8

Phrase-Structure Grammars in Prolog

8.1 Using Prolog to Write Phrase-Structure Grammars

This chapter introduces parsing using phrase-structure rules and grammars. It uses the Definite Clause Grammar (DCG) notation (Pereira and Warren 1980), which is a feature of virtually all Prologs. The DCG notation enables us to transcribe a set of phrase-structure rules directly into a Prolog program.

Prolog was designed from the very beginning for language processing. It has built-in search and unification mechanisms that make it naturally suited to implement formal models of linguistics with elegance and concision. Parsing with DCG rules comes down to a search in Prolog. Prolog recognizes the rules at load time and translates them into clauses. Its engine automatically carries out the parse without the need for additional programming.

Many natural language processing systems, both in academia and in industry, have been written in Prolog. Other languages like Perl, Python, Java, or C++ are now widely used in language engineering applications. However, much programming is often necessary to implement an idea or a linguistic theory. Prolog gets to the heart of the problem in sometimes only a few lines of code. It thus enables us to capture fundamental concepts while setting aside coding chores.

8.2 Representing Chomsky's Syntactic Formalism in Prolog

8.2.1 Constituents

Chomsky's syntactic formalism (1957) is based on the concept of constituents. Constituents can be defined as groups of words that fit together and act as relatively independent syntactic units. We shall illustrate this idea with the sentences:

The waiter brought the meal.
The waiter brought the meal to the table.
The waiter brought the meal of the day.

Phrases such as *the waiter*, *the meal*, *of the day*, or *brought the meal of the day* are constituents because they sound natural. On the contrary, the groups of words *meal to* or *meal of the* sound odd or not complete and therefore are not constituents.

The set of constituents in a sentence includes all the phrases that meet this description. Simplest constituents are the sentence's words that combine with their neighbors to form larger constituents. Constituents combine again and extend up to the sentence itself. Constituents can be pictured by boxed groups of sentence chunks (Figs. 8.1 and 8.2).

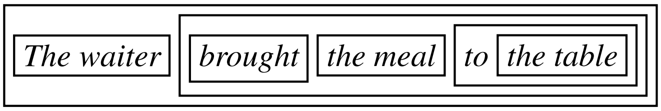

Fig. 8.1. The constituent structure of *The waiter brought the meal to the table*.

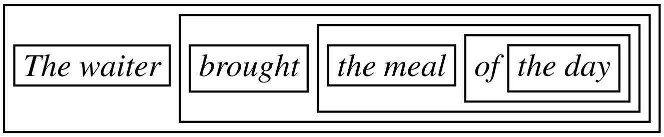

Fig. 8.2. The constituent structure of *The waiter brought the meal of the day*.

In Fig. 8.2, the phrase *the meal of the day* fits in a box, while in Fig. 8.1, *the meal* and *to the table* are separated. The reason is semantic. *The meal of the day* can be considered as a single entity, and so *of the day* is attached to *the meal*. Both can merge in a single constituent and hence fit in the same box. *To the table* is related to the sentence verb rather than to *the meal*: this phrase specifies where the waiter brought something. That is why the next enclosing box frames the phrase *brought the meal to the table* and not *the meal to the table.*

Constituents are organized around a headword that usually has the most significant semantic content. The constituent category takes its name from the headword part of speech. So, *the waiter, the meal, the day*, and *the meal of the day* are noun phrases (NPs), and *brought the meal of the day* is a verb phrase (VP). Prepositional phrases (PPs) are noun phrases beginning with prepositions such as *to the table* and *of the day*.

8.2.2 Tree Structures

Tree structures are an alternate representation to boxes where constituent names annotate the tree nodes. The symbol S denotes the whole sentence and corresponds to the top node. This node divides into two branches that lead to the NP and VP

nodes, and so on. Figure 8.3 shows the structure of *The waiter brought the meal to the table*, and Fig. 8.4 the structure of *The waiter brought the meal of the day*.

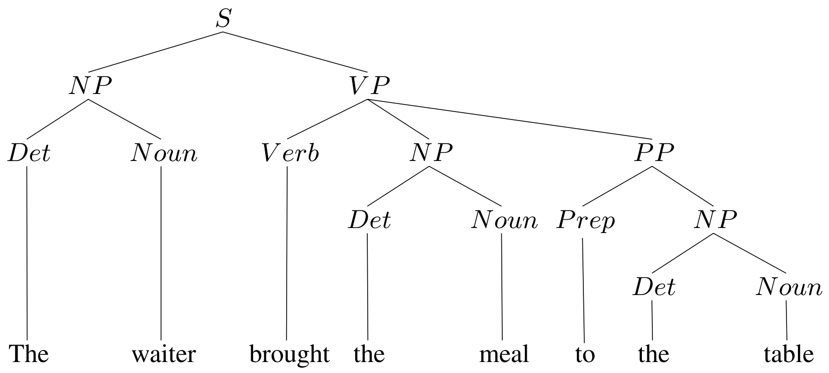

Fig. 8.3. Tree structure of *The waiter brought the meal to the table*.

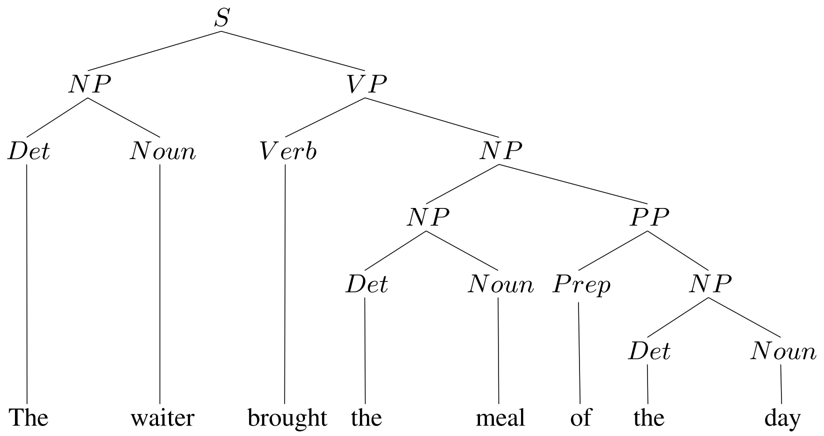

Fig. 8.4. Tree structure of *The waiter brought the meal of the day*.

