CHAPTER 4

GRAMMARS

4.1 Grammar as knowledge representation **100**
4.2 Words, rules and structures **104**
4.3 Representing simple grammars in Prolog **110**
4.4 Subcategorization and the use of features **115**
4.5 Definite Clause Grammars **127**
4.6 Classes of grammars and languages **132**

In ATNs, we have an immensely flexible medium for constructing natural language analyzers, but this flexibility is bought at the price of a procedural character that loses descriptive clarity and imposes a dependence on the type of application. In this chapter, we consider the advantages of basing a language processing system on a declarative description, a grammar, of the language concerned, which is neutral as regards the desired application. The concepts of context-free phrase structure grammars (CF-PSGs) are introduced, the notation of PATR being used to show how phenomena like subcategorization and unbounded dependencies can be elegantly described in a declarative formalism. We then go on to look at Prolog's own built-in (usually) grammar notation – definite clause grammar (DCG). Finally, the relation of CF-PSGs to other classes of grammars is considered.

4.1 Grammar as knowledge representation

A description of a recognizer or parser for a language is in some sense a description of that language, but it need not by any means be a perspicuous one. Moreover, it may well be an implementation-specific definition, but implementations – even implementations of a programming language that is thought to be well understood – can differ significantly and unexpectedly. It is for this reason that computer scientists have turned in recent years away from procedural definitions of the semantics for programming languages and towards the declarative descriptions of denotational semantics. Rather similar considerations hold when we consider writing programs that process natural language input: both syntactically and semantically, we need to have a secure definition of the natural language (or an approximation to a natural language) that we are processing if we are to have any idea how the system should behave under a wide range of conditions.

A language can be regarded as a set whose membership is precisely specifiable by rules. The set of compound linguistic expressions in a natural language is not finite, so we cannot list them all (cf., 'a slow machine', 'a very slow machine', 'a very very slow machine', ...). As far as is known, no natural language is a finite language. The range of constructions that make a language infinite is typically rather large. In English, a word like 'and' permits an unbounded number of phrases to be combined together and relative clauses can contain verb phrases which can contain noun phrases which can contain relative clauses which ... .

What we need are formal (that is, mathematical) systems that define the membership of the infinite sets of linguistic expressions and assign a structure to each member of these sets. Such formal systems are *grammars*. From this perspective, the various kinds of transition networks that we have looked at in the preceding chapters are grammars. However, the word 'grammar' is often used in a rather more restricted sense in current natural language work to refer to formal systems that meet not only the criterion just noted, but are in addition subject to two further criteria:

- Firstly, grammars, in the sense discussed in this chapter, are expressed in a declarative grammar formalism that only contains information about which objects combine together and what the properties of the resultant object are, a formalism that contains no extraneous procedural information about how to put the objects together, such as the control information implicitly encoded in transition networks.
- Secondly, grammars, as we shall present them, transparently provide each legal string with an implicit structural description, without the necessity of explicit structure-building annotations, as required in an ATN, for example.

Like FSTNs and RTNs, but unlike many ATNs, the grammars we shall present directly characterize the order of elements in the string, as opposed to attempting to reconstruct some hypothesized underlying order. And, like RTNs, our grammars will use simple recursion to manage the structural complexity. Grammars as we shall present them are declarative and, for the most part, based on a decomposition of syntactic categories (roughly 'parts of speech') into components known as features. One of the merits of such formalisms will only become apparent when we look at semantics in Chapter 8: grammars of this kind support a compositional approach to meaning, one in which every well-formed expression has a meaning of its own, a meaning that has been composed from the meaning of the subexpressions that make it up. In such a context, the syntactic structure imposed by the grammar on an expression is a key element in the determination of its meaning.

From the perspective of NLP, the study of grammar is a branch of knowledge representation: a grammar is simply a way of representing certain aspects of what we know about a language that is explicit and formal enough to be understood by a machine. Consider by analogy, a much simpler area of knowledge representation, the representation of mass: this involves mathematical questions (which is the appropriate number system to use – integers, reals, rationals, complex numbers, ...?), notational questions (do we use arabic numerals, roman numerals, points in a graph, ...?) and detailed descriptive questions (is **mass(ingot93) = 128kg** true, accurate, accurate enough, ...?). In this chapter, we will examine these three kinds of question as they apply to the grammars of natural languages.

One important mathematical question that has interested linguists, off and on, for the last 30 years is the question of just how powerful a formal system is needed to describe any natural language. We have already seen one example of this issue in Chapter 2 where we noted the inability of FSTNs to recognize exactly the strings in the language a^nb^n – that is, the language consisting of strings containing some number of *a*s followed by the same number of *b*s. And the RTNs considered in Chapter 3 can recognize exactly the strings of a^nb^n but not those of $a^nb^nc^n$. There are a number of good reasons why a computational linguist should care about this question. If, for example, we are using an RTN for our language description and we encounter an $a^nb^nc^n$ construction in the language in question, then we will know that time spent trying to get our RTN to recognize the construction will be time wasted. Or suppose that we had access to hardware that would handle FSTNs ultrafast and observed that in actual occurrences of a^nb^n constructions, the value of n never exceeded 3; then we might decide to compile our RTN descriptions down into FSTNs subject to an $n = 3$ upper bound (such a compilation is possible for any given finite upper bound on the value of n). Knowledge of the underlying mathematics can help to avoid wasting time and, sometimes, even suggest

shortcuts. We will consider some of the general mathematical questions to do with power in a bit more detail in the final sections of this chapter.

The academic linguist's primary criterion in evaluating a type of grammar has always been its ability to capture significant generalizations within the grammar of a language and across the grammars of different languages. However, expressing significant generalizations is largely a matter of notation, and classes of grammars, taken as sets of mathematical objects, have properties that are theirs independently of the notations that might be used to define them. Thus, a class of grammars determines the set of languages that can be described by its members, the set of structural analyzes that can be assigned to sentences in these languages, and so on.

Like the academic linguist, the computational linguist needs to know what natural languages are really like, mathematically speaking. But, in addition, it is useful for the computational linguist to know what languages are roughly like, or mostly like, and whether particular languages (English, say, or Japanese) exhibit certain mathematical properties. These latter questions are of little interest to the academic linguist. Suppose, for example, that Swahili has certain features that show it to be a type Y language, mathematically. But suppose further that these features are statistically rather uncommon in every day Swahili usage, so that 99.5% of such usage can be parsed under the assumption that Swahili is of a mathematically more restrictive type and has a computationally more straightforward type X grammar. Suppose, finally, that we have a working parser based on a type X grammar for Swahili which handles 85% of the input it is offered and which, for numerous practical reasons (ill-formed input, dialectal variations, non-standard spellings, idiosyncratic stylistic forms, ...), cannot be further improved (85% is actually quite a good figure in a practical context). It is clear that under such conditions, a switch to a parser based on a type Y grammar would simply not be worth the effort since, ceteris paribus, it could effect at best an improvement from 85% to 85.5%.

Turning now to notational issues, the design and choice of grammar formalisms for NLP is a topic that has attracted considerable attention in the 1980s and much progress has been made. At least the following general criteria are relevant:

- linguistic naturalness,
- mathematical power, and
- computational effectiveness.

Firstly, those who use such formalisms need a notation that allows and encourages them to encode their linguistic descriptions in a manner that is easy to understand and modify, accords with the way they think about language, and expresses the relevant generalizations about the domain. For example, if all tensed verbs in a language appear in VP-initial position, then the formalism should allow us to say exactly that, and not force us into making 40 statements, one for each subclass of verbs, or into some cumbrous circumlocution making reference to all words showing some disjunction of particular endings, a circumlocution that just happens to pick out all the tensed verbs. Secondly, if we decide that natural languages are type Y, mathematically speaking, and we wish our grammars to take this fact into account, then the grammar description language will need to be able to express type Y grammars.

As will be seen in the final sections of this chapter, some notational restrictions on grammar formalisms can have the effect of seriously limiting the class of grammars that can be expressed. Conversely, apparently minor notational changes can radically increase the potential mathematical power of the systems characterized. We may not want this to happen, either because we wish to inhibit users of the formalism from devising unnatural analyzes, or for reasons of computational effectiveness, to which we now turn.

A grammar formalism used by an academic linguist is usually only read by other academic linguists. But a computational grammar formalism, like a programming language, has to be read and understood by both humans and machines. Furthermore, given the typical goals of NLP work, we want machines to be able to understand and employ the formalism in realistic amounts of time. Indeed, the issues that arise in the design of grammar formalisms are exactly those that arise in the design of any specialist declarative computer language for knowledge representation in some particular domain.

Some existing grammar formalisms were proposed to express linguistic theories. With such formalisms, the self-imposed limitations are often intended as empirical claims about the nature of natural languages. Others were designed for use as tools by the academic or computational linguist, and these are normally free of intended expressive limitations. The formalisms we shall use in this chapter (and subsequently) to represent what are essentially context-free phrase structure grammars are definite clause grammar (DCG) and PATR, and both these formalisms belong to the latter class. PATR has, in recent years, become a potential lingua franca for NLP work, and many other grammatical formalisms can be compiled into it.

All the kinds of grammar that have been used in computational linguistics have employed, in one form or another, the following:

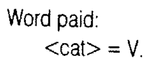
- A representation for syntactic categories or 'parts of speech'.
- A data type for words (and hence a lexicon, dictionary or word list).
- A data type for syntactic rules.
- A data type for syntactic structures.

A grammar as a whole can be viewed as a use of a particular kind of data type composed of the first three items. A parser, then, is an algorithm that takes a grammar, together with a string, and attempts to return one or more instances of the syntactic structure data type. Thus, a complete grammar formalism provides, at least, a language for specifying syntactic categories, a language for representing lexical entries, a language in which to write rules (possibly rules of more than one type) and a language for exhibiting syntactic structures. These languages may be distinct, or two or more of them may collapse.

