])}.

adjective_sequence-->adjective,adjective_sequence.
adjective_sequence-->adjective.

determiner-->[the].
determiner-->[a].
determiner-->[an].

noun(singular,X)-->[X],{member(X,[cat,mat,man,boy,
    dog])}.
noun(plural,X)-->[X],{member(X,[cats,mats,men,boys,
    dogs])}.

adjective-->[X],
    {member(X,[large,small,brown,orange,green,blue])}.
```

12.3 Converting Sentences to List Form

The most obvious difficulty with the use of grammar rule syntax and the **phrase/2** predicate is that sentences do not naturally come in neat lists of words. Rather they can be lengthy sequences of words, with embedded spaces, commas, colons etc.

To illustrate the issues involved in converting real sentences to lists of words we will make use of a file named *dickens.txt* which contains the first six sentences of the celebrated story 'A Christmas Carol' by Charles Dickens.

```
Marley was dead: to begin with.

There is no doubt whatever about that.
The register of his burial was signed by the clergyman, the clerk, the undertaker,
and the chief mourner. Scrooge signed it. And Scrooge's name was good upon
'Change, for anything he chose to put his hand to.

Old Marley was as dead as a door-nail.
```

There are a number of points to note:

- A sentence can run over more than one line of the file.
- More than one sentence can appear on the same line.
- Sentences are separated by at least one space or end of line character.
- The final sentence is followed by a blank line and then an end-of-file marker.
- The words in sentences are separated by spaces, end of line characters or punctuation marks such as commas and colons.
- Sentences end with a terminator (full stop, exclamation mark or question mark, although only full stops are used in this example).
- Spaces, end of line characters, punctuation marks (such as commas and colons) and terminators (full stops, exclamation marks and question marks) should all be removed. However apostrophes are part of words such as Scrooge's and should not be removed.

Converting sentences to list form is difficult to do, much more so than most of the other programs in this book. A Prolog program to convert the *dickens.txt* file to six sentences in list form is given below without explanation but for possible use in the reader's own programs. As in Chapter 5 we will assume that the ASCII characters corresponding to 'end of line' and 'end of file' have ASCII values 10 and −1 respectively.

```
readlineF(File):-
see(File),repeat,inputline(L),L=[end_of_file],!,seen.

inputline(L):-buildlist(L,[]),reverse(L,L1),
    writeout(L1),!.

writeout([]).
writeout([end_of_file]).
writeout(L):-write('Sentence: '),write(L),nl.

buildlist(L,OldL):-findword(Word,[]),
(
    (Word=[],L=OldL);
    (Word=[end_of_file],L=[end_of_file]);
    (Word=[sep],buildlist(L,OldL));
    (Word=[termin|Word1],name(S,Word1),L=[S|OldL]);
    (name(S,Word),buildlist(L,[S|OldL]))
).

findword(Word,OldWord):-get0(X),
(
    (terminator(X),Word=[termin|OldWord]);
    (separator(X),((OldWord=[],Word=[sep]);
    Word=OldWord));
    (X<0,Word=[end_of_file]);
    (append(OldWord,[X],New),findword(Word,New))
).

separator(10). /* end of line */
separator(32). /* space*/
separator(44). /* comma */
separator(58). /* colon */

terminator(46). /* full stop */
terminator(33). /* exclamation mark */
terminator(63). /* question mark */
```

?- readlineF('dickens.txt').
Sentence: [Marley,was,dead,to,begin,with]
Sentence: [There,is,no,doubt,whatever,about,that]
Sentence: [The,register,of,his,burial,was,signed,by,the,clergyman,the,clerk,the,undertaker,and,the,chief,mourner]
Sentence: [Scrooge,signed,it]

Sentence: [And,Scrooge's,name,was,good,upon,'Change,for,anything,he,chose, to,put,his,hand,to]
Sentence: [Old,Marley,was,as,dead,as,a,door-nail]
true.

One improvement to this program would be to replace all the words by their lower case equivalents, i.e. change 'There' to 'here', 'And' to 'and' etc. Words appearing at the start of a sentence are generally spelt with an initial capital letter and those that do not are generally spelt with an initial lower case letter. For a practical system for analyzing sentences standardizing all words to begin with a lower case letter (e.g. standardizing both 'There' and 'there' to 'there') reduces the number of words that have to be stored considerably. It can be achieved quite easily by adjusting the **get0(X)** goal in the first line of the definition of f**indword** so that if the character input is an upper case letter (values 65 to 90 inclusive) it is changed to its lower case equivalent (97 to 122 inclusive). Other characters are left unchanged.

The first line of the definition of **findword** should be changed to:

```
findword(Word,OldWord):-get0(X1),repchar(X1,X),
```

A new predicate **repchar** also needs to be added, defined as follows.