8.2.3 Phrase-Structure Rules

Phrase-structure rules (PS rules) are a device to model constituent structures. PS rules rewrite the sentence or phrases into a sequence of simpler phrases that describe the composition of the tree nodes. More precisely, a PS rule has a left-hand side that is the parent symbol and a right-hand side made of one, two, or more symbols labeling the downward-connected nodes. For instance, rule

$S \rightarrow NP\ VP$

describes the root node of the tree: a sentence can consist of a noun phrase and a verb phrase.

A phrase-structure grammar is a set of PS rules that can decompose sentences and phrases down to the words and describe complete trees. The phrase categories occurring in Figs. 8.3 and 8.4 are sentence, noun phrase, verb phrase, and prepositional phrase. In the phrase-structure formalism, these categories are called the **nonterminal symbols**. Parts of speech or lexical categories here are determiners (or articles), nouns, verbs, and prepositions. PS rules link up categories to rewrite the sentence and the phrases until they reach the words – the **terminal symbols**. Table 8.1 shows a grammar to parse the sentences in Figs. 8.1 and 8.2.

Table 8.1. A phrase-structure grammar.

Phrases	**Lexicon**	
S → NP VP	*Determiner → the*	*Noun → day*
NP → Determiner Noun	*Noun → waiter*	*Verb → brought*
NP → NP PP	*Noun → meal*	*Preposition → to*
VP → Verb NP	*Noun → table*	*Preposition → of*
VP → Verb NP PP		
PP → Preposition NP		

The first rule in Table 8.1 means that the sentence consists of a noun phrase followed by a verb phrase. The second and third rules mean that a noun phrase can consist either of a determiner and a noun, or a noun phrase followed by a prepositional phrase, and so on. The left constituent is called the **mother** of the rule, and the right constituents are its **expansion** or its **daughters**. The sequence of grammar rules applied from the sentence node to get to the words is called a **derivation**.

8.2.4 The Definite Clause Grammar (DCG) Notation

The translation of PS rules into DCG rules is straightforward. The DCG notation uses the `-->/2` built-in operator to denote that a constituent can consist of a sequence of simpler constituents. DCG rules look like ordinary Prolog clauses except that the operator `-->/2` separates the head and body instead of `:-/2`. Let us use the symbols `s`, `np`, `vp`, and `pp` to represent phrases. The grammar in Table 8.1 corresponds to DCG rules:

```
s --> np, vp.
np --> det, noun.
np --> np, pp.
```

```
vp --> verb, np.
vp --> verb, np, pp.
pp --> prep, np.
```

DCG rules encode the vocabulary similarly. The left-hand side of the rule is the part of speech, and the right-hand side is the word put inside a list – enclosed between brackets:

```
det --> [the].
det --> [a].
noun --> [waiter].
noun --> [meal].
noun --> [table].
noun --> [day].
verb --> [brought].
prep --> [to].
prep --> [of].
```

The Prolog search mechanism checks whether a fact is true or generates all the solutions. Applied to parsing, the search checks whether a sentence is acceptable to the grammar or generates all the sentences accepted by this grammar.

Once the Prolog interpreter has consulted the DCG rules, we can query it using the input word list as a first parameter and the empty list as a second. Both queries:

```
?- s([the, waiter, brought, the, meal, to, the,
table], []).
Yes

?- s([the, waiter, brought, the, meal, of, the,
day], []).
Yes
```

succeed because the grammar accepts the sentences.

In addition to accepting sentences, the interpreter finds all the sentences generated by the grammar. It corresponds to the so-called syntactically correct sentences:

```
?-s(L, []).
L = [the, waiter, brought, the, waiter] ;
L = [the, waiter, brought, the, meal] ;
L = [the, waiter, brought, the, table] ;
...
```

In the grammar above, the two first lexical rules mean that a determiner can be either *the* or *a*. This rule could have been compacted in a single one using Prolog's disjunction operator `;/2` as:

```
det --> [the] ; [a].
```

However, like for Prolog programs, using the semicolon operator sometimes impairs the readability and is not advisable.

In our grammar, nonterminal symbols of lexical rules are limited to a single word. They can also be a list of two or more words as in:

```
prep --> [in, front, of].
```

which means that the word sequence *in front of* corresponds to a preposition.

DCG rules can mix terminal and nonterminal symbols in their expansion as in:

```
np --> noun, [and], noun.
```

Moreover, Prolog programs can mix Prolog clauses with DCG rules, and DCG rules can include Prolog goals in the expansion. These goals are enclosed in braces:

```
np --> noun, [and], noun, {prolog_code}.
```

as, for example:

```
np -->
  noun, [and], noun,
  {write('I found two nouns'), nl}.
```

8.3 Parsing with DCGs

8.3.1 Translating DCGs into Prolog Clauses

Prolog translates DCG rules into Prolog clauses when the file is consulted. The translation is nearly a mapping because DCG rules are merely a notational variant of Prolog rules and facts. In this section, we will first consider a naïve conversion method. We will then outline how most common interpreters adhering to the Edinburgh's Prolog (Pereira 1984) tradition carry out the translation.

A tentative translation of DCG rules in Prolog clauses would add a variable to each predicate. The rule

```
s --> np, vp.
```

would then be converted into the clause

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

so that each variable unifies with the word list corresponding to the predicate name. With this kind of translation and the input sentence *The waiter brought the meal*, variable

- `L` would match the input list `[the, waiter, brought, the, meal]`;
- `L1` would match the noun phrase list `[the, waiter]`; and
- `L2` would match the verb phrase `[brought, the, meal]`.

To be complete, the Prolog clause requires an `append/3` predicate at the end to link `L1` and `L2` to `L`:

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

Although this clause might seem easy to understand, it would not gracefully scale up. If there were three daughters, the rule would require two `append`s, and if there were four daughters, the rule would then need three `append`s, and so on.

In most Prologs, the translation predicate adds two variables to each DCG symbol to the left-hand side and the right-hand side of the rule. The DCG rule

```
s --> np, vp.
```

is actually translated into the Prolog clause

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

where `L1`, `L2`, and `L` are lists of words. As with the naïve translation, the clause expresses that a constituent matching the head of the rule is split into subconstituents matching the goals in the body. However, constituent values correspond to the difference of each pair of arguments.