4.2 Words, rules and structures

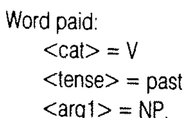
A *lexicon* is minimally a list of words that associates each word with its syntactic properties. The most important of these properties is its (gross) syntactic category – for example, whether the word is a verb or a noun. In addition, depending on the sophistication of the overall grammar, the lexicon will contain information as to the subcategory of the word (such as whether a particular verb is transitive or intransitive), other syntactic properties (such as the gender of a noun in a language that makes gender distinctions), and perhaps also morphological and semantic information. In the very simplest grammars, then, a lexical entry might look like this (using PATR notation):

```
Word paid:
    <cat> = V.
```



which simply tells us that 'paid' is a word whose category (cat) is verb (V), whereas a slightly more sophisticated grammar might require a lexical entry like this:

```
Word paid:
    <cat> = V
    <tense> = past
    <arg1> = NP.
```



which tells, in addition, that the verb is in the past tense and that it is transitive – how <arg1> = NP comes to mean transitive will be explained later. Note that a feature description is always *partial* – when a feature is not mentioned, the description is consistent with it having *any* value.

The character of syntactic rules varies considerably depending on the precise character of the theory of grammar involved, and often such theories permit the use of more than one kind of rule. One very frequently employed type of syntactic rule is the *phrase structure rule*, and most current computational linguistic frameworks presuppose the existence of such rules. A

phrase structure rule simply tells what a particular syntactic category (the 'left-hand side' – LHS) can be composed of (the 'right-hand side' – RHS). Thus, S → NP VP tells us that a sentence can consist of a noun phrase followed by a verb phrase, while VP → V NP tells us that a verb phrase can consist of a verb followed by a noun phrase. If you treat the arrow → as a convenient abbreviation for the expression 'can consist of' when looking at phrase structure rules, then you will not go far wrong. Phrase structure rules can also be used to introduce lexical items; thus, a rule like V → paid is just an alternative way of representing the information we encoded above as:

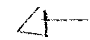
```
Word paid:
    <cat> = V.
```

In principle, then, we could at this stage, if we wished, collapse our rules and our lexicon into a single component, since both components use phrase structure rules in this example. In practice, however, we will sometimes treat lexical entries as grammatical rules and sometimes consider them as a separate component, depending on which seems clearest at a given moment.

So far, we have discussed grammar in the abstract without ever exhibiting an example of one. In this section and the next, we will look in some detail at what grammars look like and how they work. To start with, let us consider a very straightforward grammar (Grammar1) employing four syntactic categories; namely, noun phrase (NP), sentence (S), verb phrase (VP) and verb (V). We will supply the grammar with only three rules: one telling us that a sentence can consist of a noun phrase followed by a verb phrase, another which lets a verb phrase consist of a verb and a noun phrase, and a third which lets a verb phrase consist of a verb standing on its own. We also need a lexicon to supply us with some ready-made verbs and noun phrases. We will imagine that we are beginning to build a natural language front end to a hospital database, so all our lexical items will be drawn from this domain. The lexicon tells us that 'Dr Chan', 'nurses', 'MediCenter', and 'patients' are all noun phrases while 'died' and 'employed' are both verbs.

EXAMPLE: Grammar1

```
Rule {simple sentence formation}
  S → NP VP.
Rule {transitive verb}
  VP → V NP.
Rule {intransitive verb}
  VP → V.
```



```
Word Dr Chan:
    <cat> = NP.
Word nurses:
    <cat> = NP.
Word MediCenter:
    <cat> = NP.
Word patients:
    <cat> = NP.
Word died:
    <cat> = V.
Word employed:
    <cat> = V.
```

Grammars such as Grammar1 are called *context-free phrase structure grammars (CF-PSGs)*. The key features of a CF-PSG are the employment of a finite set of grammatical categories and a finite set of rules for specifying how LHS categories can be realized as sequences of RHS elements. An RHS element may then be either a category or a particular symbol of the language. When a CF-PSG rule specifies that an LHS category can be realized as a particular RHS, this realization is deemed to be possible regardless of the context in which the LHS category appears. The rule does not make any restriction on the context in which this can happen – hence, the term 'context free'. There are a variety of possible notations for CF-PSGs, the Backus–Naur Form used for programming language grammars being one, but we will use (a subset of) the PATR notation for Grammar1.

We have now completely defined Grammar1. But what is it good for? What can it do? Well, a grammar has two functions. The first is to define the sets of grammatical strings of words in the language under consideration (or, more realistically, part of that language). Such strings are said to be 'generated' by the grammar. This use of the word 'generate' does not imply the existence of any actual procedure that produces sentences; it indicates merely that certain strings are allowed as grammatical by the grammar. So, in the case of a natural language like English, a grammar should define the set of grammatical sentences, the set of grammatical verb phrases, and so on. In defining the sets of grammatical strings it will, by implication, also define the ungrammatical strings, since an ungrammatical string is simply a string that is not grammatical – that is, a string that is not 'generated' by the grammar. The second function the grammar has is to associate one or more structures with each grammatical string. The syntactic structure of a phrase can be thought of as a kind of justification that the phrase is grammatical. It displays the various parts of

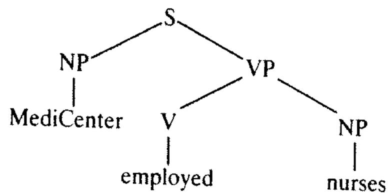


Figure 4.1 Phrase structure tree.

the phrase and the ways in which the combinations of parts are licenced by the rules of the grammar. As we will see, this breaking up of a phrase into parts is a key factor in the computation of the semantics of the phrase. Typically, if a given string has two different structures, then it will also have two different meanings, and hence be ambiguous. Part of the job of the grammar, then, is to make correct predictions about this kind of structural ambiguity.

What form do these structures take? The answer, in mathematical parlance, is *directed acyclic graphs (DAGs)*: all contemporary grammatical frameworks, without exception, employ some kind of directed acyclic graph to represent structure. We will not delve into graph theory here, however, and directed acyclic graphs will anyway receive more attention in Chapter 7. Conveniently, most, but by no means all, contemporary linguists have elected to use one particular kind of directed acyclic graph – namely, the tree – for their structural descriptions. And a tree is what it sounds like: it has a root, it has branches and it terminates in leaves. But computational linguists, like genealogists, conventionally exhibit their trees upside down, with the root poking into the sky and the leaves on the ground. Figure 4.1 shows such a tree.

How does Grammar1 define sets of strings and associate such strings with structures like the one shown in Figure 4.1? Consider the top of the tree: here we have an S which has as daughters an NP and a VP, and nothing else, appearing in that order. The first rule of Grammar1 says that an S can consist of an NP followed by a VP, and that is exactly what we have here. So, the grammar legitimates this part of the structure. If we turn our attention now to the VP, then again we find that the grammar contains a rule that corresponds to this part of the structure. Finally, we look in the lexicon and find that 'MediCenter' and 'nurses' are both NPs, as the tree claims, and that 'employed' is a V. So, every substructure in the tree corresponds to some rule in the grammar, and the tree is thus admitted by the grammar. Since the tree is a legitimate structure, it follows that the string of words made up of the leaves of the tree – namely, 'MediCenter employed nurses' – is, according to Grammar1, a grammatical expression of category S – that is, a sentence. This is not the only string of words that

Grammar1 will admit as a sentence – there are 39 more such strings, including 'Nurses died', 'Dr Chan employed patients', and so on.

We have represented our illustrative tree graphically here, but such diagrams are not, given the limitations of present computers, a convenient data structure to use for NLP. The simplest way to handle trees is to manipulate them as lists where the first element is the root category and subsequent elements are the daughter subtrees in order of occurrence. Thus, our tree would be represented as follows:

```
[s,[np, 'MediCenter'], [vp, [v,employed], [np, nurses]]]
```

This kind of list would appear in a 'pretty-printed' format as follows:

```
[s
  [np
    'MediCenter']
  [vp
    [v
       employed]
    [np
       nurses]]]
```

The general principle used here is that a tree is represented by a list whose first element is the top label and whose subsequent elements are the immediate subtrees in order. These elements are then themselves lists, representing the subtrees according to the same convention. In this scheme, a leaf node of a tree is treated as a tree with a label but no immediate subtrees. Thus, it will be represented by a list with one element – the label. An alternative and equivalent representation of a tree is as a term:

```
s(np('MediCenter'), vp(v(employed), np(nurses)))
```

We will conclude this section with some comments about the appropriate way to evaluate a particular grammar for a (fragment of a) language, given a grammar formalism and an underlying mathematical interpretation for that formalism. There are basically three empirical criteria:

(1) Does it undergenerate?
(2) Does it overgenerate?
(3) Does it assign appropriate structures to the strings that it generates?

Once again, we are using the word 'generate' in a neutral sense here, simply discussing the adequacy of the grammar as a description of a set of strings and ignoring the procedural complications that would be involved if we wished to produce actual example sentences from the description.

A grammar for English undergenerates if there are some syntactically well-formed expressions of English to which the grammar assigns no structure. But the sets of rules that grammarians propose are rarely intended to generate the whole of a natural language and so questions of undergeneration are normally considered relative to the goals of the grammar. If the grammar is intended as a set of rules to handle all English relative clauses and yet it fails to assign a structure to 'that saw you' in 'the man that saw you', then the grammar is clearly empirically inadequate for reasons of undergeneration. In a computational context, undergeneration by the grammar is not necessarily a problem, if the goal is to *produce* appropriate language; on the other hand, it could prove fatal if the goal is to *recognize* or *understand*.

