Now try again...

```
?- reverse([a,b,c,d], X).
no
```

Now reverse doesn't work. Try a longer definition...

```
?- consult(user).

append([], X, X).
append([A|B], C, [A|D]) :- append(B, C, D).

/* the end of file character is typed here */

yes
```

Now try again...

```
?- reverse([a,b,c,d,e], X).
X = [e,d,c,b,a]
yes
```

In this session, the programmer starts by entering clauses for the predicates append and reverse from the terminal. Of course, the programmer could have typed these into a file first and then told Prolog to consult that file, but for an example of this small size that might not have been worthwhile. Unfortunately, there is a mistake in the first clause for append. The goal contains an A where there should be a B. This mistake is revealed when the system cannot answer the append and reverse questions. Somehow, the programmer realises that the definition of append is wrong (in a real session, this would probably happen after use was made of the debugging aids). So he decides to replace his existing definition with a new one, using consult. Unfortunately, in the new definition, he forgets to specify the boundary condition (the [] case). So the program still does not work. At this point, the original two-clause definition of append has been replaced by a new one-clause definition, which is not complete. The programmer sees what he has done, and can rectify the situation by simply adding a new clause to the existing definition. This is achieved with another use of consult. The program now works.

In conclusion, when you are making changes to a program, exercise the same care that you take when you write the first version of a program. Make sure that what you add is still compatible with your conventions about which variables should be instantiated when and what arguments are used for what purposes. Above all, take the opportunity to look over the program again: there may be some other mistakes in it!

9

Using Prolog Grammar Rules

9.1 The Parsing Problem

Sentences in a language such as English are much more than just arbitrary sequences of words. We cannot string together any set of words and make a reasonable sentence. At the very least, the result must conform to what we consider to be grammatical.

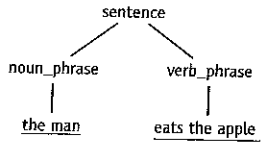
A grammar for a language is a set of rules for specifying what sequences of words are acceptable as sentences of that language. It specifies how the words must group together into phrases and what orderings of these phrases are allowed. Given a grammar for a language, we can look at any sequence of words and see whether it meets the criteria for being an acceptable sentence. If the sequence is indeed acceptable, the process of verifying this will have established what the natural groups of words are and how they are put together. That is, it will have established something of the underlying structure of the sentence.

A particularly simple kind of grammar is known as a "context free" grammar. Rather than give a formal definition of what such a thing is, we will illustrate it by means of a simple example. The following might be the start of a grammar of English sentences:

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

determiner --> [the].

noun --> [apple].
noun --> [man].