- *The waiter brought the meal* corresponds to the `s` symbol and unifies with `L1\L`, where `L1\L` denotes `L1` minus `L`.
- *The waiter* corresponds to the `np` symbol and unifies with `L1\L2`.
- *brought the meal* corresponds to the `vp` symbol and unifies with `L2\L`.

In terms of lists, `L1\L` corresponds to `[the, waiter, brought, the, meal]`; `L1\L2` corresponds to the first noun phrase `[the, waiter]`; and `L2\L` corresponds to the verb phrase and `[brought, the, meal]`.

`L1` is generally set to the input sentence and `L` to the empty list, `[]`, when querying the Prolog interpreter, as in:

```
?- s([the, waiter, brought, the, meal], []).
Yes
```

So the variables `L1` and `L2` unify respectively with `[the, waiter, brought, the, meal]` and `[brought, the, meal]`.

The lexical rules are translated the same way. The rule

```
det --> [the].
```

is mapped onto the fact:

```
det([the | L], L).
```

Sometimes, terminal symbols are rewritten using the `'C'/3` (connects) built-in predicate. In this case, the previous rule could be rewritten into:

```
det(L1, L) :- 'C'(L1, the, L).
```

The `'C'/3` predicate links `L1` and `L` so that the second parameter is the head of `L1` and `L`, its tail. `'C'/3` is defined as:

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

In many Prologs, the translation of DCG rules into Prolog clauses is carried out by a predicate named `expand_term/2`.

8.3.2 Parsing and Generation

DCG parsing corresponds to Prolog's top-down search that starts from the **start symbol**, `s`. Prolog's search mechanism rewrites `s` into subgoals, here `np` and `vp`. Then it rewrites the leftmost symbols starting with `np` and goes down until it matches the words of the input list with the words of the vocabulary. If Prolog finds no solution with a set of rules, it backtracks and tries other rules.

Let us illustrate a search tracing the parser with the sentence *The waiter brought the meal* in Table 8.2. The interpreter is launched with the query

```
?- s([the, waiter, brought, the, meal], []).
```

The Prolog clause

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

is called first (Table 8.2, line 1). The leftmost predicate of the body of the rule, `np`, is then tried. Rules are examined in the order they occur in the file, and

```
np(L1, L) :- det(L1, L2), noun(L2, L).
```

is then called (line 2). The search continues with `det` (line 3) that leads to the terminal rules. It succeeds with the fact

```
det([the | L], L).
```

and unifies `L` with `[waiter, brought, the, meal]` (line 4). The search skips from `det/2` to `noun/2` in the rule

```
np(L1, L) :- det(L1, L2), noun(L2, L).
```

`noun/2` is searched the same way (lines 5 and 6). `np` succeeds and returns with `L` unified with `[brought, the, meal]` (line 7). The rule

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

proceeds with `vp` (line 8) until `s` succeeds (line 18).

The search is pictured in Fig. 8.5.

Table 8.2. Trace of *The waiter brought the meal*.

```
1  Call: s([the, waiter, brought, the, meal], [])
2  Call: np([the, waiter, brought, the, meal], _2)
3  Call: det([the, waiter, brought, the, meal], _6)
4  Exit: det([the, waiter, brought, the, meal], [waiter,
         brought, the, meal])
5  Call: noun([waiter, brought, the, meal], _2)
6  Exit: noun([waiter, brought, the, meal], [brought,
         the, meal])
7  Exit: np([the, waiter, brought, the, meal], [brought,
         the, meal])
8  Call: vp([brought, the, meal], [])
9  Call: verb([brought, the, meal], _10)
10 Exit: verb([brought], [the, meal])
11 Call: np([the, meal], [])
12 Call: det([the, meal], _11)
13 Exit: det([the, meal], [meal])
14 Call: noun([meal], [])
15 Exit: noun([meal], [])
16 Exit: np([the, meal], [])
17 Exit: vp([brought, the, meal], [])
18 Exit: s([the, waiter, brought, the, meal], [])
```

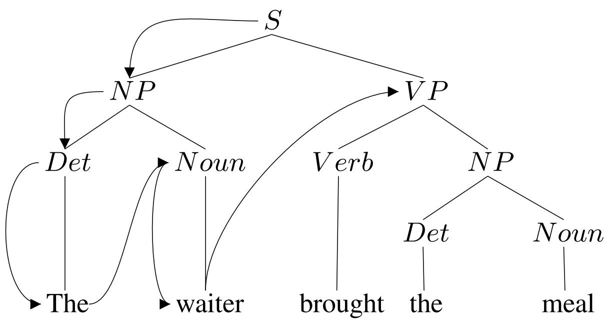

Fig. 8.5. The DCG parsing process.

8.3.3 Left-Recursive Rules

We saw that the DCG grammar in Table 8.1 accepts and generates correct sentences, but what about incorrect ones? A first guess is that the grammar should reject them. In fact, querying this grammar with *The brought the meal* (*) never returns or even crashes Prolog. This is due to the left-recursive rule

```
np --> np, pp.
```

Incorrect strings, such as:

The brought the meal (*)

trap the parser into an infinite loop. Prolog first tries to match *The brought* to

```
np --> det, noun.
```

Since *brought* is not a noun, it fails and tries the next rule

```
np --> np, pp.
```

Prolog calls `np` again, and the first `np` rule is tried anew. The parser loops hopelessly.

The classical method to get rid of the left-recursion is to use an auxiliary rule with an auxiliary symbol (`ngroup`), which is not left-recursive, and to rewrite the noun phrase rules as:

```
ngroup --> det, noun.
np --> ngroup.
np --> ngroup, pp.
```

When a grammar does not contain left-recursive rules, or once left-recursion has been removed, any sentence not accepted by the grammar makes Prolog fail:

```
?- s([the, brought, the, meal, to, the, table], []).
No
```

8.4 Parsing Ambiguity

The tree structure of a sentence reflects the search path that Prolog is traversing. With the rule set we used, verb phrases containing a prepositional phrase can be parsed along to two different paths. The rules

```
vp --> verb, np.
np --> np, pp.
```

give a first possible path. Another path corresponds to the rule

```
vp --> verb, np, pp.
```

This alternative corresponds to a syntactic ambiguity.