A grammar overgenerates if it legitimates strings that cannot be construed as grammatical expressions of the language in question, given the category assigned to them by the grammar. This appears simpler than the undergeneration criterion, since the grammarian's goals do not seem to be bound up with the criterion. However, two factors serve to make it less straightforward than it at first appears. Firstly, our intuitions about what is or is not grammatical are often hard to disentangle from our judgements about what is meaningful; for example, is 'stones bet stones stones stones' ungrammatical, or merely unlikely to have a useful meaning? And, secondly, the grammar may decide grammaticality on the basis of more than one component, so the fact that one component fails to forbid some string does not mean that the grammar as a whole will fail in this way. Of course, if all the components of the grammar are fully specified, then there should be no problem of evaluation. But often, in academic linguistic work, one or more syntactic components exist in name only (logical form, interpretative rules, and so on). This makes it difficult or impossible to evaluate the empirical adequacy of the rule components that are fully specified. In a computational context, overgeneration by the grammar is not necessarily a problem if the goal is to *recognize* or *understand* well-formed language. On the other hand, it could prove fatal if the goal is to *produce* well-formed language.

The question of whether a grammar assigns the correct structures is likewise a subtle one. Natural language expressions do not come to us with their structures ready marked: we have to infer the relevant structures indirectly by considering, for instance, what other phrases we could substitute for a given subphrase while maintaining grammaticality.

In addition to these empirical criteria, standard scientific considerations such as simplicity and generality apply to grammars in much the same way as they do to any other theories about natural phenomena. Other things being equal, a grammar with seven rules is to be preferred to one with 93 rules. And, other things being equal, a grammar for a fragment of English that can be converted into a grammar for the corresponding fragment of French with a few minor changes is to be preferred to a

grammar for the English fragment that bears no relation whatsoever to its French counterpart. Likewise, a grammar for Swedish relative clauses that can be extended to handle a class of Swedish questions with the addition of a couple of rules and a feature is to be preferred to one that requires a complete new set of parallel rules and features to encompass the questions.

Exercise 4.1 Convert the RTN recognizer program into a parser that returns phrase structure trees represented as in the text. [*intermediate*]

Exercise 4.2 Write programs that will take a set of rules and a lexicon in a Prolog format and generate:

(a) example strings of the permitted language,
(b) example phrase structure trees represented as lists, and
(c) example phrase structure trees represented as terms.

For generating example strings, you will need to use difference lists. What would be required to make your programs generate *randomly*? [*intermediate*]

Exercise 4.3 Justify the claim that Grammar1 admits precisely 40 strings as sentences. [*easy*]

Exercise 4.4 Every CF-PSG can be converted into an RTN that accepts exactly the same set of strings as the grammar defines. Write a compiler that will take an arbitrary grammar in the PATR format used by Grammar1, or a similar format of your devising, and convert it into an equivalent RTN represented in the way shown in Chapter 3. [*intermediate*]

also see page 69

Exercise 4.5 The simplest way of doing the compilation of the last question will lead to RTNs with many redundant arcs. Write a program to optimize the resulting RTN by eliminating these redundant arcs. [*hard*]

Exercise 4.6 Every RTN can be mapped into a CF-PSG that generates the same language. Write a compiler that will take an arbitrary RTN represented in the way shown in Chapter 3 and convert it into a CF-PSG that generates the same language. What aspect of RTNs makes this exercise somewhat less trivial than the reverse compilation? [*hard*]

4.3 Representing simple grammars in Prolog

It is as straightforward to represent a CF-PSG in Prolog data structures as it is to represent transition networks. One way is to have a predicate **rule**

which takes two arguments, for the LHS category and the sequence of categories appearing as the RHS (represented as a list). Here is how the rules of Grammar1 would appear in this representation:

```
rule(s, [np, vp]).
rule(vp, [v, np]).
rule(vp, [v]).
```

For lexical entries, we can adopt the same convention as with transition networks, introducing assertions like:

```
word(np, 'Dr Chan').
word(np, nurses).
word(np, 'MediCenter').
word(v, died).
```

A grammar is a formal set of rules describing what it is to be a string in a particular language. Prolog, as a language based on logic, is eminently suitable for writing formal descriptions and definitions of various kinds. It is therefore natural to consider expressing a grammar *directly* as a logical specification in Prolog. In fact, it is extremely simple to translate the little grammar that we have been using as an example into Prolog clauses that directly describe strings of words and define which ones are grammatical. As it happens, the resulting Prolog code can be used to recognize exactly the strings of words that the grammar claims to be grammatical and to generate example grammatical sentences. The translation of grammar rules into Prolog clauses just requires thinking carefully about what each rule is actually saying. Our sentence rule can be readily paraphrased as claiming that a string of words, Z, counts as a sentence if there is a string of words, X, that counts as a noun phrase, a string of words, Y, that counts as a verb phrase and that Z is merely Y appended to X. And this paraphrase can be more concisely expressed as a Prolog Horn clause:

```
s(Z) :-
  np(X),
  vp(Y),
  append(X, Y, Z).
```

The rule for verb plus object verb phrases is isomorphic:

```
vp(Z) :-
  v(X),
  np(Y),
  append(X, Y, Z).
```

The intransitive case is trivial:

```
vp(Z) :-
  v(Z).
```

We need a lexicon, of course, and this can be provided as follows:

```
np(['Dr Chan']).
np(['MediCenter']).
np([nurses]).
np([patients]).
v([died]).
v([employed]).
```

And that is it – we have a recognizer for a tiny fragment of English which can be invoked thus:

```
?-s([patients, died]).
?-s(['MediCenter', employed, nurses]).
```

Thanks to its wholly declarative character, we can use this very same code to do generation as well as recognition. Try the following:

```
?-s(Sentence).
```

Although elegant in comparison with the code that would be needed to perform the same task in a procedural language, this scheme for writing recognizers is less than ideal. Considerations of both efficiency and aesthetics suggest that we should eliminate the explicit **append** statements if we can. The difference list technique, which we have already exploited in our net traversal programs, allows us both to eliminate the offending **append** statements and to make the Prolog code close to isomorphic to the rules of the grammar whose language is recognized, thus simultaneously addressing our efficiency and aesthetic concerns. We recode the recognizer as follows:

```
s(X, Z) :-
  np(X, Y),
  vp(Y, Z).
vp(X, Z) :-
  v(X, Z).
vp(X, Z) :-
  v(X, Y),
  np(Y, Z).
np(['Dr Chan' | X], X).
np(['MediCenter' | X], X).
np([nurses | X], X).
np([patients | X], X).
```



```
v([died | X], X).
v([employed | X], X).
?-s([patients, died], [ ]).
?-s(['MediCenter', employed, nurses], [ ]).
```

The first rule here can be paraphrased as saying that we can find an s at the start of X (with Z being the portion of the string left over) if we can find an **np** at the beginning of X, leaving Y behind, and a **vp** at the beginning of Y, with Z left behind after that.

As before, we can use this very same code to do generation and recognition. Try the following:

```
?-s(Sentence, [ ]).
```

It is straightforward to convert the recognizer into a parser, simply by the addition of extra arguments. Each predicate in the program, which captures the notion of a particular grammatical category, is provided with an extra argument (appearing as the first argument, say) which describes (as a list) the structure of the phrase. In the logic translation of each phrase structure rule, the tree corresponding to the LHS category will be simply a list consisting of the category name followed by the trees of the RHS categories. For example:

```
s([s, NP, VP], X, Z) :-
  np(NP, X, Y),
  vp(VP, Y, Z).
np([np, nurses], [nurses | X], X).
```

Translated into English (and forgetting about the arguments concerned with portions of the string), the first clause says that one form of a sentence is an NP followed by a VP. If a sentence has this form, then its parse tree will be a list of the form **[s, NP, VP]**, where **NP** is the parse tree of the noun phrase and **VP** is the parse tree of the verb phrase.

The code for our difference list recognizer bears such a close relationship to the corresponding PSG that we might reasonably expect to be able to get from the latter to the former automatically. We would like to be able to get a program to convert clauses written essentially as phrase structure rules into clauses that would implement difference list recognition. Thus, our input to the program might be expressed thus:

```
s ---> np, vp.
np ---> [nurses].
```

(given an appropriate operator declaration for --->), where the annotated ('variabilized') difference list output we want is:

```
s(_1, _2) :-
  np(_1, _3),
  vp(_3, _2).
np([nurses | _4], _4).
```

This turns out not to be too difficult. We define a predicate **translate** as follows:

```
translate((LHS_in ---> RHS_in), (LHS_out :- RHS_out)) :-
  LHS_out =.. [LHS_in, S0, Sn],
  add_vars(RHS_in, S0, Sn, RHS_out).
```

This sets up templates for the input rule and output clause, forms the LHS of the output by wrapping a couple of difference list variables up with the LHS of the input and concludes by handing the RHS over to **add_vars**:

```
add_vars((RHS_in1, RHS_in2), S0, Sn, RHS_out) :- !,
  add_vars(RHS_in1, S0, S1, RHS_out1),
  add_vars(RHS_in2, S1, Sn, RHS_out2),
  combine(RHS_out1, RHS_out2, RHS_out).
```

If there is more than one item on the RHS of the input, then we split it into two at the first comma and recursively reapply **add_vars** to the two pieces, combining the results.

```
add_vars(RHS_in, S0, Sn, true) :-
  islist(RHS_in), !,
  append(RHS_in, Sn, S0).
```

If the RHS just consists of a list, and is thus a lexical entry, append the terminating difference list variable to it and then unify the resulting list with the initial difference list variable.

```
add_vars(RHS_in, S0, Sn, RHS_out) :-
  atom(RHS_in),
  RHS_out =.. [RHS_in, S0, Sn].
```

If RHS is just a single atom, then treat it the same way as the LHS symbol.