verb --> [eats].
```

```
verb --> [sings].
```

The grammar consists of a set of rules, here shown one to a line. Each rule specifies a form that a certain kind of phrase can take. The first rule says that a sentence consists of a phrase called a noun_phrase followed by a phrase called a verb_phrase. These two phrases are what are commonly known as the subject and predicate of the sentence, and their configuration can be illustrated in a tree as follows:



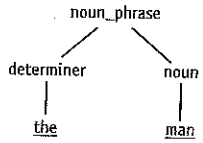
To see what a rule in a context free grammar means, read "X --> Y" as saying "X can take the form Y", and read "X,Y" as saying "X followed by Y." Thus, the first rule can be read as:

A sentence can take the form: a noun_phrase followed by a verb_phrase.

This is all very well, but what is a noun_phrase and what is a verb_phrase? How are we to recognise such things and to know what constitute grammatical forms for them? The second, third and fourth rules of the grammar go on to answer these questions. For instance,

A noun_phrase can take the form: a determiner followed by a noun.

Informally, a noun phrase is a group of words that names a thing (or things). Such a phrase contains a word, the "noun", which gives the main class that the thing belongs to. Thus "the man" names a man, "the program" names a program and so on. Also, according to this grammar, the noun is preceded by a phrase called a "determiner":



Similarly, the internal structure for a verb_phrase is described by the rules. Notice that there are two rules for what a verb_phrase is. This is because (according to this

grammar) there are two possible forms. A verb_phrase can contain a noun_phrase, as in "the man eats the apple", or it need not, as in "the man sings."

What are the other rules in the grammar for? These express how some phrases can be made up in terms of actual words, rather than in terms of smaller phrases. The things inside square brackets name actual words of the language, so that the rule:

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

can be read as:

A determiner can take the form: the word the.

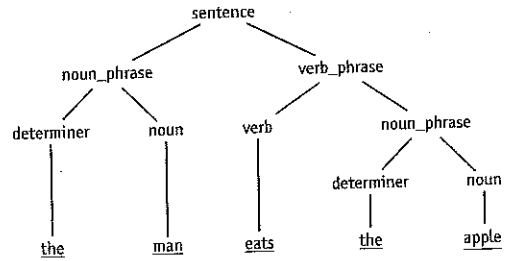
Now that we have got through the whole of the grammar, we can begin to see which sequences of words are actually sentences according to the grammar. This is a very simple grammar and needs extending in many ways, especially as it will only accept sentences formed out of five different words. If we wish to investigate whether a given sequence of words is actually a sentence according to these criteria, we need to apply the first rule, and this reduces the problem to:

Does the sequence decompose into two phrases, such that the first is an acceptable noun_phrase and the second is an acceptable verb_phrase?

Then in order to test whether the first phrase is a noun phrase, we need to apply the second rule, asking,

Does it decompose into a determiner followed by a noun?

and so on. At the end, if we succeed, we will have located all the phrases and sub-phrases of the sentence, as specified by the grammar, and will have established a structure for that sentence such as, for instance:



This diagram showing the phrase structure of the sentence is called a *parse tree* for the sentence.

We have seen how having a grammar for a language means that we can construct parse trees to show the structure of sentences of the language. The problem of constructing a parse tree for a sentence, given a grammar, is what we call the *parsing problem*. A computer program that constructs parse trees for sentences of a language we shall call a parser.

This chapter illustrates how the parsing problem can be formulated in Prolog, and introduces the Prolog grammar rule formalism, which makes it rather more convenient to write parsers in Prolog. Although a parser for the grammar rule formalism (also called "Definite Clause Grammars" or "DCGs") is not actually part of the definition of Standard Prolog, it is provided automatically by many Prolog implementations. The usefulness of DCGs is not confined to applications concerned with the syntax of natural languages. Indeed, the same techniques apply to any problem where we are presented with an ordered sequence of items that seem to fall into natural groups and where the arrangement of these groups can be specified by a set of rules. However, for the sake of simplicity the rest of the chapter will concentrate on the problem of parsing English sentences and the generalisation to other fields will be left to you.

9.2 Representing the Parsing Problem in Prolog

The primary structure that we are talking about in discussing the parsing problem is the sequence of words whose structure is to be determined. We expect to be able to isolate subsequences of this structure as being various phrases accepted by the grammar, and to show in the end that the whole sequence is acceptable as a phrase of type sentence. Because a standard way of representing a sequence is as a list, we shall represent the input to the parser as a Prolog list. What about the representation of the words themselves? For the moment, there seems to be no point in giving the words internal structure. All we want to do is compare words with one another. Hence it seems reasonable to represent them as Prolog atoms.

Let us develop a program to see if a given sequence of words is a sentence according to the grammar shown above. In order to do this, it will have to establish the underlying structure of the sentences it is given. We will later on consider how to develop a program that remembers this structure and displays it to us, but for now it will be easier if we ignore this extra complexity. Since the program involves testing to see if something is a sentence, let us define a predicate sentence. The predicate will only need one argument, and we will give it a meaning as follows:

sentence(X) means that:
X is a sequence of words forming a grammatical sentence.

So we anticipate asking questions such as:


```
?- sentence([the,man,eats,the,apple]).
```

This will succeed if "the man eats the apple" is a sentence and fail otherwise.

It is clumsy to specify sentences artificially by giving lists of Prolog atoms. For a more serious application, we would probably want to be able to type English sentences at the terminal in the normal way. In Chapter 5, we saw how a predicate read_in can be defined so that we can convert a sentence typed in to a list of Prolog atoms. We could obviously build this into our parser to allow a more natural means of communication with the program's user. However, we will ignore these "cosmetic" matters for now and concentrate on the real problem of parsing.

What is involved in testing to see whether a sequence of words is a sentence? Well, according to the first rule of the grammar, the task decomposes into finding a noun_phrase at the beginning of the sequence and then finding a verb_phrase in what is left. At the end of this, we should have used up exactly the words of the sequence, no more and no less. Let us introduce the predicates noun_phrase and verb_phrase to express the properties of being a noun phrase or verb phrase, so that:

noun_phrase(X) means that: sequence X is a noun phrase.

Also,

verb_phrase(X) means that: sequence X is a verb phrase.

We can put together a definition of sentence in terms of these predicates. A sequence X is a sentence if it decomposes into two subsequences Y and Z, where Y is a noun_phrase and Z is a verb_phrase. Since we are representing sequences as lists, we already have available the predicate append for decomposing one list into two others. So we can write:

```
sentence(X) :-
    append(Y, Z, X), noun_phrase(Y), verb_phrase(Z).
```

Similarly,

```
noun_phrase(X) :-
    append(Y, Z, X), determiner(Y), noun(Z).

verb_phrase(X) :-
    append(Y, Z, X), verb(Y), noun_phrase(Z).
verb_phrase(X) :- verb(X).
```

Notice that the two rules for verb_phrase give rise to two clauses for the predicate, corresponding to the two ways of verifying that a sequence is a verb_phrase. Finally, we can easily deal with the rules that introduce words:

```
determiner([the]).
```

```
noun([apple]).
noun([man]).

verb([eats]).
verb([sings]).
```

So our program is complete. Indeed, this program will successfully tell us which sequences of words are sentences according to the grammar. However, before we consider the task complete, we should have a look at what actually happens when we ask questions about some example sequences. Consider just the sentence clause:

```
sentence(X) :-
    append(Y, Z, X), noun_phrase(Y), verb_phrase(Z).
```

and a question:

```
?- sentence([the,man,eats,the,apple]).
```

Variable X in the rule will be instantiated (to [the, man, eats, the, apple]), but initially Y and Z will be uninstantiated, so the goal will generate a possible pair of values for Y and Z such that when Z is appended to Y the result is X. On backtracking, it will generate all the possible pairs, one at a time. The noun_phrase goal will only succeed if the value for Y actually is an acceptable noun_phrase. Otherwise it will fail, and append will have to propose another value. So the flow of control for the first part of the execution will be something like:

1. The goal is sentence([the, man, eats, the, apple]).
2. Decompose the list into two lists Y and Z. The following decompositions are possible:

   ```
   Y = [], Z = [the,man,eats,the,apple]
   Y = [the], Z = [man,eats,the,apple]
   Y = [the,man], Z = [eats,the,apple]
   Y = [the,man,eats], Z = [the,apple]
   Y = [the,man,eats,the], Z = [apple]
   Y = [the,man,eats,the,apple], Z = []
   ```

3. Choose a possibility for Y and Z from the above list of possibilities, and see if Y is a noun_phrase. That is, try to satisfy noun_phrase(Y).
4. If Y is a noun_phrase, then succeed (and then look for a verb_phrase). Otherwise, go back to Step 3 and try another possibility.

There seems to be a lot of unnecessary searching in this approach. The goal append(Y,Z,X) generates a large number of solutions, most of which are useless from

the point of view of identifying noun phrases. There must be a more directed way of getting to the solution. As our grammar stands, a noun_phrase must have precisely two words in it, and so we might think of using this fact to avoid searching among possible decompositions of the sequence. The trouble is that this state of affairs may not stay true if we change the grammar. Even a small change in the rules for determiner could affect the possible lengths of noun phrases and hence affect the way in which the presence of noun phrases would be tested. In designing the program it would be nice to retain some modularity. If we wish to change one clause, it should not necessarily have ramifications for the whole program.

So, the heuristic about the length of noun phrases is too specific to be built into the program. Nevertheless, we can see it as a specific case of a general principle. If we wish to select a subsequence that is a noun phrase, then we can look at the properties of noun phrases to restrict what kinds of sequences are actually proposed. If the noun phrase definition is liable to change, we cannot do this, unless we hand over the whole responsibility to the noun_phrase clauses. Since it is the noun_phrase clauses that express what the properties of noun phrases are, why not expect them to decide how much of the sequence is to be looked at? Let us require the noun_phrase clauses to decide how much of the sequence is to be consumed, and what is to be left for the verb_phrase definition to work on.

This discussion leads us to consider a new definition for the noun_phrase predicate, this time involving two arguments:

```
noun_phrase(X,Y) is true if
        there is a noun phrase at the beginning of sequence X
        and the part of the sequence left after the noun phrase is Y.
```

So we might expect these questions:

```
?- noun_phrase([the,man,eats,the,apple], [eats,the,apple]).
?- noun_phrase([the,apple,sings], [sings]).
?- noun_phrase([the,man,eats,the,apple], X).
?- noun_phrase([the,apple,sings], X).
```

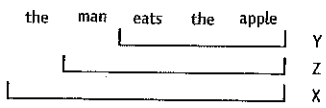
all to succeed, the last two instantiating the variable X to whatever in the list follows the noun_phrase.

We must now revise the definition of noun_phrase to reflect this change of meaning. In doing this, we must resolve how the sequence taken up by a noun phrase decomposes into a sequence taken up by a determiner followed by a sequence taken up by a noun. We can again delegate the problem of how much of the sequence is taken up to the clauses for the embedded phrases, giving the following:

```
noun_phrase(X, Y) :- determiner(X, Z), noun(Z, Y)
```

So a noun_phrase exists at the beginning of sequence X if we can find a determiner at the front of X, leaving behind Z, and we can then find a noun at the front of Z. The

amount of the sequence left behind by the whole noun phrase is the same as that left behind after the noun (Y). Expressed diagramatically:



In order for this to work, we will have to adopt a similar convention with determiner and noun as we did with noun_phrase.

This clause tells us how the problem of finding a sequence that is a noun phrase decomposes into finding subsequences that is a determiner followed by one which is a noun. Similarly, the problem of finding a sentence decomposes into finding a noun phrase followed by a verb phrase. This is all very abstract. None of this tells us how many words are actually consumed in the determiner, noun phrase or sentence. The information must be built up from our version of the rules that actually introduce English words. We can again express these as Prolog clauses, but this time we need to add an extra argument, to give for example:

```
determiner([the|X], X).
```

This rule expresses the fact that one can find a determiner at the front of a sequence beginning with the word the. Moreover, the determiner only takes up the first word of the sequence, and leaves the rest behind.

In fact, we can add an extra argument to every predicate that recognises a kind of phrase, to express how that kind of phrase "uses up" some of the words of a sequence and leaves the rest. In particular, for consistency, it would be sensible to do this with the sentence predicate. How does the initial goal that we give to the program look now? We must decide on what two arguments to give to sentence in the question. The arguments indicate the sequence that it starts from and the sequence it is to leave behind. The first of these is obviously the same as the argument we gave to sentence before. Moreover, since we want to find a sentence that occupies the whole of the sequence, we want nothing left after the sentence has been found. We want only the empty sequence to be left. Hence we must give the program a goal like:

```
?- sentence([the,man,eats,the,apple], []).
```

Let us now see how the complete grammar looks after we have rewritten it with the above discussion in mind:


```
sentence(S0, S) :-
        noun_phrase(S0, S1),
        verb_phrase(S1, S).

noun_phrase(S0, S) :- determiner(S0, S1), noun(S1, S).

verb_phrase(S0, S) :- verb(S0, S).
verb_phrase(S0, S) :- verb(S0, S1), noun_phrase(S1, S).

determiner([the|S], S).

noun([man|S], S).
noun([apple|S], S).

verb([eats|S], S).
verb([sings|S], S).
```

So we now have a more efficient version of our program to recognise sentences accepted by the grammar. It is a pity, though, that the code looks more messy than that of the previous version. All the extra arguments seem to clutter it up unnecessarily. We shall now see how to cope with this problem.

9.3 The Grammar Rule Notation

The Prolog grammar rule notation was developed as an aid to people writing parsers using the techniques we have just described. The notation makes the code easier to read, because it suppresses information that is not interesting. Because the notation is more concise than ordinary Prolog, there is also less chance of making silly typing mistakes if you use grammar rules for writing your parsers.

Although the grammar rule notation is self-contained, it is important to realise that it is only a shorthand for ordinary Prolog code. You can use grammar rules either because they are built-in to your Prolog system already, or because there is a library package (such as Appendix D that enables you to use a special form of consult. In either case, the way that the system handles grammar rules is to recognise them when they are input and then translate them into ordinary Prolog. So your grammar rules end up as ordinary Prolog clauses, although naturally looking a bit different from what you typed in.

The actual notation is built around the notation for context-free grammars that we introduced at the beginning of this chapter. In fact, if the grammar as presented there (reproduced below) were given to Prolog, it would be translated into clauses exactly the same as what we ended up with as the final version of the parsing program:

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.

determiner --> [the].

noun --> [man].
noun --> [apple].

verb --> [eats].
verb --> [sings].
```

The actual grammar rules are Prolog structures, with main functor "-->", which is declared as an infix operator. All the Prolog system has to do is check whether a term read in (in a consult or similar) has this functor, and if so translate it into a proper clause.

What is involved in this translation? First of all, every atom that names a kind of phrase must be translated into a predicate with two arguments. One argument is for the sequence provided, and the other is for the sequence left behind, as in our program above. Second, whenever a grammar rule mentions phrases coming one after another, it must be arranged that that the arguments reflect the fact that what is left behind by one phrase forms the input to the next. Finally, whenever a grammar rule mentions that a phrase can be realised as a sequence of subphrases, the arguments must express the fact that the amount of words consumed by the whole phrase is the same as the total consumed by the subphrases mentioned on the right of the "-->". These criteria ensure, for instance, that:

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

translates into:

```
sentence(S0, S) :-
        noun_phrase(S0, S1), verb_phrase(S1, S).
```

or, in English,

> There is a sentence between S0 and S if: there is a noun phrase between S0 and S1, and if there is a verb phrase between S1 and S.

Finally, the system has to know how to translate those rules that introduce actual words. This involves inserting the words into the lists forming the arguments of the predicates, so that, for instance,

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

translates into:


```
determiner([the|S], S).
```

Once we have expressed our parsing program as grammar rules, how do we specify the goals that we want it to work at? Since we now know how grammar rules translate into ordinary Prolog, we can express our goals in Prolog, adding the extra arguments ourselves. The first argument to add is the list of words that is to be looked at, and the second is the list that is going to be left, which is normally the empty list, []. So, we can specify goals such as:

```
?- sentence([the,man,eats,the,apple], []).
?- noun_phrase([the,man,sings], X).
```

As an alternative, some Prolog implementations provide a built-in predicate phrase which simply adds the extra arguments for you. The predicate phrase is defined by:

phrase(P, L) is true if: list L can be parsed as a phrase of type P.

So we could replace the first of the above goals by the alternative:

```
?- phrase(sentence, [the,man,eats,the,apple]).
```

Note that the definition of phrase involves the whole list being parsed, with the empty list being left. Therefore we could not replace the second goal above by a use of phrase.

If your Prolog implementation does not provide phrase already defined, you can easily provide a clause for it, as follows:

```
phrase(P,L) :- Goal =.. [P, L, []], call(Goal).
```

Note, however, that this definition will not be adequate when we consider more general grammar rules in the next section.

9.4 Adding Extra Arguments

The grammar rules we have considered so far are only of a fairly restricted kind. In this section we will consider one useful extension, which allows phrase types to have extra arguments. This extension is still part of the standard grammar rule facility that most Prolog systems provide.

We have seen how an occurrence of a phrase type in a grammar rule translates to the use of a Prolog predicate with two extra arguments. So the rules we have seen so far give rise to a lot of two-argument predicates. Now Prolog predicates can have any number of arguments, and we may sometimes want to have extra arguments used in our parsers, apart from the ones dealing with the consumption of the input sequence. The grammar rule notation supports this.