```
repchar(X,New):-X>=65,X=<90,New is X+32,!.
repchar(Char,Char).
```

Now the effect of using readlineF is:

?- readlineF('dickens.txt').
Sentence: [marley,was,dead,to,begin,with]
Sentence: [there,is,no,doubt,whatever,about,that]
Sentence: [the,register,of,his,burial,was,signed,by,the,clergyman,the,clerk,the, undertaker,and,the,chief,mourner]
Sentence: [scrooge,signed,it]
Sentence: [and,scrooge's,name,was,good,upon,'change,for,anything,he,chose,to, put,his,hand,to]
Sentence: [old,marley,was,as,dead,as,a,door-nail]
true.

A second desirable change is to write the new sentences into a text file so that they can be read in again for subsequent processing. This can be achieved by giving the **readlineF** predicate an extra argument and changing its definition to

```
readlineF(File,Outfile):-
    see(File),tell(Outfile),repeat,inputline(L),
    L=[end_of_file],!,told,seen.
```

and changing the final clause of **writeout** to:

```
writeout(L):-writeq(L),write('.'),nl.
```

Note the use of the **writeq** rather than the **write** predicate here and also that each list is output with a full stop and a newline character after it.

?- readlineF('dickens.txt','newdickens.txt').
true.

The file *newdickens.txt* now contains

```
[marley,was,dead,to,begin,with].
[there,is,no,doubt,whatever,about,that].
[the,register,of,his,burial,was,signed,by,the,clergyman,the,clerk,the,
undertaker,and,the,chief,mourner].
[scrooge,signed,it].
[and,'scrooge \ 's',name,was,good,upon,' \ 'change',for,anything,he,
chose,to,put,his,hand,to].
[old,marley,was,as,dead,as,a,'door-nail'].
```

Thanks to the use of **writeq**, the original word Scrooge's has been changed to the atom 'scrooge\'s' enclosed in quotes and with the embedded quote character written as \'. Similarly Dickens's word 'Change (an archaic way of writing the word Exchange) has been changed to '\'change'.

In this form each list is a valid Prolog term, terminated by a full stop, which can be processed by a separate Prolog program which analyses the contents of sentences.

The extract from 'A Christmas Carol' was chosen to illustrate the complexity of even a small number of sentences of 'real' English. To parse just the first six sentences of this famous story would require a considerably more complex grammar than the one we have developed so far, which would take us far outside the scope of this book.

Sadly we will go back to much simpler examples and use a file *sentences.txt* containing five sentences that are valid with the grammar defined in Section 12.2 and one sentence (the third) that is invalid.

```
[the,large,green,men,see,a,small,blue,dog].
[the,large,green,men,will_see,a,small,blue,dog].
[the,man].
[the,green,small,men,see,a,blue,dog].
[the,green,small,men,took,a,blue,dog].
[the,green,small,man,sees].
```

To process this file we need to add just a few lines to the program for analyzing sentences given at the end of Section 12.2. The second clause of predicate **proc2** enables us to trap the case where a list of words is not a valid sentence, at least as far as the grammar we have defined is concerned.

```
process(File):-
    see(File),repeat,read(S),proc(S),S=end_of_file,!,
    seen.

proc(end_of_file).
proc(S):-write('Sentence: '),write(S),nl,proc2(S).

proc2(S):-phrase(sentence(L1),S),write('Structure: '),
    write(L1),nl,nl,!.
proc2(S):-write('Invalid sentence structure'),nl,nl.
```

?-process('sentences.txt').
Sentence: [the,large,green,men,see,a,small,blue,dog]
Structure: [s2,plural,see,np1,men,np1,dog]

Sentence: [the,large,green,men,will_see,a,small,blue,dog]
Structure: [s1,both,will_see,np1,men,np1,dog]

Sentence: [the,man]
Invalid sentence structure

Sentence: [the,green,small,men,see,a,blue,dog]
Structure: [s2,plural,see,np1,men,np1,dog]

Sentence: [the,green,small,men,took,a,blue,dog]
Structure: [s1,both,took,np1,men,np1,dog]

Sentence: [the,green,small,man,sees]
Structure: [s4,singular,sees,np1,man]

true.

Chapter Summary

This chapter describes the use of the special syntax provided in Prolog for analyzing grammar rules: the operator –>**/2**, the predicate **phrase/2** and braces to enclose 'regular' Prolog used in conjunction with grammar rules. A simple grammar able to deal with basic sentences is defined. Predicates are given to enable the validity of sentences presented as lists of words to be established and to extract important information such as the type of each *noun_phrase* from valid sentences. Finally, predicates are defined to convert sentences in standard English into the 'list of words' form required by the grammar rules.

Practical Exercise 12

Extend the grammar rules given at the end of Section 12.2 to allow for the possibility of an adverb at the end of a sentence of type s3 or s4. Define the following words as adverbs: well, badly, quickly, slowly.