It only remains to define **combine**:

```
combine(true, RHS_out2, RHS_out2) :-!.
combine(RHS_out1, true, RHS_out1) :-!.
combine(RHS_out1, RHS_out2, (RHS_out1, RHS_out2)).
```

This just says that RHS results that unify with **true** can be ignored – otherwise you get what you expect.

Exercise 4.7 Add a clause to the recognizer just given in the text to allow acceptance of ditransitive verbs such as 'showed' in 'patients showed deference to Dr Chan'. [*easy*]

Exercise 4.8 Complete the code for the parser given in the text. [*easy*]

Exercise 4.9 Augment the PSG translator given in the text so that it can open a file, read in a set of terms using ---> which represent phrase structure rules and lexical entries in the manner exemplified, translate these terms into the difference list format and assert the results into the database. [*easy*]

Exercise 4.10 Can you redefine the PSG ⇒ Prolog recognizer translation by means of an FST? If so, do so. If not, explain why not. [*easy*]

Exercise 4.11 The translator given yields a recognizer as output. Modify it so that it produces a parser of the kind just illustrated – that is, one that builds up parse trees as lists. [*intermediate*]

4.4 Subcategorization and the use of features

As the reader may already have noticed, the ungrammatical strings, '*Dr Chan died patients' and '*MediCenter employed' are among those that Grammar1 claims to be sentences (linguists use '*' to mark strings that are intuitively ungrammatical in the language under consideration, in this case English). The problem here involves what linguists call 'subcategorization'. Although 'died' and 'employed' are both verbs, as Grammar1 claims, they belong in different subcategories of the class of verbs. Specifically, 'employed' is transitive and requires a following NP, whereas 'died' is intransitive and cannot tolerate a following NP. We could patch up Grammar1 by replacing V with two categories, IV (intransitive verb) and TV (transitive verb), and revising the VP rules and lexicon accordingly, but such an approach rapidly proliferates a host of distinct parts of speech once a larger class of sentences is considered.

Instead of pursuing the *ad hoc* solution just mentioned, we will turn to a more principled approach: one that employs syntactic features. Most, if not all, contemporary theories of grammar employ features, and the extent and sophistication of their use has grown massively in the 1980s, although their use in computational linguistics goes back to the late 1950s. In a modern feature–theoretic syntax, atomic categories such as NP and V are replaced by sets of feature specifications. Each feature specification

consists of a feature, say case, and a value for that feature, say accusative. The familiar names for categories such as NP and V can then be reintroduced as the value of a particular feature, which we shall call cat. In fact, we have already made this move in the lexical entries we have shown.

In the light of this, what we will do now is to rebuild Grammar1 using features, thus creating Grammar2, and solving the problem noted earlier with subcategorization. To make Grammar2 a little more interesting than it would otherwise be, we will also introduce a rule for coordination, which will allow us to illustrate the role of recursion in grammars. Having done that, we will then extend Grammar2 to Grammar3, and in so doing illustrate the descriptive power of feature–theoretic techniques on a range of syntactic phenomena.

We will employ just two features in Grammar2 – namely, cat(egory) and arg1(ument). cat will have as its values the labels that were used for categories themselves in Grammar1, but with one addition, C, which we will discuss later. arg1 will, for the moment, have just two values: 0 (that is, nothing) and NP. How are we to interpret these features and their values? Well, consider a category that has V as the value of its cat feature, which means that it is a verb, and NP as the value of its arg1 feature. We will interpret the latter to mean that it is the kind of verb that requires a following NP; that is, it is a transitive verb. A verb with 0 as its arg1 value will be a verb that requires nothing to follow; that is, it is an intransitive verb.

We now have the problem of how to write rules, given that we have moved away from unanalyzed (monadic) categories to bundles of featural information. To keep things clear, we will continue to write, for example:

```
Rule {simple sentence formation}
  S → NP VP.
```

But this must now be construed as elliptical for something that would be more verbosely expressed as follows:

```
Rule {simple sentence formation}
  X0 → X1 X2:
    <X0 cat> = S
    <X1 cat> = NP
    <X2 cat> = VP.
```

So, in this verbose version of our rule notation, we have place holders X0, X1 and X2 in the rule itself, and then a collection of equations such as <X0 cat> = S, which is to be read as saying 'the value of X0's category feature is S'. In the abbreviated rule notation, we simply substitute the value of cat for the place holder in the rule and in accompanying equations. Given these remarks, we can now exhibit the rules employed by Grammar2.

EXAMPLE: Grammar2

```
Rule {simple sentence formation}
  S → NP VP.

Rule {intransitive verb}
  VP → V:
    <V arg1> = 0.

Rule {single complement verbs}
  VP → V X:
    <V arg1> = <X cat>.

Rule {coordination of identical categories}
  X0 → X1 C X2:
    <X0 cat> = <X1 cat>
    <X0 cat> = <X2 cat>
    <X0 arg1> = <X1 arg1>
    <X0 arg1> = <X2 arg1>.


Word died:
    <cat> = V
    <arg1> = 0.

Word recovered:
    <cat> = V
    <arg1> = 0.

Word slept:
    <cat> = V
    <arg1> = 0.

Word employed:
    <cat> = V
    <arg1> = NP.

Word paid:
    <cat> = V
    <arg1> = NP.

Word nursed:
    <cat> = V
    <arg1> = NP.

Word and:
    <cat> = C.

Word or:
    <cat> = C.
```

The first rule looks the same as its counterpart in Grammar1 and performs exactly the same function. The second and third rules in Grammar2 perform the desired functions of the second and third rules in Grammar1, but, together with the lexical entries shown, they ensure that a transitive verb will be followed by a noun phrase, whereas an intransitive verb will be followed by nothing. Grammar2 thus makes the correct claims about the grammaticality of the following examples:

Nurses died.
*Nurses died patients.
*MediCenter employed.
MediCenter employed nurses.

The grammar will assign structures to the first and last examples, but no structures are available for the second and third. The verbs 'die' and 'employ' belong to two featurally distinct categories in Grammar2: the former may not be followed by a noun phrase, the latter must be followed by a noun phrase. Hence, as far as Grammar2 is concerned, the second and third examples are ungrammatical.

The fourth and final rule is a schema, which takes us beyond the domain covered by Grammar1. This schema introduces the coordinate construction and says that a given category can consist of two further instances of the same category separated by an item of category C, which will turn out to be realized as 'and' or 'or'.

Turning now to the lexicon for Grammar2, while we can simply carry over the NP entries from Grammar1, we do need to revise the V entries, and add a couple of C entries. Notice that a few extra verbs have been added to enhance the plausibility of the examples given.

Apart from the introduction of features, which at this stage may appear to have been of much more trouble than it is worth, Grammar2 appears very little different from Grammar1. But there is one very fundamental difference between them. Grammar1 claimed that exactly 40 strings of words were grammatical instances of the category S, whereas Grammar2 admits infinitely many strings of words as grammatical instances of S. How can this be? The answer lies in the coordination schema we have introduced into Grammar2. This allows any category to split into two instances of the same category. These new instances may themselves split, and so on. This aspect of the grammar provides an example of recursion in syntactic rules (see the examples following Grammar3 for a rather different example of this phenomenon). Grammar2 permits recursion, whereas Grammar1 did not. This should become clearer from the example in Figure 4.2 which exhibits one of the infinitely many trees that Grammar2 admits as grammatical. As can be seen, we have two sentences coordinated, the second sentence itself containing a coordinate verb phrase. Thus, this example illustrates how the S category can reintroduce the S category, and how the

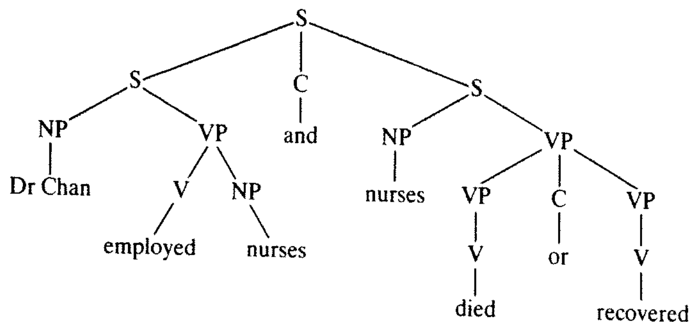


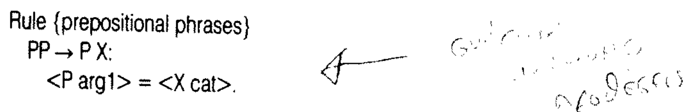
Figure 4.2 Phrase structure tree for sentence with conjunction.

VP category can reintroduce the VP category. The grammar imposes no limit on how many times categories can be reintroduced by the coordination rule, and so there is no limit on the number of expressions that Grammar2 can admit. Hence, Grammar2 admits all of the following examples:

Nurses died.
Nurses died and patients recovered.
Nurses died and patients recovered and Dr Chan slept.
Nurses died and patients recovered and Dr Chan slept and MediCenter employed nurses.
Nurses died and patients recovered and Dr Chan slept and MediCenter employed nurses and ... .

Although Grammar2 allows for the grammaticality of infinitely many sentences, we would rapidly grow bored with an enumeration of them. This is partly because the lexicon for Grammar2 is very small and so lengthy examples will force us into the repetition of lexical items. Let us therefore expand Grammar2 to Grammar3 and, in so doing, enlarge our lexicon by exploiting more fully the technique of using a feature (arg1) to encode the category of complements that we employed for subcategorization in Grammar2. The rules for Grammar3 are those employed in Grammar2 with one addition, a rule for expanding prepositional phrases:

```
Rule {prepositional phrases}
  PP → P X:
    <P arg1> = <X cat>.
```



It is only when we come to the lexicon of Grammar3 that differences really become apparent. Grammar3 allows us to employ a much wider range of distinct subtypes of lexical item, including six different kinds of verb, by augmenting the lexicon of Grammar2.