Let us look at an example where extra arguments may be useful. Consider the problem of "number agreement" between the subject and verb of a sentence. Sequences like

★ The boys eats the apple.
★ The boy eat the apple.

are not grammatical English sentences, even though they might be allowed by a simple extension of our grammar (the "★" is a convention used to denote an ungrammatical sentence). The reason they are not grammatical is that if the subject of a sentence is singular then the sentence must also use the singular form of the verb. Similarly, if the subject is plural, the plural form of the verb must be used. We could express this in grammar rules by saying that there are two kinds of sentences, singular sentences and plural sentences. A singular sentence must start with a singular noun phrase, which must have a singular noun, and so on. We would end up with a set of rules like the following:

```
sentence --> singular_sentence.
sentence --> plural_sentence.

noun_phrase --> singular_noun_phrase.
noun_phrase --> plural_noun_phrase.

singular_sentence -->
    singular_noun_phrase, singular_verb_phrase.

singular_noun_phrase -->
    singular_determiner, singular_noun.

singular_verb_phrase --> singular_verb, noun_phrase.
singular_verb_phrase --> singular_verb.

singular_determiner --> [the].

singular_noun --> [boy].

singular_verb --> [eats].
```

and also a whole lot of rules for plural phrases. This is not very elegant, and obscures the fact that singular and plural sentences have a lot of structure in common. A better way is to associate an extra argument with phrase types, according to whether they are singular or plural. Thus sentence(singular) names a phrase which is a singular sentence and, in general, sentence(X) a sentence of plurality X. The rules about number agreement then amount to consistency conditions on the values of these arguments. The plurality of the subject noun phrase must be the same as that of the verb phrase, and so on. Rewriting the grammar in this way, we get:


```
sentence --> sentence(X).
sentence(X) --> noun_phrase(X), verb_phrase(X).
noun_phrase(X) --> determiner(X), noun(X).
verb_phrase(X) --> verb(X).
verb_phrase(X) --> verb(X), noun_phrase(Y).
noun(singular) --> [boy].
noun(plural) --> [boys].
determiner(_) --> [the].
verb(singular) --> [eats].
verb(plural) --> [eat].
```

Note the way in which we can specify the plurality of the. This word could introduce a singular or a plural phrase, and so its plurality is compatible with anything. Also note that in the second rule for verb_phrase the naming of the variables expresses the fact that the plurality of a verb phrase (the thing that must agree with the subject) is taken from that of the verb, and not that of the object, if there is one.

We can introduce arguments to express other important information as well as number agreement. For instance, we can use them to keep a record of constituents that have appeared outside their "normal" position, and hence deal with the phenomena that linguists call "movement". Or we can use them to record items of semantic significance, for example to say how the meaning of a phrase is composed of the meanings of the subphrases. We will not investigate these any more here, although Section 9.6 gives a simple example of incorporating semantics into the parser. However, one point should be noted here. Linguists may be interested to know that once we introduce extra arguments into grammar rules, we cannot guarantee that the language defined by the grammar is still context-free, although it often will be.

An important use of extra arguments is to return a parse tree as a result of the analysis. In Chapter 3 we saw how trees can be represented as Prolog structures, and we will now make use of that in extending the parser to make a parse tree. Parse trees are helpful because they provide a structural representation of a sentence. It is convenient to write programs that process this structural representation in a way analogous to processing the arithmetic formulae and lists in Chapter 7. The new program, given a grammatical sentence like:

The man eats the apple.

will generate a structure like this:

```
sentence(
        noun_phrase(
                determiner(the),
                noun(man)),
```

```
        verb_phrase(
                verb(eats),
                noun_phrase(
                        determiner(the),
                        noun(apple))
                )
        )
```

as a result. In order to make it do this, we only need to add an extra argument to each predicate, saying how the tree for a whole phrase is constructed from the trees of the various sub-phrases. Thus we can change the first rule to:

```
sentence(X, sentence(NP, VP)) -->
        noun_phrase(X, NP), verb_phrase(X, VP).
```

This says that if we can find a sequence constituting a noun phrase, with parse tree NP, followed by a sequence constituting a verb phrase, with parse tree VP, then we have found a sequence constituting a complete sentence, and the parse tree for that sentence is sentence(NP,VP). Or, in more procedural terms, to parse a sentence one must find a noun phrase followed by a verb phrase, and then combine the parse trees of these two constituents, using the functor sentence to make the tree for the sentence.

It is only coincidental that we have named the grammar rule sentence as well as the sentence node of the parse tree. We could have used, say, s to name the parse tree node instead. Note that the X arguments are just the number agreement arguments used earlier, and that the decision to put the tree generating arguments after rather then before them was arbitrary. If you have any difficulty understanding this extension, it helps to see that this is all just a shorthand for an ordinary Prolog clause:

```
sentence(X, sentence(NP, VP), S0, S) :-
        noun_phrase(X, NP, S0, S1),
        verb_phrase(X, VP, S1, S).
```

where S0, S1 and S stand for parts of the input sequence. We can introduce tree-building arguments throughout the grammar in a routine way. Here is an excerpt from what is produced if we do this (number agreement arguments being left out for clarity):

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

verb_phrase(verb_phrase(V)) --> verb(V).

noun(noun(man)) --> [man].