Two parse trees reflect the result of a different syntactic analysis for each sentence. Parsing

The waiter brought the meal to the table

corresponds to the trees in Figs. 8.3 and 8.6. Parsing

The waiter brought the meal of the day

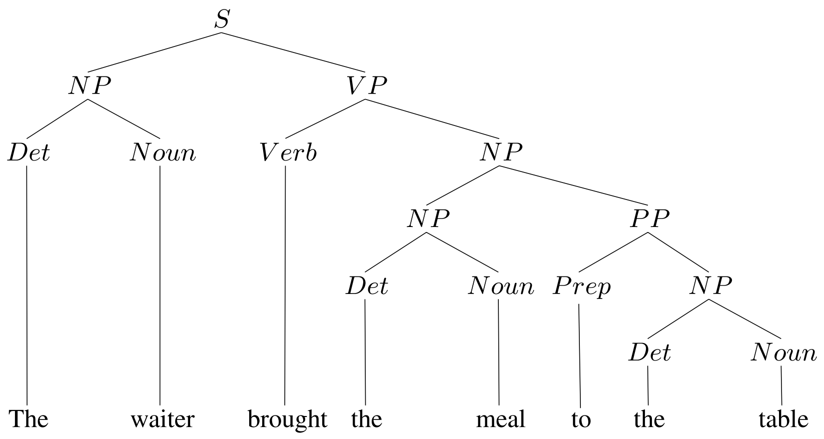

Fig. 8.6. A possible parse tree for *The waiter brought the meal to the table*.

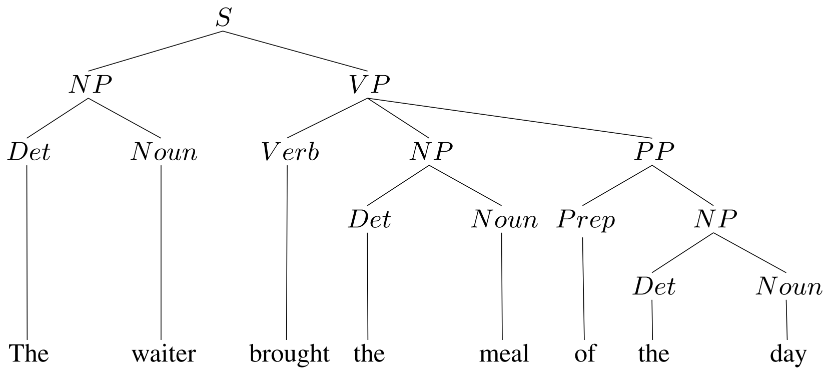

Fig. 8.7. A possible parse tree for *The waiter brought the meal of the day*.

corresponds to the trees in Figs. 8.4 and 8.7.

In fact, only Figs. 8.3 and 8.4 can be viewed as correct because the prepositional phrases attach differently in the two sentences. In

The waiter brought the meal to the table

the object is *the meal* that the waiter brings to a specific location, *the table*. These are two distinct entities. In consequence, the phrase *to the table* is a verb adjunct and must be attached to the verb phrase node.

In the sentence

The waiter brought the meal of the day the verb object is *the meal of the day*, which is an entity in itself. The phrase *of the day* is a **postmodifier** of the noun *meal* and must be attached to the noun phrase node.

When we hear such ambiguous sentences, we unconsciously retain the one that is acceptable from a pragmatic viewpoint. Prolog does not have this faculty, and the parser must be hinted. It can be resolved by considering verb, preposition, and noun types using logical constraints or statistical methods. It naturally requires adding some more Prolog code. In addition, sentences such as

I saw a man with a telescope

remain ambiguous, even for humans.

8.5 Using Variables

Like Prolog, DCG symbols can have variables. These variables can be used to implement a set of constraints that may act on words in a phrase. Such constraints govern, for instance, the number and gender agreement, the case, and the verb transitivity. Variables can also be used to get the result from a parse. They enable us to build the parse tree and the logical form while parsing a sentence.

DCG variables will be kept in their Prolog predicate counterpart after consulting. Variables of a DCG symbol appear in front of the two list variables that are added by `expand_term/2` while building the Prolog predicate. That is, the DCG rule

```
np(X, Y, Z) --> det(Y), noun(Z).
```

is translated into the Prolog clause

```
np(X, Y, Z, L1, L) :-
  det(Y, L1, L2),
  noun(Z, L2, L).
```

8.5.1 Gender and Number Agreement

French and German nouns have a gender and a number that must agree with that of the determiner and the adjective. Genders in French are masculine and feminine. German also has a neuter gender. Number is singular or plural. Let us use variables `Gender` and `Number` to represent them in the noun phrase rule and to impose the agreement:

```
np(Gender, Number) -->
  det(Gender, Number), noun(Gender, Number).
```

To keep the consistency along with all the rules of the grammar, lexical rules must also describe the gender and number of words (Table 8.3).

A Prolog query on `np` with the French vocabulary loaded generates two noun phrases whose determiner and noun agree in gender:

Table 8.3. A vocabulary with gender and number.

French	German
`det(masc, sing) --> [le].`	`det(masc, sing) --> [der].`
`det(fem, sing) --> [la].`	`det(fem, sing) --> [die].`
`det(_, plur) --> [les].`	`det(neut, sing) --> [das].`
`noun(masc, sing) --> [garçon].`	`det(_, plur) --> [die].`
`noun(fem,sing) --> [serveuse].`	`noun(masc, _) --> ['Ober'].`
	`noun(fem,sing) --> ['Speise'].`

```
?- np(Gender, Number, L, []).
Gender = masc, Number = sing, L = [le, garçon];
Gender = fem, Number = sing, L = [la, serveuse];
No
```

In addition to number and gender, German nouns are marked with four cases: nominative, dative, genitive, and accusative. Determiner case must agree with that of the adjective and the noun. To implement the case agreement, let us mark the noun phrase rule with an extra variable `Case`.

```
np(Gender, Number, Case) -->
  det(Gender, Number, Case),
  adj(Gender, Number, Case),
  noun(Gender, Number, Case).
```