EXAMPLE: Lexicon3

```
Word approved:
    <cat> = V
    <arg1> = PP.
Word disapproved:
    <cat> = V
    <arg1> = PP.
Word appeared:
    <cat> = V
    <arg1> = AP.
Word seemed:
    <cat> = V
    <arg1> = AP.
Word had:
    <cat> = V
    <arg1> = VP.
Word believed:
    <cat> = V
    <arg1> = S.
Word thought:
    <cat> = V
    <arg1> = S.
Word of:
    <cat> = P
    <arg1> = NP.
Word fit:
    <cat> = AP.
Word competent:
    <cat> = AP.
Word well-qualified:
    <cat> = AP.
```

Grammar3 provides us with structures for all the following examples, as well as an infinity of others:

Nurses thought Dr Chan seemed competent.

Dr Chan appeared well-qualified and disapproved of MediCenter.

Patients had believed nurses thought Dr Chan had slept.

This last example illustrates recursion through the sentential (clausal) or verb phrase complements of a verb – we saw an RTN treatment of this phenomenon in Chapter 3. This is a very common form of recursion found in natural languages.

Our final example grammar will maintain the overall structure employed in Grammar2 and Grammar3. As Lexicon4 will be identical in every respect to Lexicon3, we shall not repeat it here. In fact, the only real change we will make in Grammar4 is the addition of a new feature, which we will call slash in virtue of a notational convention whereby we will use X/Y to represent a category X0 whose cat is X and whose slash is Y; that is, <X0 cat> = X and <X0 slash> = Y. The addition of this feature to the system requires certain consequential additions to the rules. Expressions like S/NP and VP/PP, then, stand for a particular kind of category, but what kind? What is the intuitive content of the notation? The answer is quite straightforward: an expression of category X/Y is an expression of category X from which an expression of category Y is missing. Thus, an S/NP (read S-slash-NP) is a sentence (or clause) that has got a noun phrase missing, whereas a VP/PP is a verb phrase that is missing a prepositional phrase. To make use of, or even to make sense of, these slash categories, we will need to make a number of additions to the rules that Grammar3 came equipped with.

EXAMPLE: Rules for Grammar4

```
Rule {simple sentence formation}
  S → NP VP:
    <S slash> = <VP slash>
    <NP slash> = 0.
Rule {intransitive verb}
  VP → V:
    <V arg1> = 0
    <V slash> = 0
    <VP slash> = 0.
Rule {single complement verbs}
  VP → V X:
    <V arg1> = <X cat>
    <V slash> = 0
    <VP slash> = <X slash>.
Rule {prepositional phrases}
  PP → P X:
    <P arg1> = <X cat>
    <P slash> = 0
    <PP slash> = <X slash>.
```

```
Rule {coordination}
  X0 → X1 C X2:
    <X0 cat> = <X1 cat>
    <X0 cat> = <X2 cat>
    <C slash> = 0
    <X0 slash> = <X1 slash>
    <X0 slash> = <X2 slash>
    <X0 arg1> = <X1 arg1>
    <X0 arg1> = <X2 arg1>.
Rule {topicalization}
  X0 → X1 X2:
    <X0 cat> = S
    <X1 empty> = no
    <X2 cat> = S
    <X2 slash> = <X1 cat>
    <X2 empty> = no
    <X0 slash> = <X1 slash>.
Rule {slash elimination}
  X0 → :
    <X0 cat> = <X0 slash>
    <X0 empty> = yes.
```

The first five rules of Grammar4 are simply revisions of those in Grammar3 to allow for our new feature: in each case, the value of slash (if any) on the mother is equated with the value of slash (if any) on the complement daughter, or all the daughters, in the case of the coordination rule. The sixth and seventh rules are wholly new. The essence of the topicalization rule can be more perspicuously expressed using our slash notation:

S → X S/X.

This rule says that a sentence can consist of some category followed by a sentence which is missing an expression of that category. And, these in turn, taken together with the rest of Grammar4, will permit such sentences as the following (without the commas):

MediCenter, nurses disapproved of _.
Of MediCenter, nurses disapproved _.
Well-qualified, Dr Chan had seemed _.

As can be readily seen, the sentences that follow the comma in these examples all have something missing. The rule that is responsible for this

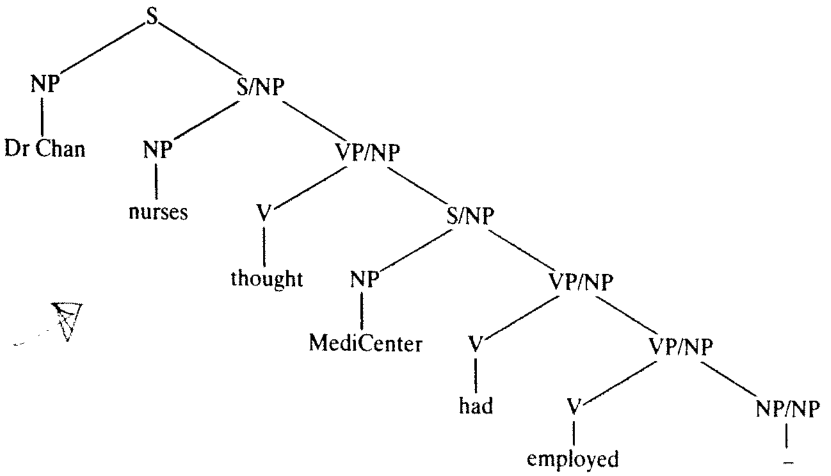


Figure 4.3 Phrase structure tree with slash categories.

missing item in Grammar4 is the last one, which can also be notated as follows:

X/X → .

This just says that an X missing an X can be realized as nothing. Thus, NP/NP, PP/PP and AP/AP are all allowed to appear as nothing in the structures defined by Grammar4.

Note that our modifications to the rules in Grammar3 mean, for example, that a sentence missing a Y can consist of a noun phrase followed by a verb phrase missing a Y (where Y might be NP, say). So, what the modified rules now permit is a transfer of information about a missing category from mother to daughter. The way this works should become clearer by looking at the relevant tree exhibited in Figure 4.3.

There is, in principle, no limit on the amount of intervening material that can occur between the *displaced* constituent at the front of the sentence and the *empty* constituent that corresponds to it and which occurs within the sentence. Such constructions manifest what are known as *unbounded dependencies* and are surprisingly common in the world's languages. English, for example, makes extensive use of this type of construction, most notably in questions and relative clauses. In Chapter 3, we indicated how an ATN treatment might tackle such constructions using a global register HOLD to remember the displaced material. Such a procedural approach, which is specialized to parsing, differs substantially from the declarative description of the slash feature given in Grammar4. Although the analysis of topicalization given here is rudimentary, it does serve to

illustrate the sort of feature-passing technique that is very common in current computational linguistic work on syntax.

The lexicon of Grammar3 is almost adequate for Grammar4, except for one problem. The grammar rules explicitly require that words like verbs and prepositions have 0 as the value of their slash feature. This is to prevent such words being extraposed from their normal position and untopicalized constructions having gaps. As a result, lexical entries need to state their value of slash explicitly, as in:

```
Word died:
    <cat> = V
    <slash> = 0
    <arg1> = 0.
```

It is straightforward, but laborious, to make the necessary alterations to all the lexical entries for Grammar3. The use of *macros* in lexical entries, an idea to be introduced in Chapter 7, is one possible way of making such global assignments of features to lexical items.

Notice that our use of features in Grammar2, Grammar3 and Grammar4 has been very restricted. We have simply replaced monadic category names like S and NP with small finite sets of attribute value pairs, where both attribute and value are drawn from small finite sets of atomic elements. Consequently, all these grammars could be converted into equivalent grammars with monadic categories quite unproblematically, although there are no obvious computational reasons for doing so. For instance, the rule:

```
Rule {simple sentence formation}
  S → NP VP:
    <S slash> = <VP slash>.
```

could be replaced by a finite number of rules like:

```
Rule
  S/S → NP VP/S.
Rule
  S/NP → NP VP/NP.
Rule
  S/VP → NP VP/VP.
Rule
  S/AP → NP VP/AP.
```

where S/S, VP/S, and so on, are to be read as single symbols, just like NP and S. Since the number of categories in these grammars is finite, and the rules are context free and finite in number, we have not left the domain of CF-PSGs, even though the descriptive apparatus used is much more sophisticated than that employed in Grammar1. We consider a mathematically

more far-reaching use of the feature system offered by the PATR formalism at the end of this chapter.

The grammars we have been elaborating are what linguists would call 'toy grammars'. They serve to illustrate something of the way contemporary feature–theoretic grammars work, but they should not be taken too seriously as analyses of English. The fact that Grammar3 and Grammar4 work as well as they do owes a lot to the particular lexicon that they employ. The reader who has mastered the mechanics of at least Grammar3 might usefully consider the questions that follow.