verb(verb(eats)) --> [eats].
```


The translation mechanism needed to deal with grammar rules with extra arguments is a simple extension of the one described before. Previously a new predicate was created for each phrase type, with two arguments to express how the input sequence was consumed. Now it is necessary to create a predicate with two more arguments than are mentioned in the grammar rules. By convention, these two extra arguments are added as the last arguments of the predicate (although this may vary between Prolog systems). Thus the grammar rule:

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

translates into:

```
sentence(X, S0, S) -->
        noun_phrase(X, S0, S1), verb_phrase(X, S1, S).
```

When we want to invoke goals involving grammar rules from the top level of the interpreter or from ordinary Prolog rules, we must explicitly add the extra arguments. Thus appropriate goals involving this definition of sentence would be:

```
?- sentence(X, [a,student,eats,a,cake],[]).
?- sentence(X, [every,bird,sings,and,pigs,can,fly],L).
```

Exercise 9.1: This may be a difficult exercise for some. Define in Prolog a procedure translate, such that the goal translate(X, Y) succeeds if X is a grammar rule of the type seen in previous sections, and Y is the term representing the corresponding Prolog clause.

Exercise 9.2: Write a new version of phrase that allows grammar rules with extra arguments, so that one can provide goals such as:

```
?- phrase(sentence(X), [the,man,sings]).
```

9.5 Adding Extra Tests

So far in our parser, everything mentioned in the grammar rules has had to do with how the input sequence is consumed. Every item in the rules has had something to do with those two extra argument positions that are added by the grammar rule translator. So every goal in the resulting Prolog clause has been involved with consuming some amount of the input. Sometimes we may want to specify Prolog goals that are not of this type, and the grammar rule formalism allows us to do this. The convention is that any goals enclosed inside curly brackets {} are to be left unchanged by the translator.

Let us look at some examples of where it would be beneficial to use this facility, in improving the "dictionary" of the parser, that is, the parser's knowledge about words of the language. First, consider the overhead involved in introducing a new word into the program with both sets of extra arguments. If we wished to add the new noun banana, for instance, we would have to add at least the rule:

```
noun(singular, noun(banana)) --> [banana].
```

which amounts to:

```
noun(singular, noun(banana), [banana|S], S).
```

in ordinary Prolog. This is a lot of information to specify for each noun, especially when we know that every noun will only occupy one element of the input list and will give rise to a small tree with the functor noun. A much more economical way would be to express the common information about all nouns in one place and the information about particular words somewhere else. We can do this by mixing grammar rules with ordinary Prolog. We express the general information about how nouns fit into larger phrases by a grammar rule, and then the information about what words are nouns in ordinary clauses. The solution that results looks like:

```
noun(S, noun(N)) --> [N], {is_noun(N, S)}.
```

where the normal predicate is_noun can be provided to express which words are nouns and whether they are singular or plural:

```
is_noun(banana, singular).
is_noun(bananas, plural).
is_noun(man, singular).
```

Let us look carefully at what this grammar rule means. It says that a phrase of type noun can take the form of any single word N (a variable is specified in the list) subject to a restriction. The restriction is that N must be in our is_noun collection, with some plurality S. In this case, the plurality of the phrase is also S, and the parse tree produced consists just of the word N underneath the node noun. Why does the goal is_noun(N,S) have to be put inside curly brackets? Because it expresses a relationship that has nothing to do with the input sequence. If we were to leave out the curly brackets, it would be translated to something like is_noun(N, S, S1, S2), which would never match our clauses for is_noun. Putting it inside the curly brackets stops the translation mechanism from changing it, so that our rule will be correctly translated to:

```
noun(S, noun(N), [N|Seq], Seq) :- is_noun(N, S).
```

In spite of this change, our treatment of individual words is still not very elegant. The trouble with this technique is that we will have to have to write two is_noun clauses for every new noun that is introduced — one for the singular form, and one

for the plural form. This is unnecessary, because for many nouns the singular and plural forms are related by a simple rule:

> If X is the singular form of a noun, then the word formed by adding an "s" on the end of X is the plural form of that noun.

We can use this rule about the form of nouns to revise our definition of noun. The revisions will give a new set of conditions that the word N must satisfy in order to be a noun. Because these conditions are about the internal structure of the word, and do not have anything to do with the consumption of the input sequence, they will appear within curly brackets. We are representing English words as Prolog atoms, and so considerations about how words decompose into letters translate into considerations about the characters that go to make up the appropriate atoms. So we will need to use the predicate atom_chars in our definition. The amended rule looks as follows:

```
noun(plural, noun(RootN)) -->
        [N],
        {atom_chars(N, Ptname),
        append(RootN, [s], Ptname),
        atom_chars(RootN, Singname),
        is_noun(RootN, singular)}.
```

Of course, this expresses a general rule about plurals that is not always true (for instance, the plural of "fly" is not "flys"). We will still have to express the exceptions in an exhaustive way.[1] We need now only specify is_noun clauses for the singular forms of regular nouns. Note that under the above definition the item inserted into the parse tree will be the "root" noun, rather than the inflected form. This may be useful for subsequent processing of the tree. Note also the syntax of curly brackets. Inside the curly brackets you can put any Prolog goal or sequence of goals that could appear as the body of a clause.

In addition to knowing about curly brackets, most Prolog grammar rule translators will know about certain other goals that are not to be translated normally. Thus it is not normally necessary to enclose "!"s or disjunctions (";") of goals involving the input sequence inside curly brackets.

[1] Some Prolog implementations support a version of atom_chars that produces from an atom a list of character codes (numbers) rather than characters (single element atoms). For such an implementation, the append goal here must be amended to specify the list containing the character code of s as its second argument. In some Prolog implementations, this list can be specified by putting the s inside double quotes.

9.6 Summary

We shall now summarise the syntax of grammar rules as described so far. We shall then indicate some of the possible extensions to the basic system and some of the interesting ways that grammar rules can be used. The best way to describe the syntax of grammar rules is by grammar rules themselves. So here is an informal definition. Note that it is not completely rigorous, because it neglects the influence of operator precedences on the syntax.

```
grammar_rule --> grammar_head, ['-->'], grammar_body.

grammar_head --> non_terminal.
grammar_head --> non_terminal, [','], terminal.

grammar_body --> grammar_body, [','], grammar_body.
grammar_body --> grammar_body, [';'], grammar_body.
grammar_body --> grammar_body_item.

grammar_body_item --> ['!'].
grammar_body_item --> ['{'], prolog_goals, ['}'].
grammar_body_item --> non_terminal.
grammar_body_item --> terminal.
```

This leaves several items undefined. Here are definitions of them in English. A non_terminal indicates a kind of phrase that may occupy part of the input sequence. It takes the form of a Prolog structure, where the functor names the category of the phrase and the arguments give extra information, like the number class, the meaning, etc. A terminal indicates a number of words that may occupy part of the input sequence. It takes the form of a Prolog list (which may be [] or a list of any determinate length). The items of the list are Prolog items that are to match against the words as they appear in the order given. prolog_goals are any Prolog goals. They can be used to express extra tests and actions that constrain the possible analysis paths taken and indicate how complex results are built up from simpler ones.

When translated into Prolog, prolog_goals are left unchanged and non_terminals have two extra arguments inserted after the ones that appear explicitly, corresponding to the sequence provided to, and the sequence left behind by, the phrase. The terminals appear within the extra arguments of the non_terminals. When a predicate defined by grammar rules is invoked at the top level of the interpreter or by an ordinary Prolog rule, the two extra arguments must be provided explicitly.

The second rule for grammar_head in the above mentions a kind of grammar rule that we have not met before. Up to now, our terminals and non-terminals have only been defined in terms of how they consume the input sequence. Sometimes we might like to define things that insert items into the input sequence (for other rules to find). For instance, we might like to analyse an imperative sentence such as:

Eat your supper.

as if there were an extra word you inserted:

You eat your supper.

It would then have a nice noun phrase/verb phrase structure, which conforms to our existing ideas about the structure of sentences. We can do this by having a grammar that looks in part like:

```
sentence --> imperative, noun_phrase, verb_phrase.

imperative, [you] --> [].
imperative --> [].
```

There is only one rule here that deserves mention. The first rule for imperative actually translates to:

```
imperative(L, [you|L]).
```

So this involves a sequence being returned that is longer than the one originally provided. In general, the left-hand side of a grammar rule can consist of a non-terminal separated from a list of words by a comma. The meaning of this is that in the parsing, the words are inserted into the input sequence after the goals on the right-hand side have had their chance to consume words from it.

Exercise 9.3: The definition given for grammar rules, even if made complete, would not constitute a useful parser, given a sequence of tokens as its input. Why?

9.7 Translating Language into Logic

To give an indication of how DCGs can be used to compute more complex analyses of language, we present an example (taken from Pereira and Warren's paper in the journal *Artificial Intelligence* Volume 13) of grammar rules used to obtain the meaning of sentences directly, without an intermediate parse tree. The following rules translate from (a restricted number of) English sentences into a representation of their meaning in Predicate Calculus. For a description of Predicate Calculus and our notation for it, the reader is referred to Chapter 10. As an example of the program at work, the meaning obtained for "every man loves a woman" is the structure:

```
all(X, (man(X) -> exists(Y, (woman(Y) & loves(X, Y)))))
```

Here are the grammar rules:

```
?- op(500,xfy,&).
?- op(600,xfy,->).
```

```
sentence(P) -->
        noun_phrase(X, P1, P), verb_phrase(X, P1).

noun_phrase(X, P1, P) -->
        determiner(X, P2, P1, P),
        noun(X, P3),
        rel_clause(X, P3, P2).

noun_phrase(X, P, P) --> proper_noun(X).

verb_phrase(X, P) -->
        trans_verb(X, Y, P1), noun_phrase(Y, P1, P).

verb_phrase(X, P) --> intrans_verb(X, P).

rel_clause(X, P1, (P1&P2)) -->
        [that], verb_phrase(X, P2).
rel_clause(_, P, P) --> [].

determiner(X, P1, P2, all(X,(P1 -> P2))) --> [every].
determiner(X, P1, P2, exists(X, (P1&P2))) --> [a].

noun(X, man(X)) --> [man].
noun(X, woman(X)) --> [woman].