Let us also write a small vocabulary:

```
det(masc, sing, nominative) --> [der].
det(masc, sing, dative) --> [dem].
det(masc, sing, genitive) --> [des].
det(masc, sing, accusative) --> [den].

adj(masc, sing, nominative) --> [freundliche].
adj(masc, sing, dative) --> [freundlichen].
adj(masc, sing, genitive) --> [freundlichen].
adj(masc, sing, accusative) --> [freundlichen].

noun(masc, _, Case) -->
  ['Ober'],
  {Case \= genitive}.
noun(masc, _, genitive) --> ['Obers'].
```

Querying `np` with the German vocabulary

```
?- np(G, N, C, L, []).
```

generates four noun phrases whose determiner, adjective, and noun agree in gender and case:

```
G = masc, N = sing, C = nominative,
  L = [der, freundliche, 'Ober'];
G = masc, N = sing, C = dative,
  L = [dem, freundlichen, 'Ober'];
G = masc, N = sing, C = genitive,
  L = [des, freundlichen, 'Obers'];
G = masc, N = sing, C = accusative,
  L = [den, freundlichen, 'Ober'];
No
```

So far, we have seen agreement within the noun phrase. It can also be applied to categorize verbs. Some verbs such as *sleep*, *appear*, or *rushed* are never followed by a noun phrase. These verbs are called intransitive (`iv`). Transitive verbs such as *bring* require a noun phrase after them: the object (`tv`). We can rewrite two verb phrase rules to mark transitivity:

```
vp --> verb(iv).
vp --> verb(tv), np.

verb(tv) --> [brought].
verb(iv) --> [rushed].
```

8.5.2 Obtaining the Syntactic Structure

We used variables to implement constraints. Variables can also return the parse tree of a sentence. The idea is to unify variables with the syntactic structure of a constituent while it is being parsed. To exemplify this, let us use a simplified version of our grammar:

```
s --> np, vp.
np --> det, noun.
vp --> verb, np.
```

The parse tree of

The waiter brought the meal

is reflected by the Prolog term

```
T = s(np(det(the), noun(waiter)),
      vp(verb(brought), np(det(the), noun(meal))))
```

To get this result, the idea is to attach an argument to all the symbols of rules, where each argument represents the partial parse tree of its corresponding symbol. Each right-hand-side symbol will have a variable that corresponds to the structure it matches, and the argument of the left-hand-side symbol will unify with the structure it has parsed. Each rule carries out a part of the tree construction when it is involved in the derivation. Let us consider the rule:

```
s --> np, vp.
```

We add two variables to np and vp, respectively NP and VP, that reflect the partial structure they map. When the whole sentence has been parsed, NP and VP should be

```
NP = np(det(the), noun(waiter))
```

and

```
VP = vp(verb(brought), np(det(the), noun(meal)))
```

When NP and VP are unified, s combines them into a term to form the final structure. This term is s(NP, VP). We obtain the construction of the parse tree by changing rule

```
s --> np, vp
```

into

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

The rest of the rules are modified in the same way:

```
np(np(D, N)) --> det(D), noun(N).
vp(vp(V, NP)) --> verb(V), np(NP).

det(det(the)) --> [the].
det(det(a)) --> [a].

noun(noun(waiter)) --> [waiter].
noun(noun(meal)) --> [meal].
noun(noun(table)) --> [table].
noun(noun(tray)) --> [tray].

verb(verb(bring)) --> [brought].
```

The query:

```
?- s(Structure, L, [])
```

generates all the sentences together with their syntactic structure:

```
Structure = s(np(det(the), noun(waiter)),
   vp(verb(brought), np(det(the), noun(waiter)))),
L = [the, waiter, brought, the, waiter] ;

Structure = s(np(det(the), noun(waiter)),
   vp(verb(brought), np(det(the), noun(meal)))),
L = [the, waiter, brought, the, meal] ;
```

```
Structure = s(np(det(the), noun(waiter)),
   vp(verb(brought), np(det(the), noun(table)))),
L = [the, waiter, brought, the, table]
...
```

8.6 Application: Tokenizing Texts Using DCG Rules

We can use DCG rules for many applications other than sentence parsing, which we exemplify here with a tokenization grammar.

8.6.1 Word Breaking

The first part of a tokenizer takes a character list as an input and breaks it into tokens. Let us implement this with a DCG grammar. We start with rules describing a sequence of tokens (`tokens`) separated by blanks. Blank characters (`blank`) are white spaces, carriage returns, tabulations, or control codes. A token (`token`) is a sequence of alphanumeric characters (`alphanumerics`) or another symbol (`other`). Finally, alphanumerics are digits, uppercase letters, lowercase letters, or accented letters:

```
tokens(Tokens) --> blank, {!}, tokens(Tokens).
tokens([FirstT | Tokens]) -->
  token(FirstT), {!}, tokens(Tokens).
tokens([]) --> [].

% A blank is a white space or a control character
blank --> [B], {B =< 32, !}.

% A token is a sequence of alphanumeric characters
% or another symbol

token(Word) -->
  alphanumerics(List), {name(Word, List), !}.
token(Symbol) -->
  other(CSymbol), {name(Symbol, [CSymbol]), !}.

% A sequence of alphanumerics is an alphanumeric
% character followed by the rest of alphanumerics
% or a single alphanumeric character.

alphanumerics([L | LS]) -->
  alphanumeric(L), alphanumerics(LS).
alphanumerics([L]) --> alphanumeric(L).
```

```
% Here comes the definition of alphanumeric
% characters: digits, uppercase letters without
% accent, lowercase letters without accent,
% and accented characters. Here we only consider
% letters common in French, German, and Swedish

% digits
alphanumeric(D) --> [D], { 48 =< D, D =< 57, !}.

% uppercase letters without accent
alphanumeric(L) --> [L], {65 =< L, L =< 90, !}.

% lowercase letters without accent
alphanumeric(L) --> [L], {97 =< L, L =< 122, !}.

% accented characters
alphanumeric(L) -->
  [L], {name(A, [L]), accented(A), !}.

accented(L) :-
  member(L,
    ['à', 'â', 'ä', 'å', 'æ, 'ç', 'é', 'è', 'ê', 'ë',
    'î', 'ï', 'ô', 'ö', 'œ', 'ù', 'û', 'ü', 'ß',
    'À', 'Â', 'Ä', 'Å', 'Æ', 'Ç', 'É', 'È', 'Ê', 'Ë',
    'Î', 'Î', 'Ï', 'Ô', 'Ö', 'Œ', 'Ù', 'Û', 'Ü']).

% All other symbols come here
other(Symbol) --> [Symbol], {!}.
```