Exercise 4.12 What is the purpose of the feature empty in Grammar4? What syntactic structure(s) would the grammar assign to the sentence 'MediCenter employed nurses' if the equations involving empty were removed? [*intermediate*]

Exercise 4.13 What happens if you try to add the past form(s) of the irregular verb 'go' to the lexicon? Can the analysis of 'had' be maintained in the light of such additions? What fact about the existing choice of verbs makes the treatment of 'had' seem superficially plausible? Does Grammar3, as it stands, permit any ungrammatical sentences containing 'had' to be generated? [*intermediate*]

Exercise 4.14 All the verb forms given are in the past tense. What happens if you try to add the corresponding present tense forms? [*intermediate*]

Exercise 4.15 The NP section of the lexicon does not contain any pronouns. What happens if you try to add 'I', 'him', and so on, to the NP lexicon? Which English pronoun can be added without leading to ungrammatical sentences? [*easy*]

Exercise 4.16 The particular featural treatment of subcategorization adopted in Grammar2 suffers from a fundamental restriction. What is it? What English verbs lie outside its scope? [*easy*]

Exercise 4.17 In Grammar4, there is a generalization to be made about the form of the rules that will transfer slash information from mother to daughter. This generalization is not, however, expressed in the form of Grammar4. What is the generalization? [*easy*]

Being able to find fault in a grammar is one thing, but being able to repair or improve a descriptively inadequate grammar is a whole lot harder. The reader should pursue the following projects in order to get a feel for the latter enterprise.

Exercise 4.18 Grammar2 generates the ungrammatical string '*Dr Chan employed nurses and or and patients'. Work out the structure that it assigns to this string and consider how the grammar might be modified to eliminate it (and other similar examples). [*intermediate*]

Exercise 4.19 Modify Grammar3 so that it can distinguish between a past participle and the past tense form. Add the relevant forms of the verbs 'go', 'eat' and 'know' to the lexicon. Check to see that 'had' now works properly. [*intermediate*]

Exercise 4.20 Augment Grammar3 with a rule that will allow the generation of questions such as 'Had Dr Chan employed nurses?' Estimate how much work would be involved in getting the grammar to allow 'Was Dr Chan employed?' in a principled manner and make a list of the problems such an example gives rise to. [*intermediate*]

Exercise 4.21 Add a feature (or features) to Grammar3 so that it can distinguish plural and singular. Encode this in the NP part of the lexicon in an appropriate manner. Augment the verbal part of the lexicon to include the present tense forms of the verbs. Now modify the rules so as to ensure subject–verb agreement. What problems would you encounter if you tried to add the verb 'be' to your modified grammar? [*intermediate*]

Exercise 4.22 Add a feature (or features) to Grammar3 so that it can distinguish between subject and non-subject noun phrases. Add 'she', 'him', 'they' and 'them' to the NP part of the lexicon. Make sure that the grammar does not legitimate such ungrammatical strings as '*them approved of she'. [*easy*]

Exercise 4.23 Add exactly one rule (and make no other changes) to Grammar4 to enable it to handle examples like 'Dr Chan, nurses thought seemed competent'. [*hard*]

Exercise 4.24 Add a rule (or rules) to Grammar4 so that it can handle two-word noun phrases like 'a doctor', 'the hospital', and so on. Augment the lexicon accordingly. Now add one further rule to permit noun phrases containing simple relative clauses such as 'a nurse doctors had employed' and 'the hospital Dr Chan approved of'. To do this, you will have to employ the slash feature. [*intermediate*]

Exercise 4.25 Augment Grammar4 with the rules necessary to permit questions such as 'Who had MediCenter employed?' and 'Who had Dr Chan thought MediCenter approved of?' [*hard*]

Exercise 4.26 If you have some mastery of a language other than English, try to work out a version of Grammar4 for that language using a lexicon that employs the relevant translations of the words appearing in Grammar4's English lexicon. [*intermediate project*]

4.5 Definite Clause Grammars

Although translating CF-PSGs directly into Prolog clauses is straightforward, there are important new issues to be faced in translating general feature-based grammars. In such grammars, a category is a bundle of features and so cannot be represented simply by an atomic predicate relating strings of words. What we have to do is to translate each category into a predicate together with a sequence of arguments that encode the features in the bundle, as well as saying the right things about portions of the string. As before, it would be useful to have an abbreviated notation that allows us to write such grammars as Prolog programs but without having to worry about the difference lists. Fortunately, many Prolog systems come complete with a built-in interpreter for such a notation, the notation of definite clause grammars (DCGs).

In fact, we have seen a form of DCGs in Section 4.3: the following expressions are DCG rules:

```
s --> np, vp.
np --> [kim].
```

Prolog interpreters equipped with a DCG translator will automatically convert these expressions into the difference list recognition format that we have already discussed. It might seem from this example as if DCG is just a fancy name for a minor variant on standard PSG formalism. But the example is misleading in that the DCG formalism allows us to do something not countenanced in PSG – namely, to associate terms and/or logical variables, as many as you want, as arguments to categories. Hence, the following rule is also a legitimate statement in the DCG formalism:

```
s --> np(Person, Number, nominative),
      vp(Person, Number, tensed).
```

and one which a DCG-equipped Prolog system will translate into something equivalent to this:

```
s(_12, _13) :-
  np(_14, _15, nominative, _12, _16),
  vp(_14, _15, tensed, _16, _13).
```

Assuming that the variables and constants in the original DCG rule are not perversely named, this rule claims that a sentence can consist of a nominative NP followed by a tensed VP, where the NP and VP share the same values for both person and number. In PATR, this would be rendered by something like:

```
Rule
  S → NP VP:
    <NP person> = <VP person>
    <NP number> = <VP number>
    <NP case> = nominative
    <VP vform> = tensed.
```

Note that in the DCG version the number and order of arguments is fixed and crucial. Thus, while **np(third, plural, Case)** is probably both meaningful and useful in the context of the rule just given, the superficially similar **np(third, plural)** and **np(plural, Case, third)** are most likely to be both useless and misleading. (We hedge our claims here, since their truth depends on how the rest of the grammar is set up. We could use **Case** as a variable to range over number values, or use **tensed** as the name of a case, but it would be thoroughly perverse to do so.)

The PATR notation does not suffer from these fixed arity and fixed order requirements on the equations that annotate its rules. Notice also that while we have chosen mnemonic names for the variables in our example rule, these names will typically not be maintained internally by the Prolog system. So, the effect of tracing a parse, say, may be to exhibit expressions like **np(_43, _76, accusative, _15, _16)**, **vp(_43, _56, _97, _16, _19)**, thus requiring us to remember that the third argument in **vp** deals with tense, the second argument in **np** deals with number, and so on. This latter problem can be overcome, albeit at the cost of a little prolixity, by using terms separated by a colon (or '=' or some other symbol that is declared as an infix operator) in place of bare variables. The first component of these terms can then be a constant that provides a mnemonic name for the function of the slot in which it appears, thus:

```
s -->
  np(person : P, number : N, sex : S, case : nominative),
  vp(person : P, number : N, sex : S, verb_form : tensed).
```



```
vp(person : P1, number : N1, sex : S1, verb_form : V1) -->
  xv(person : P1, number : N1, verb_form : V1),
  np(person : P2, number : N2, sex : S2, case : accusative),
  vp(person : P2, number : N2, sex : S2, verb_form : infinitive).
```

And, in a grammar along these lines, the lexical entries need to look like this:

```
np(person : 3, number : singular, sex : _, case : _) --> [kim].
np(person : 3, number : singular, sex : female, case : nominative) --> [she].
det(number : singular) --> [a].
det(number : _) --> [the].
nn(number : _, sex : _) --> [sheep].
nn(number : plural, sex : none) --> [scissors].
bv(person : _, number : plural, verb_form : tensed) --> [are].
tv(person : _, number : _, verb_form : tensed) --> [ate].
xv(person : 3, number : singular, verb_form : tensed) --> [sees].
```

Since DCGs allow the nesting of terms and variables to arbitrary depth, rules such as the following are quite permissible:

```
s -->
  np(person : P, number : N, sex : S, case : C),
  s(slash : np(person : P, number : N, sex : S, case : C)).
vp(person : P, number : N, sex : _, verb_form : V,
   slash : np(person : _, number : _, sex : _, case : accusative)) -->
  tv(person : P, number : N, verb_form : V).
```

Notice that the : notation, although it leads to more perspicuous grammars and traces, *does not* liberate the grammar writer from having to worry about always using the same argument position of a predicate consistently for the same feature (something the PATR user does not have to worry about).

So far, our examples have concentrated on using DCGs as feature-based recognizers, but nothing about the formalism constrains them to mere recognition. Since the arguments to the categories can contain arbitrary terms, we can use them for parsing and, more generally, syntactic translation. Thus, to parse, we could write rules like this:

```
s([s, NP, VP]) --> np(NP), vp(VP).
vp([vp, XV, NP, VP]) --> xv(XV), np(NP), vp(VP).
```

or like this:

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

depending on whether we want the parse returned in the form of a list or a term. And, since there is no limit to the number of permissible arguments, we can, if we wish, simply add this parse construction argument to the feature arguments already illustrated.

The formulation of feature-based grammar in DCG that has been illustrated relies on a fundamental property of the grammars concerned – namely, that there is a feature (we have called it cat) whose value is atomic and always provided in any description of a category given in the grammar. This property has been needed because we have consistently used the cat value as the Prolog predicate in the formulation, and in Prolog it is impossible to have a goal whose predicate is unknown (a variable). Of course, in a rule like:

```
Rule {coordination}
  X0 → X1 C X2:
  ...
```

the whole point is not to have to specify the cat features of X0, X1 and X2. With a grammar having rules of this kind, we have little choice but to treat cat just like every other feature and provide a dummy predicate, **p**, say, that is used with every category. This predicate must always take the same number of arguments (have an argument for every possible feature) so that there can be category descriptions analogous to X0 above that will match with every possible category. The result of translating a PATR grammar of this kind into a DCG will be a set of rules such as the following:

```
p(s, Person1, Number1, Case1, Vform1) -->
  p(np, Person, Number, nominative, Vform2),
  p(vp, Person, Number, Case2, tensed).
```

where the **p** arguments represent cat, person, number, case and vform in that order.