proper_noun(john) --> [john].

trans_verb(X, Y, loves(X,Y)) --> [loves].

intrans_verb(X, lives(X)) --> [lives].
```

In this program, arguments are used to build up structures representing the meanings of phrases. For each phrase, it is the last argument that actually specifies the meaning of that phrase. However, the meaning of a phrase may depend on several other factors, given in the other arguments. For instance, the verb lives gives rise to a proposition of the form lives(X), where X is something standing for the person who lives. The meaning of lives cannot specify in advance what X will be. The meaning has to be applied to some specific object in order to be useful. The context in which the verb is used will determine what this object is. So the definition just says that, for any X, when the verb is applied to X, the meaning is lives(X). A word like every is much more complicated. In this case, the meaning has to be applied to a variable and two propositions containing that variable. The result is something that says that, if substituting an object for the variable in the first proposition yields something true then substituting that same object for the variable in the second proposition will also yield something true.

Exercise 9.4: Read and understand this program. Try running the program, giving it goals like


```
?- sentence(X, [every,man,loves,a,woman],[]).
```

What meaning does the program generate for the sentence "every man that lives loves a woman", "every man that loves a woman lives"? The sentence "Every man loves a woman" is actually ambiguous. There could either be a single woman that every man loves, or there could be a (possibly) different woman that each man loves. Does the program produce the two possible meanings as alternative solutions? If not, why not? What simple assumption has been made about how the meanings of sentences are built up?

9.8 More General Use of Grammar Rules

The grammar rule notation can be used more generally to hide an extra pair of arguments used as an accumulator or difference structure. That is, apart from the handling of terminals (actual words in the list), the two extra arguments added by the grammar rule translation mechanism can be used to track the situation as regards any single piece of information which changes as the Prolog computation proceeds. Thus a more neutral reading of:

```
noun_phrase(X, Y) :- determiner(X, Z), noun(Z, Y).
```

would be something like "noun_phrase is true in the situation characterised by X if determiner is true in situation X and, in the situation that results after that (Z), noun is true. The situation after noun_phrase is the same as that resulting from the noun (Y)." For grammar rules, the "situation" is usually a list of words that remains to be processed. But we'll see below that there are other possibilities.

The occurrence of a terminal in a grammar rule marks a transition from one "situation" to another. For the usual use of grammar rules, this is the transition from having some list of unprocessed words to having the same list minus its first word. All changes of "situation" in the end reduce to sequences of changes of this kind (the only way we can move through the list of words is to repeatedly find terminals as specified in the grammar rules).

In order to generalise the use of grammar rules, it is useful to define a predicate that says how a terminal induces a transition from one situation to another. By providing different definitions for this predicate, we can make our grammar rules perform as usual or make them do something different. This predicate is often called 'C'/3 (note that quotes are required because the C is upper case), and its definition for normal grammar rules is as follows:

```
% 'C'(Prev, Terminal, New)
%
```

```
% Succeeds if the terminal Terminal causes a transition from
% situation Prev to situation New

'C'([W|Ws], W, Ws).
```

That is, if the situation (list of unused words) is [W|Ws] and the terminal W is specified as the next thing in the current grammar rule, we can move to a new situation where the list of unused words is just Ws.

In the translation into Prolog, terminals in grammar rules can be expressed in terms of 'C' rather than directly in terms of what they mean for the list arguments. Indeed, many Prolog systems translate grammar rules in terms of 'C', thus producing for:

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

the translation

```
determiner(In,Out) :- 'C'(In,the,Out).
```

rather than the:

```
determiner([the|S],S).
```

that we have shown above. Given the definition of 'C' shown above, these two Prolog definitions for determiner always produce exactly the same answers (you might want to convince yourself of this!). So actually when you are using grammar rules in the normal way, you don't need to know which of these translation methods is used.

Giving 'C'/3 a modified definition allows grammar rules to be used for maintaining a record of something other than a shrinking list through the computation. For instance, if one is computing the length of a list, one can maintain a record of the number of items encountered so far. Here is a version of the code for this in section 3.7 re-expressed using grammar rules:

```
listlen(L, N) :- lenacc(L, 0, N).

lenacc([]) --> [].
lenacc([H|T]) --> [1], lenacc(T).
```

In this situation, encountering the terminal 1 causes 1 to be added to the total so far. So the definition we want for 'C' is:

```
'C'(Old, X, New) :- New is Old + X.
```

As another example, the computation of a parts list in section 3.7 can be recast as maintaining an increasing list of parts found so far. Here is what this could look like using grammar rules:

```
partsof(X, P) :- partsacc(X, [], P).
```



```
partsacc(X) --> [X], {basicpart(X)}.
partsacc(X) --> {assembly(X SubParts)}, partsacclist(Subparts).

partsacclist([]) --> [].
partsacclist([P|Tail]) --> partsacc(P), partsacclist(Tail).
```

In this case, the appropriate definition for 'C' is:

```
'C'(Old, X, [X|Old]).
```

Important note: The phrase/2 predicate makes the assumption that one is interested in the final situation represented during the computation being []. If you change the definition of 'C', you may need to define your own modified version of phrase/2 that does not make this assumption.