Before applying the tokens rules, we need to read the file to tokenize and to build a character list. We do it with the read_file/2 predicate. We launch the complete word-breaking program with

```
?- read_file(myFile, CharList), tokens(TokenList,
  CharList, []).
```

8.6.2 Recognition of Sentence Boundaries

The second role of tokenization is to delimit sentences. The corresponding grammar takes the token list as an input. The sentence list (sentences) is a list of words making a sentence (words_of_a_sentence) followed by the rest of the sentences. The last sentence can be a punctuated sentence or a string of words with no final punctuation (words_without_punctuation). We define a sentence as tokens terminated by an end punctuation: a period, a colon, a semicolon, an exclamation point, or a question mark.

```
sentences([S | RS]) -->
  words_of_a_sentence(S),
  sentences(RS).
% The last sentence (punctuated)
sentences([S]) --> words_of_a_sentence(S).
% Last sentence (no final punctuation)
sentences([S]) --> words_without_punctuation(S).

words_of_a_sentence([P]) -->
  end_punctuation(P).
words_of_a_sentence([W | RS]) -->
  word(W),
  words_of_a_sentence(RS).

words_without_punctuation([W | RS]) -->
  word(W),
  words_without_punctuation(RS).
words_without_punctuation([W]) --> [W].

word(W) --> [W].

end_punctuation(P) --> [P], {end_punctuation(P), !}.

end_punctuation(P) :-
  member(P, ['.', ';', ':', '?', '!']).
```

We launch the whole tokenization program with

```
?- read_file(myFile, CharacterList),
   tokens(TokenList, CharacterList, []),
   sentences(SentenceList, TokenList, []).
```

8.7 Semantic Representation

8.7.1 λ-Calculus

One of the goals of semantics is to map sentences onto logical forms. In many applications, this is a convenient way to represent meaning. It is also a preliminary step to further processing such as determining whether the meaning of a sentence is true or not.

In some cases, the logical form can be obtained simultaneously while parsing. This technique is based on the principle of compositionality, which states that it is possible to compose the meaning of a sentence from the meaning of its parts. We shall explain this with the sentence

Bill is a waiter

and its corresponding logical form

```
waiter('Bill').
```

If *Pierre* replaces *Bill* as the waiter, the semantic representation of the sentence is

```
waiter('Pierre').
```

This means that the constituent *is a waiter* retains the same meaning independently of the value of the subject. It acts as a property or a function that is applied to other constituents. This is the idea of compositional analysis: combine independent constituents to build the logical form of the sentence.

The λ-calculus (Church 1941) is a mathematical device that enables us to represent intermediate constituents and to compose them gracefully. It is a widely used tool in compositional semantics. The λ-calculus maps constituents onto abstract properties or functions, called λ-expressions. Using a λ-expression, the property *is a waiter* is represented as

$$\lambda x.waiter(x)$$

where λ is a right-associative operator. The transformation of a phrase into a property is called a λ-abstraction. The reverse operation is called a β-reduction. It is carried out by applying the property to a value and is denoted

$$\lambda x.waiter(x)(Bill)$$

which yields

$$waiter(Bill)$$

Since there is no λ character on most computer keyboards, the infix operator ^ classically replaces it in Prolog programs. So $\lambda x.waiter(x)$ is denoted `X^waiter(X)`. λ-expressions are also valid for adjectives, and *is fast* is mapped onto `X^fast(X)`. A combination of nouns and adjectives, such as *is a fast waiter*, is represented as: `X^(fast(X), waiter(X))`.

While compositionality is an elegant tool, there are also many sentences where it does not apply. *Kick* is a frequently cited example. It shows compositional properties in *kick the ball* or *kick the box*. A counter example is the idiom *kick the bucket*, which means to die, and where *kick* is not analyzable alone.

8.7.2 Embedding λ-Expressions into DCG Rules

It is possible to use DCG rules to carry out a compositional analysis. The idea is to embed λ-expressions into the rules. Each rule features a λ-expression corresponding to the constituent it can parse. Parsing maps λ-expressions onto constituents rule-by-rule and builds the semantic representation of the sentence incrementally.

The sentence we have considered applies the property of being a waiter to a name: *Pierre* or *Bill*. In this sentence, the verb *is*, as other verbs of being, only links a name to the predicate `waiter(X)`. So the constituent *is a waiter* is roughly equivalent to *waiter*. Then, the semantic representation of common nouns or adjectives is that of a property: $\lambda x.waiter(x)$. Nouns incorporate their semantic representation as an argument in DCG rules, as in:

```
noun(X^waiter(X)) --> [waiter].
```

As we saw, verbs of being have no real semantic content. If we only consider these verbs, verb phrase rules only pass the semantics of the complement to the sentence. Therefore, the semantics of the verb phrase is simply that of its noun phrase:

```
vp(Semantics) --> verb, np(Semantics).
```

The `Semantics` variable is unified to `X^waiter(X)`, where `X` is to represent the sentence's subject. Let us write this in the sentence rule that carries out the β-reduction

```
s(Predicate) --> np(Subject),
vp(Subject^Predicate).
```

The semantic representation of a name is just this name:

```
np('Bill') --> ['Bill'].
np('Mark'] --> ['Mark'].
```

We complement the grammar with an approximation: we consider that determiners have no meaning. It is obviously untrue. We do it on purpose to keep the program simple. We will get back to this later:

```
np(X) --> det, noun(X).
det --> [a].
verb --> [is].
```

Once the grammar is complete, querying it with a sentence results in a logical form:

```
?- s(S, ['Mark', is, a waiter], []).
S = waiter('Mark').
```

The reverse operation generates a sentence from the logical form:

```
?- s(waiter('Bill'), L, []).
L = ['Bill, is, a, waiter].
```

8.7.3 Semantic Composition of Verbs

We saw that verbs of being played no role in the representation of a sentence. On the contrary, other types of verbs, as in

Bill rushed
Mr. Schmidt called Bill

are the core of the sentence representation. They correspond to the principal functor of the logical form:

```
rushed('Bill')
called('Mr. Schmidt', 'Bill')
```

Their representation is mapped onto a λ-expression that requires as many arguments as there are nouns involved in the logical form. *Rushed* in the sentence *Bill rushed* is intransitive. It has a subject and no object. It is represented as

```
X^rushed(X)
```

where `X` stands for the subject. This formula means that to be complete the sentence must supply `rushed(X)` with `X = 'Bill'` so that it reduces to `rushed('Bill')`.