In summary, the DCG formalism represents a rather natural extension of CF-PSG within the Prolog context. It is convenient, being standardly provided with Prolog systems, and perspicuous, it has a semantics that is as declarative as that of Prolog and, of course, it integrates well with Prolog. The arguments associated with categories permit both parsing and translation more generally – for example, translation from English to logic, as we shall see in Chapter 8. Furthermore, as we will see in Chapter 5, the parser that the standard interpretation of the formalism provides is a left-to-right, top-down, backtracking one: a parser, in fact, whose search strategy is exactly that of Prolog itself. For this reason, DCG parsers are fast (since they employ Prolog very directly) but inefficient (since they fail to store intermediate results), although no worse in this respect than a typical ATN of comparable coverage. From the point of view of the grammar writer, the major disadvantage of the DCG formalism is its assumption of 'term unification' as opposed to the 'graph unification' of PATR. This means that all the slots for featural information have to be explicitly provided in a fixed order on every category to which they apply, even when no information is available and the slot just contains a variable. This makes for grammatical and lexical verbosity, whereas formalisms based on graph unification allow unknown or irrelevant featural information to be suppressed. This is a topic we shall be returning to in Chapter 7.

Exercise 4.27 Convert Grammar4 into a **predicate(attribute : value, ...)** style DCG that will recognize exactly the same set of sentences as Grammar4 does. Can you make the predicates the cat values, or do you have to use a different encoding? Write half a side comparing the merits of the two grammars for that language fragment. [*intermediate*]

Exercise 4.28 Incorporating all the **predicate(attribute : value, ...)** rules and lexical entries given as examples in the text, complete the specification of a DCG that will allow you to recognize all the following examples:

kim dies
we see a duck
i am a man
the women who were pregnant are numerous
this duck eats fish that die

but none of these:

we sees a duck
i am these man
the man who was pregnant is numerous
this duck eats scissors who die

and the following example in two quite distinct ways (use **spy** to monitor this):

he saw her duck

[*hard*]

Exercise 4.29 Add an appropriate extra parse construction argument to each of the rules in the DCG completed for the previous exercise, thereby converting it from a recognizer to a parser. [*easy*]

Exercise 4.30 Write a DCG that will translate RTNs written in NATR into RTNs represented in Prolog. [*hard*]

4.6 Classes of grammars and languages

In this section, we are concerned, not with the formalisms that have been employed in grammatical work, but rather with the underlying formal grammars that have been assumed, and with the classes of languages that they induce. The Chomsky hierarchy of languages, illustrated in Figure 4.4, is a crude dimension of grammar power whose lowest rung is the type 3 class of languages, also known as the regular languages or finite-state languages, and whose highest rung is the type 0 class of languages, also known as the recursively enumerable sets. In between come a number of other classes of languages including the indexed languages and the type 2 or context-free languages.

4.6.1 Context-free grammars and languages

The languages that can be described by CF-PSGs, which are the same as the languages that can be described by RTNs, are known as the *context-free languages (CFLs)*. CF-PSGs are relatively simple to write and to modify. They are associated with a successful tradition of work in computation, which has provided us with a thorough understanding of how to parse, translate and compile them. One of the motivations for taking an interest in CF-PSGs is the existence of relatively high-efficiency algorithms for recognizing and parsing CFLs. Context-free parsing is a basic tool of computer science, including NLP, and there have even been proposals for implementing CF-PSG parsers in special-purpose hardware.

Although equivalent in power, CF-PSGs are, in general, to be preferred to RTNs for the description of natural and formal languages. RTNs involve an ontology of states, along with properties of states (initial, final), neither of which are needed in CF-PSGs. And the standard way of thinking about RTNs is strongly conducive to a particular kind of processing ('top-down left-to-right depth-first' in the jargon of Chapter 5), whereas CF-PSG leaves all processing issues completely open. RTNs do have one minor advantage over CF-PSGs in that they can handle iteration directly (by an arc that loops back to the state it started from), whereas CF-PSGs are forced to use recursion to capture iteration.

Introductory linguistics and even AI textbooks and other pedagogically oriented works often claim that such phenomena as the dependencies between non-adjacent words exemplified in subject–verb agreement show English to be non-context free. This is not so. Even finite-state languages can exhibit dependencies between symbols arbitrarily far apart. To take an artificial example, suppose the last word in every sentence had to bear some special marking that was determined by what the first morpheme in the sentence was; a finite automaton to accept the ... could simply encode in its state the information about what the

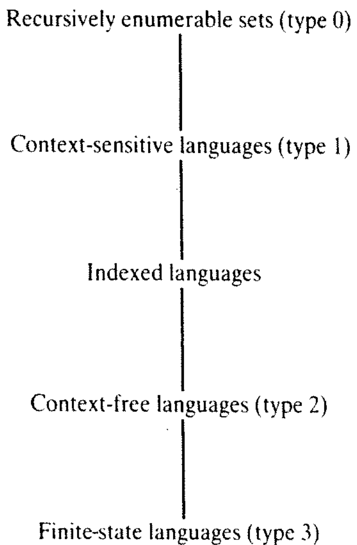


Figure 4.4 The Chomsky hierarchy of languages.

sentence-initial morpheme was, and check the last word's marking against the state before accepting.

However, recently, at least one apparently valid instance of a natural language that is not a CFL has been found – Swiss German. What attitude should NLP research and development work take towards the few pieces of evidence that indicate that natural languages are not all CFLs? The fundamental thing that should be kept in mind is that the overwhelming majority of the structures of any natural languages can be elegantly and efficiently parsed using context-free parsing techniques. Nearly all constructions in nearly all languages can be parsed using techniques that limit the system to the analysis of CFLs.

EXAMPLE: Swiss German

The dialects of German spoken around Zürich, Switzerland, show evidence of a mathematically relevant pattern of word order in certain subordinate infinitival clauses: an arbitrary number of noun phrases (NPs) may be followed by a finite verb and a specific number of non-finite verbs, the number of NPs being a function of the lexical properties of the verbs. In addition, the semantic relations between verbs and NPs exhibit a crossed serial pattern: verbs further to the right in the string of verbs take as their objects NPs further to the right in the string of NPs. The crucial substrings

have the form $NP^m V^n$. In a simple case, where $m = n = 5$, such a substring might have a meaning like:

Claudia watched Helmut let Eva help Hans make Ulrike work.

but with a word order corresponding to:

Claudia	Helmut	Eva	Hans	Ulrike	watched	let	help	make	work
NP_1	NP_2	NP_3	NP_4	NP_5	V_1	V_2	V_3	V_4	V_5

In Swiss German, there is a specific property that makes this phenomenon relevant to the grammar power issue: certain verbs demand dative rather than accusative case marking on their objects, as a matter of pure syntax. This pattern will, in general, not be one that a CF-PSG can describe. For example, if we restrict the situation (technically, by intersecting with an appropriate regular language) to clauses in which all accusative NPs (NP_a) precede all dative NPs (NP_d), then the grammatical clauses will be just those where the accusative-demanding verbs (V_a) precede the dative-demanding verbs (V_d) and the numbers match up; schematically:

$$NP_a^{\,m} \quad NP_d^{\,n} \quad V_a^{\,m} \quad V_d^{\,n}$$

But this schema has the form of the language of strings of the form $a^m b^n c^m d^n$, where n is greater than 0, which can be shown not to be a CFL.

4.6.2 Finite-state grammars and regular languages

What is the smallest known natural class of formal languages that can reasonably be taken to include all the natural languages? The most restricted candidate that has ever been taken seriously is the class of finite-state, regular or type 3 languages. These are exactly the languages that can be recognized by FSTNs as presented in Chapter 2. They can also be characterized by grammars like Grammar1, but subject to the condition that every rule introduces at most one category (non-terminal) in the final position – alternatively, all rules introducing categories must place them in the initial position. Here is a finite-state grammar for the language generated by Grammar1, which happens to be a regular language:

```
Rule
  NP → Dr Chan.
Rule
  S → Dr Chan VP.
Rule
  NP → nurses.
```



```
Rule
  S → nurses VP.
Rule
  NP → MediCenter.
Rule
  S → MediCenter VP.
Rule
  NP → patients.
Rule
  S → patients VP.
Rule
  VP → died.
Rule
  VP → died NP.
Rule
  VP → employed.
Rule
  VP → employed NP.
```

However, there is a valid argument that not all natural languages are regular languages, which is based on the fact that a string of the following form:

A doctor (whom a doctor)^m (hired)^n hired another nurse.

only constitutes a legal sentence of English if m and n are the same. In ordinary grammatical terms, this is because each occurrence of the phrase 'a doctor' is a noun phrase that needs a verb such as 'hired' to complete the clause of which it is the subject.

The fact that natural languages are not regular does not necessarily mean that techniques for parsing regular languages are irrelevant to NLP. A number of scholars over the years have proposed that hearers process sentences as if they were finite automata (or as if they were pushdown automata with a finite stack depth limit), rather than showing the behaviour that would be characteristic of a more powerful device. To the extent that progress along these lines casts light on natural language parsing, the theory of regular grammars and finite automata will continue to be important in the study of natural languages, even though they are not regular languages. Indeed, we have already seen the utility of FSTs in the area of morphology in Chapter 2.

4.6.3 Indexed grammars and languages

If, in addition to a finite set of attribute value pairs, categories can also employ a single recursive feature called stack, say, and if rules are able to

assign to, test and augment this feature, then the resulting expressive power is that of a class of grammars known as the indexed grammars. For example, suppose we allow categories X that can be described in the following way:

```
<X cat> = S,
<X stack top> = a,
<X stack stack top> = a,
<X stack stack stack top> = z.
```

and had a grammar with rules such as:

```
Rule {push endmarker on to stack}
  S → a A:
    <A stack top> = z
    <A stack stack> = <S stack>.
Rule {push 'a' on to stack}
  A0 → a A1:
    <A1 cat> = A
    <A2 cat> = A
    <A1 stack top> = a
    <A1 stack stack> = <A0 stack>.
Rule {copy stack from A to B}
  A → B:
    <A stack> = <B stack>.
Rule {pop 'a' off stack}
  B0 → b B1 c:
    <B0 cat> = B
    <B1 cat> = B
    <B0 stack top> = a
    <B0 stack stack> = <B1 stack>.
Rule {pop endmarker off stack}
  B → b c:
    <B stack top> = z.
```

Then we have a grammar for the language $a^n b^n c^n$, a language that is not a CFL. How have we managed to describe a non-context-free language in the same PATR notation that produced CF-PSGs like Grammar1 and Grammar4? By introducing the recursive feature stack, we have moved to a situation where there are *infinitely many* categories – a situation not permitted in CF-PSGs. Thus, for instance, for every n there is a category X that we can describe by:

```
<X stack^n top> = a
```

and no two such categories are the same.

The languages describable by indexed grammars – namely, the indexed languages – are a natural class of formal languages which form a proper superset of the CFLs and a proper subset of the context-sensitive or type 1 languages. They are of interest to NLP researchers because no phenomena are yet known which would lead them to believe that the natural languages do not form a subset of the indexed languages. In particular, it is clear that indexed grammars are available for the Swiss German facts and for most other sets of facts that have been even conjectured to hold problems for context-free description.

The indexed languages thus provide us, at least for the moment, with a kind of upper bound for natural language syntax. We can no longer be surprised by non-CFL patterns, although their rarity is a matter of some interest, but we should be very surprised at, and duly suspicious of, putatively non-indexed language phenomena.

4.6.4 Writing DCGs for non-context-free languages

In this final section of the chapter we shall briefly illustrate how the use of arguments and variables in DCGs allows them to be used to define recognizers for languages that fall outside the CFL class. As we saw earlier, one of the distinguishing features of a context-free rule is that it places no restriction on the context in which its LHS can be realized as its RHS. There is no reason, however, why we should not consider other kinds of rules in describing languages – for instance, rules like 'LHS can be realized as RHS if it is preceded by an A and followed by a B'. If we allow ourselves to use such rules, then we will be able to describe languages that cannot be descibed by CF-PSGs. One such language is the language $a^n b^n c^n$. Since the rule format for DCGs is that of the CF-PSGs and since there is no apparent reference to syntactic context, it may not be immediately obvious how DCGs are able to provide recognizers for non-CFLs. The key to this puzzle lies in the fact that DCGs can take account of context by recording information in extra arguments to the predicates. We begin by considering a grammar for the $a^n b^n c^n$ language that is standardly used as an example of a non-CFL language. In this grammar, the extra arguments are used to remember how many *b*s and *c*s are to be accepted after the *a*s. This information is stored as a term of the form:

```
i(i(i(... i(i) ...)))
```

where the number of is is the number of symbols to be accepted. The first *a* seen is recognized by the rule:

```
s --> [a], s(i).
```

Every subsequent *a* seen pushes an *i* on to the *i*-stack kept in the category's argument slot, thus:

```
s(I) --> [a], s(i(I)).
```

We are through with *a*s, so we copy the entire *i*-stack, which now records the exact number of *a*s seen, on to two new categories:

```
s(I) --> bn(I), cn(I).
```

and use the first of these to check the length of the *b*-string by popping one *i* off the stack for each *b* seen:

```
bn(i(I)) --> [b], bn(I).
bn(i) --> [b].
```

and then doing exactly the same with the *c*-string:

```
cn(i(I)) --> [c], cn(I).
cn(i) --> [c].
```

Another familiar chestnut from the formal language literature, one that perhaps has more potential relevance to NLP than the case we have just looked at, is the string copying language xx, x in $\{a, b, c\}^*$. This is the language of strings made out of the symbols a, b and c where each string consists of two halves, the second being an exact copy of the first. This succumbs to a rather similar technique: get the first character in the string and put it on the (empty) stack:

```
x --> [a], x(a).
x --> [b], x(b).
x --> [c], x(c).
```

push subsequent characters on to the stack as they occur:

```
x(I) --> [a], x(a(I)).
x(I) --> [b], x(b(I)).
x(I) --> [c], x(c(I)).
```

copy the stack to a new category:

```
x(I) --> y(I).
```

and start popping the characters off, as you encounter them in the string:

```
y(a(I)) --> y(I), [a].
y(b(I)) --> y(I), [b].
y(c(I)) --> y(I), [c].
```


until you reach the last character in the string:

```
y(a) --> [a].
y(b) --> [b].
y(c) --> [c].
```

It is worth reflecting on the efficiency of this grammar used as a DCG recognizer under the standard interpretation of DCGs. Arranged with the order of clauses used here, the program will begin by putting the entire string on the stack, reach the end, backtrack one character, apply the x(I) --> y(I) rule, try and accept the final character, backtrack over the penultimate character, retry the x(I) --> y(I) rule, and so forth, until it has backtracked to the exact middle of the string. Only then will it be able to make it through to the end of the string. Compare this with the operation of the DCG recognizer for $a^n b^n c^n$ given earlier: that does not need to backtrack at all when presented with grammatical strings. The moral is a simple one: the fact that DCGs make it easy to write elegant grammars for such languages does not entail that the recognition process they induce will be efficient.

One problem that has much interested linguists in the last few years, and even attracted the attention of some computational linguists, is that posed by languages with very free word or constituent order. One of the simplest free word-order languages is the language $\{a, b\}^*$ such that $n(a) = n(b)$. This simple language is outside the class of regular languages, but inside the class of CFLs. Here is a DCG version of a CF-PSG for the language:

```
s1 --> [a], y1.
s1 --> [b], x1.
x1 --> [a], s1.
x1 --> [b], x1, x1.
x1 --> [a].
y1 --> [b], s1.
y1 --> [a], y1, y1.
y1 --> [b].
```

This works, but its workings are remarkably opaque for such a simple grammar. Much more perspicuous is the following DCG, which exploits to the full the possibilities offered by using arguments as stacks in order to count occurrences:

```
%
% Start two counters:
x --> x(a, b).
% If there is an a, then increment the b counter:
x(A, B) --> [a], x(A, b(B)).
```

```
% Alternatively, if there is an a, then decrement the a counter:
x(a(A), B) --> [a], x(A, B).
% If there is a b, then increment the a counter:
x(A, B) --> [b], x(a(A), B).
% Alternatively, if there is a b, then decrement the b counter:
x(A, b(B)) --> [b], x(A, B).
% eliminate the non-terminal:
x(a, b) --> [].
```

This section has served to illustrate some of the formal techniques that can be used to handle grammatical phenomena, such as sequences of matching length, isomorphic sequences and very free order, that sometimes prove awkward, or even mathematically impossible, to handle with just the resources of CF-PSG. To the extent that such phenomena show up in the languages of the world, and it is a very limited extent, these techniques are of potential relevance to NLP. We have restricted ourselves to relatively simple formal language examples, since the background necessary to provide grammar fragments from languages like Swiss German (isomorphic sequences) or Djirbal (free order) would take more space than these relatively exotic topics warrant.

Exercise 4.31 Write a DCG for the language $a^m b^n c^m d^n$. [*easy*]

Exercise 4.32 Could any reordering of the clauses in the *xx* grammar avoid backtracking during the recognition of grammatical strings? If so, demonstrate this. [*intermediate*]

Exercise 4.33 Write an *ad hoc* Prolog program that will recognize strings in the *xx* language without backtracking. [*easy*]

Exercise 4.34 Can you define a CF-PSG in DCG format (that is, a DCG where the predicates have no extra arguments) for the language MIX in $\{a, b, c\}^*$, $n(a) = n(b) = n(c)$? This is the language of strings made from the symbols a, b and c where in each string the number of *a*s is the same as the number of *b*s, which is the same as the number of *c*s. [*hard*]

Exercise 4.35 Define a DCG where the predicates can have extra arguments to provide recognition for MIX. Translate your DCG into the PATR grammar notation. [*easy*]

SUMMARY

- Grammar design is a branch of knowledge representation.
- NLP grammars must satisfy both linguistic and computational criteria.
- Gramars define sets of strings and associate structures with them.
- Tree diagrams are used to represent syntactic structure.
- A lexicon associates words with their properties.
- Syntactic categories are bundles of grammatical information.
- Syntactic features can be used to define subcategories.
- PATR combines CF-PSG rules with a powerful feature system.
- The DCG grammar formalism is built into most Prolog systems.
- DCGs supplement CF-PSG rules with arguments and variables.
- Most constructions in most languages can be described by CF-PSGs.

Further reading

The methodological issues involved in the design of NLP grammar formalisms are usefully discussed in Shieber (1985a, 1987). The use of feature–theoretic grammar formalisms for NLP appears to have its origins in the work of Yngve in the 1950s (1958). The best introduction to the PATR grammar formalism is Shieber (1986a), while compilation of other grammar formalisms into PATR is discussed in Shieber (1986b). Hirsh (1988) shows how PATR grammars can be compiled into equivalent DCGs. Pereira and Warren (1980/1986) provide the standard DCG reference but see also Clocksin and Mellish (1981) as well as Pereira and Shieber (1987). DCGs have their origins in the metamorphosis grammars of Colmerauer (1978) and have spawned many progeny of their own, such as extraposition grammars (Pereira 1980), slot grammars (McCord 1980), gapping grammars (Dahl and Abramson 1984), Definite Clause Translation Grammars (DCTG – Abramson 1984), GDL0 (Morishita and Hirakawa 1984), Modular Logic Grammars, (MLG – McCord 1985) puzzle grammars (Sabatier 1985), and restriction grammars (Hirschman and Puder 1986).