Called in the sentence *Mr. Schmidt called Bill* is transitive: it has a subject and an object. We represent it as

```
Y^X^called(X, Y)
```

where `X` and `Y` stand respectively for the subject and the object. This expression means that it is complete when `X` and `Y` are reduced.

Let us now examine how the parsing process builds the logical form. When the parser considers the verb phrase

called Bill

it supplies an object to the verb's λ-expression. The λ-expression reduces to one argument, $\lambda x.called(x, Bill)$, which is represented in Prolog by

```
X^called(X, 'Bill')
```

When the subject is supplied, the expression reduces to

```
called('Mr. Schmidt', 'Bill').
```

Figure 8.8 shows graphically the composition.

Let us now write a complete grammar accepting both sentences. We add a variable or a constant to the left-hand-side symbol of each rule to represent the constituent's or the word's semantics. The verb's semantics is a λ-expression as described previously, and `np`'s value is a proper noun. The semantic representation is built compositionally – at each step of the constituent parsing – by unifying the argument of the left-hand-side symbol.

```
s(Semantics) --> np(Subject),
vp(Subject^Semantics).
vp(Subject^Semantics) --> verb(Subject^Semantics).
vp(Subject^Semantics) -->
  verb(Object^Subject^Semantics), np(Object).
np('Bill') --> ['Bill'].
np('Mr. Schmidt') --> ['Mr. Schmidt'].

verb(X^rushed(X)) --> [rushed].
verb(Y^X^called(X, Y)) --> [called].

?- s(Semantics, ['Mr. Schmidt', called, 'Bill'], []).
Semantics = called('Mr. Schmidt', 'Bill')
```

In this paragraph, proper nouns were the only noun phrases we considered. We have set aside common nouns and determiners to simplify the presentation. In addition, prepositions and prepositional phrases can also be mapped onto λ-expressions in the same way as verbs and verb phrases. We will examine the rest of semantics in more detail in Chap. 12.

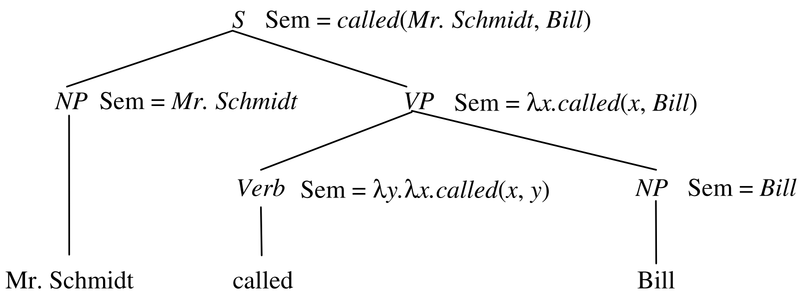
S Sem = called(Mr. Schmidt, Bill)
NP Sem = Mr. Schmidt
VP Sem = λx.called(x, Bill)
Verb Sem = λy.λx.called(x, y)
NP Sem = Bill
Mr. Schmidt
called
Bill

Fig. 8.8. Parse tree with a semantic composition.

8.8 An Application of Phrase-Structure Grammars and a Worked Example

As we saw in Chap. 1, the Microsoft Persona agent uses a phrase-structure grammar module to parse sentences and gets a logical form from them. Ball et al. (1997) give an example of order:

I'd like to hear something composed by Mozart.

that Persona transforms in the logical form:

```
like1 (+Modal +Past +Futr)
   Dsub: i1 (+Pers1 +Sing)
   Dobj: hear1
     Dsub: i1
     Dobj: something1 (+Indef +Exis +Pers3 +Sing)
        Prop: compose1
           Dsub: mozart1 (+Sing)
           Dobj: something1
```

Although Persona uses a different method (Jensen et al. 1993), a small set of DCG rules can parse this sentence and derive a logical form using compositional techniques. To write the grammar, let us simplify the order and proceed incrementally. The core of the sentence means that the user would like something or some Mozart. It is easy to write a grammar to parse sentences such as:

I would like something
I would like some Mozart

The sentence and the noun phrase rules are close to those we saw earlier:

```
s(Sem) --> np(Sub), vp(Sub^Sem).
```

In anticipation of a possible left-recursion, we use an auxiliary `npx` symbol to describe a nonrecursive noun phrase:

```
npx(SemNP) --> pro(SemNP).
npx(SemNP) --> proper_noun(SemNP).
npx(SemNP) --> det, proper_noun(SemNP).

np(SemNP) --> npx(SemNP).
```

The verb phrase is slightly different from those of the previous sections because it contains an auxiliary verb. A possible expansion would consist of the auxiliary and a recursive verb phrase:

```
vp --> aux, vp.
```

Although some constituent grammars are written this way, the treatment of auxiliary *would* is disputable. In some languages – notably in Romance languages – the conditional auxiliary is rendered by the inflection of the main verb, as in French: *j'aimerais*. A better modeling of the verb phrase uses a verb group that corresponds either to a single verb or to a sequence, including an auxiliary to the left and the main verb, here

```
verb_group(SemVG) --> aux(SemAux), verb(SemVG).
verb_group(SemVG) --> verb(SemVG).

vp(SemVP) --> verb_group(SemVP).
vp(SemVP) --> verb_group(Obj^SemVP), np(Obj).
```

The vocabulary is also similar to what we saw previously:

```
verb(Obj^Sub^like(Sub, Obj)) --> [like].
verb(Obj^Sub^hear(Sub, Obj)) --> [hear].

aux(would) --> [would].

pro('I') --> ['I'].
pro(something) --> [something].

proper_noun('Mozart') --> ['Mozart'].

det --> [some].
```

This grammar answers queries such as:

```
?- s(Sem, ['I', would, like, some, 'Mozart'], []).
Sem = like('I', 'Mozart')
```

Now let us take a step further toward the original order, and let us add the infinitive verb phrase *to hear*:

I would like to hear something
I would like to hear some Mozart

The infinitive phrase has a structure similar to that of a finite verb phrase except that it is preceded by the infinitive marker *to*:

```
vp_inf(SemVP) --> [to], vp(SemVP).
```

We must add a new verb phrase rule to the grammar to account for it. Its object is the subordinate infinitive phrase:

```
vp(SemVP) --> verb_group(Obj^SemVP), vp_inf(Obj).
```

The new grammar accepts queries such as:

```
?- s(Sem, ['I', would, like, to, hear, some,
   'Mozart'], []).
Sem = like('I', X^hear(X, 'Mozart'))
```

In the resulting logical form, the subject of *hear* is not reduced. In fact, this is because it is not explicitly indicated in the sentence. This corresponds to an anaphora within the sentence – an intrasentential anaphora – where both verbs *like* and *hear* implicitly share the same subject.

To solve the anaphora and to understand how Prolog composes the logical forms, instead of using the variable `Obj`, let us exhibit all the variables of the λ-expressions at the verb phrase level. The nonreduced λ-expression for *hear* is

```
ObjectHear^SubjectHear^hear(SubjectHear, ObjectHear).
```

When the infinitive verb phrase has been parsed, the `ObjectHear` is reduced and the remaining expression is

```
SubjectHear^hear(SubjectHear, 'Mozart').
```

The original λ-expression for *like* is

```
ObjectLike^SubjectLike^like(SubjectLike, ObjectLike)
```

where `ObjectLike` unifies with the λ-expression representing *hear*. Since both subjects are identical, λ-expressions can be rewritten so that they share a same variable in **`Subject`**`^SemInf` for *hear* and `SemInf^`**`Subject`**`^SemVP` for *like*. The verb phrase is then:

```
vp(Subject^SemVP) -->
  verb_group(SemInf^Subject^SemVP),
  vp_inf(Subject^SemInf).
```

and the new grammar now solves the anaphora:

```
?- s(Sem, ['I', would, like, to, hear, some,
  'Mozart'], []).
Sem = like('I', hear('I', 'Mozart'))
```

Let us conclude with the complete order, where the track the user requests is *something composed by Mozart*. This is a noun phrase, which has a passive verb phrase after the main noun. We model it as:

```
np(SemNP) --> npx(SemVP^SemNP), vp_passive(SemVP).
```

We also need a model of the passive verb phrase:

```
vp_passive(SemVP) -->
  verb(Sub^SemVP), [by], np(Sub).
```

and of the verb:

```
verb(Sub^Obj^compose(Sub, Obj)) --> [composed].
```

Finally, we need to modify the pronoun *something* so that it features a property:

```
pro(Modifier^something(Modifier)) --> [something].
```

Parsing the order with the grammar yields the logical form:

```
?- s(Sem, ['I', would, like, to, hear, something,
composed, by, 'Mozart'], []).
Sem = like('I', hear('I',
          X^something(compose('Mozart', X))))
```

which leaves variable `X` uninstantiated.[1] A postprocessor would then be necessary to associate `X` with *something* and reduce it.

[1] Prolog probably names it `_Gxxx` using an internal numbering scheme.

8.9 Further Reading

Colmerauer (1970, 1978) created Prolog to write language processing applications and, more specifically, parsers. Pereira and Warren (1980) designed the Definite Clause Grammar notation, although it is merely a variation on the Prolog syntax. Most Prolog environments now include a compiler that is based on the Warren Abstract Machine (WAM, Warren 1983). This WAM has made Prolog's execution very efficient.

Textbooks on Prolog and natural language processing delve mostly into syntax and semantics. Pereira and Shieber (1987) provide a good description of phrase-structure grammars, parsing, and formal semantics. Other valuable books include Gazdar and Mellish (1989), Covington (1994), and Gal et al. (1989).

SRI's Core Language Engine (Alshawi 1992) is an example of a comprehensive development environment based on Prolog. It is probably the most accomplished industrial system in the domain of syntax and formal semantics. Using it, Agnäs et al. (1994) built the Spoken Language Translator (SLT) to translate spoken English to spoken Swedish in the area of airplane reservations. The SLT has been adapted to other language pairs.

Exercises

8.1. Translate the sentences of Sect. 8.2.1 into French or German and write the DCG grammar accepting them.

8.2. Underline constituents of the sentence *The nice hedgehog ate the worm in its nest.*

8.3. Write a grammar accepting the sentence *The nice hedgehog ate the worm in its nest*. Draw the corresponding tree. Do the same in French or German.

8.4. The previous grammar contains a left-recursive rule. Transform it as indicated in this chapter.

8.5. Give a sentence generated by the previous grammar that is not semantically correct.

8.6. Verbs of being can be followed by adjective phrases or noun phrases. Imagine a new constituent category, `adjp`, describing adjective phrases. Write the corresponding rules. Write rules accepting the sentences *the waiter is tall*, *the waiter is very tall*, and *Bill is a waiter*.

8.7. How does Prolog translate the rule `lex --> [in, front].`?

8.8. How does Prolog translate the rule `lex --> [in], {prolog_code}, [front].`?

8.9. Write the `expand_term/2` predicate that converts DCG rules into Prolog clauses.

8.10. Write a grammar accepting the sentence *The nice hedgehog ate the worm in its nest* with variables building the parse tree.

8.11. Replace all nouns of the previous sentence by personal pronouns, and write the grammar.

8.12. Translate the sentence in Exercise 8.10 into French or German, and add variables to the rules to check number, gender, and case agreement.

8.13. Calculate the β-reductions of expressions $\lambda x.f(x)(y)$ and $\lambda x.f(x)(\lambda y.f(y))$.

8.14. Write a grammar that accepts the noun phrase *the nice hedgehog* and that builds a syntactic representation of it.

8.15. Persona's parser accepts orders like *Play before you accuse me*. Draw the corresponding logical form. Write grammar rules that parse the order *Play a song* and that build a logical form from it.